Création d'applications multi-écrans à partir d'applications existantes

Mira Sarkis

ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIES
PARIS INSTITUTE OF TECHNOLOGY

TELECOM ParisTech

2016-ENST-0057

EDITE - ED 130

Doctorat ParisTech

T H È S E

pour obtenir le grade de docteur délivré par

Télécom-ParisTech

Spécialité « Informatique et Réseaux »

présentée et soutenue publiquement par

Mira Sarkis

le 4 Octobre 2016

Création Automatique d'Applications Multi-Ecrans à partir d'Applications Web Existantes

Directeur de thèse: **Jean-Claude DUFOURD**
Encadrant de thèse: **Cyril CONCOLATO**

THÈSE

Mme Cécile Roisin, Professeur, Université Grenoble-Alpes	Rapporteur
M. Pablo Cesar, Maitre de Conférence, Centrum Wiskunde & Informatica	Rapporteur
M. Pierre Senellart, Professeur, Télécom-ParisTech	Examinateur
M. Stephan Steglich, Docteur Ingénieur, Institut Fraunhofer FOKUS	Examinateur
M. Cyril Concolato, Maitre de Conférence, Télécom ParisTech	Encadrant
M. Jean-Claude Dufourd, Directeur d'étude, Télécom ParisTech	Directeur

Télécom-ParisTech
Grande école de l'Institut Mines-Télécom - membre fondateur de ParisTech
46 rue Barrault 75013 Paris - (+33) 1 45 81 77 77 - www.telecom-paristech.fr

Abstract

The ubiquity of web applications and the user possession and utilization of multiple devices are major factors for the increased demand for multi-screen applications. Multi-screen applications impose challenges on the application developer and designer especially if existing single-screen applications have to be transformed to the multi-screen environment. Designers should plan the user interface distribution and should adapt the layout for various devices. Developers should re-organize the application logic and associate it to the distributed user interface. They should preserve the application functionality and finally they need to adapt it to the underlying multi-screen platform. In this work, we propose an end-to-end refactoring system. The system allows the re-use of existing single-screen applications to automatically create multi-screen applications. The components of the multi-screen applications have their layout adapted to small and large device and they are ready to operate synchronously to provide a complementary usage experience. Our system is quantitatively evaluated on different sets of applications containing at least one video element and interactive content. The content division of our system corresponds to a ground truth division with an average recall of 0.84. In addition, our layout refactoring approach obtains 60% accuracy on the tested applications. In addition, we evaluate the performance of the run-time behavior of one application and we compute the delays that are caused by our system and by the network in a real physical environment: with a total delay of 5 ms, our solution is realistic.

RESUME

L'omniprésence des applications Web, la possession et l'utilisation simultanée de plusieurs appareils par un seul utilisateur sont les principaux facteurs de la demande accrue pour les applications multi-écrans. La création des applications multi-écrans imposent des défis sur le développeur d'applications et sur le designer, en particulier s'ils réutilisent les applications web existantes. Par exemple, les developpeurs doivent planifier la distribution de l'interface utilisateur et ils doivent prendre en compte la diversité des dispositifs pour mieux présenter le contenu.

En plus, ils doivent re-penser l'organisation du code de l'application afin de préserver la fonctionnalité de l'application et surtout assurer la communication entre les parties distribuées de l'application.

Dans ce travail, nous proposons un système de bout en bout pour le refactoring des applications web. Le système permet la réutilisation des applications existante, mono-écran, pour créer automatiquement des applications multi-écrans. Les parties distribuées des applications générées ont leur mise en page adaptée aux petits et grands dispositifs et ils sont prêts â fonctionner de manière synchrone tout en fournissant des tâches complémentaires.

La performance du système est évaluée quantitativement sur un ensemble d'applications contenant au moins un élément vidéo et du contenu interactif.

Contents

List of Figures

List of Tables

List of abbreviations

MSA	Multi-Screen Application
XML	eXtensible Mark-up Language
HTML	HyperText Markup Language
CSS	Cascading Style Sheets
JS	JavaScript
DOM	Document Object Model
UI	User Interface
GUI	Graphical User Interface
DUI	Distributed User Interface
RWD	Responsive Web Design
FWD	Full-Window Design
RESS	REsponsive Server Side
NSD	Network Service Discovery
UPnP	Universal Plug and Play
WebRTC	Web Real Time Communication
SSDP	Simple Service Discovery Protocol
SOAP	Simple Object Access Protocol
WSDL	Web Services Description Language
SOA	Service-Oriented Architecture
mDNS	multicast Domain Name System
DNS-SD	Domain Name System - Service Discovery
CC/PP	The Composite Capability/Preferences Profiles

Chapter 1

Introduction

1.1 The multi-screen application context and challenges

The rise of web technologies and their continuous evolution, especially with the emergence of HTML5, led to powerful web-based applications. These web applications enable users to access the same content from various devices, thus making ubiquitous applications a reality.

Due to this ubiquity and to the growing number of connected devices available for one user, each of them having different physical features, i.e., screen size and input-output methods, the user consumption for content has changed. A Google study [23] states that 90% of our media interactions are screen-based and that our interactions are no longer limited to one device. Involving multiple devices to consume and to interact with the content of an application defines the multi-screen ecosystem and makes this application a multi-screen application.

Concerning the usage of multi-screen applications, there are three forms of interactions [1,2]:

i) The sequential usage happens when an application is moved seamlessly from one device to another to continue running it. For example, using a PC to select items on a shopping site and moving to a smart phone to pay for the items.

[1] See `http://uxmag.com/articles/designing-for-context-the-multiscreen-ecosystem`

[2] See `https://www.thinkwithgoogle.com/research-studies/the-new-multi-screen-world-study.html`

ii) The multi-tasking usage happens when multiple independent applications are running at the same time over multiple devices. For example, watching news on a TV while checking social media applications on the tablet.

iii) The complementary usage happens when multiple complementary applications are running at the same time over multiple devices to provide one global task. For example, watching a movie on the TV while controlling it using a smart phone or using multiple small-screen devices to construct a bigger display [18].

In the context of this PhD, we focus on the complementary usage. We assume a Multi-Screen Application (MSA) as consisting of multiple components, each having a specific role, running on a distinct device and always communicating with each other to provide the complementary usage. As a matter of simplicity, the web application term is herein referred to as application.

Developing and designing multi-screen applications from scratch for a complementary usage is more challenging than the development of MSA for sequential or multi-tasking usages. Additionally, it is more challenging than the development of traditional single-screen applications (SSA). We have identified these additional challenges for developer as follows:

- Developers have to determine how the application content will be distributed across devices based on their capabilities. This challenge is specific for MSA for complementary usage.
- Developers have to manage the synchronization and consistency of the distributed content. While this content consistency should also be addressed when developing MSA for sequential usage, especially that the application state should be preserved across devices, the synchronization challenge is specific for complementary usage.
- Application designers have to provide a consistent rendering of the content across devices. This challenge is common for all types of applications, especially if the developer wants his application to be used on the main classes of devices (i.e., PC, tablet, smart phones, TV).
- Finally, they have to adapt these applications to the multi-screen platform on which they are going to run, notably to ensure communication. This challenge is absent for SSA and for multi-tasking usage.

The use of web technologies helps reducing the complexity of the developer chal-

lenges listed above, and increases the possibility of deploying ubiquitous applications on any device connected to the web and having a browser, but it is not sufficient.

1.2 Thesis motivations and challenges

The motivation of this thesis is to face/eliminate/reduce these challenges while avoiding that a developer has to intervene at any level of the process of creating multi-screen applications for complementary usage. In fact, we want to conceive a system for authoring automatically multi-screen applications and that is dedicated for end-users and not for specialized users, e.g, developers among others.

While some efforts aim at developing multi-screen applications from scratch, the originality of this PhD is to re-use existing single screen applications to author multi-screen applications.

Multiple problems arise when we try to re-use applications that initially were not intended for the complementary multi-screen usage. These problems are added to the classical MSA developer challenges listed in the previous section.

First, applications are not necessarily developed in a modular manner that facilitates the identification of independent components of content and their distribution.

Second, existing applications were not intended to run on multiple devices with different physical characteristics, especially on mobile devices with a reduced screen size. This implies a non-coherent user experience across devices. A coherent user experience is defined based on Seok [49] as follows: “In spite of the variation of user interfaces provided for each screen, when a user accesses content through the screen, the user experiences remain consistent without any sense of difference, irrespective of the change of devices when user utilizes the contents".

Third, there exist tight relations between the documents that form a web application, i.e., HTML, script logic and styling documents. Though these documents fulfill different purposes, i.e., the HTML document defines the content and its structure, the JavaScript document contains the logic that describes the application behavior and the styling sheet designs the application layout, they can overlap and make the application decomposition a tedious task. Thus, separating the content of an application implies an additional workload on the logic and on the layout if we want to provide the user with a functional multi-screen application and an acceptable layout.

1.3 Thesis context

The work reported in this PhD thesis was partly carried within the context of COLTRAM [22], a 3-year research cooperation between Telecom ParisTech and Fraunhofer FOKUS. The aim of COLTRAM is to design, develop and promote an open cross-device platform for multi-screen services and applications in the heterogeneous Home Network. It features a new model of service as a collaboration of multiple applications running on multiple devices, and where each application is not tied to one device and able to move seamlessly to another device.

In addition, COLTRAM aims at assisting the application authors and developers for the development of multi-screen applications by proposing semi or automatic tools for displaying applications according to the device on which they are running, mapping applications to the best-fit device and finally refactoring existing applications and converging them towards the multi-screen environment in general and to the COLTRAM platform in specific. This PhD fits in that objective.

1.4 Thesis objectives

The main objective of this PhD is to study and address the challenges related to the creation of multi-screen applications, in particular when starting from a single-screen application. The objectives include 1) the creation of the distributed graphical user interface, 2) the layout adaptation to a multitude of devices and to the amount of content on each side of the multi-screen application, and 3) the support of the cross-device synchronization to preserve the complementary usage. Moreover, this PhD work includes the characterization of video-centric applications to identify the presence of a duality between the content and the features of common devices. This is in the objective of mapping the application content to the “best-fit" device on the network. The ultimate objective of this PhD is to conceive an automated system dedicated for end-users and that eliminates the need for the developer intervention.

1.5 Thesis contributions

During this PhD, we came up with a re-factoring system that is guided by the features of the multi-screen environment and that re-uses existing single-screen applications to produce multi-screen applications automatically.

The refactoring system executes during run-time and can be used directly and transparently by an end-user. It assumes the presence of at least two devices discovered in the network. This assumption is crucial since our system exploits the physical features of devices to guide the analysis of the single-screen application and the design of the multi-screen application at last.

The system is dedicated mainly, but not limited, for multimedia applications and specifically video-centric applications that consist of at least one video element.

The contributions of this PhD to the scientific community are six as follows:

- A system dedicated for end-users to create multi-screen applications.
- A hybrid approach for segmenting the graphical user interface of single-screen applications.
- A simple mapping algorithm that analyzes the application content segments and associates them to devices on the network following device features.
- A master-slave model of multi-screen applications consisting of a master component and one or more slave components.
- A tool that re-layouts automatically user interfaces of all components to provide a seamless viewing experience on diverse devices.
- A runtime tool that provides a complementary-task experience for the created multi-screen application during run-time. It consists of synchronizing the components DOM trees to preserve the application main functionality.

This PhD thesis resulted in two papers published in the proceedings of the ACM Symposium on Document Engineering (DocEng).

- The virtual splitter: refactoring web applications for the multi-screen environment[47] in 2014. This paper presents a primitive refactoring system that is the foundation of this dissertation.
- MSoS: a multi-screen-oriented web page segmentation approach [48] in 2015. This paper addresses specifically the processing and the analysis of web applications to partition their user interfaces among multiple devices based on the multi-screen environment.

A journal article was submitted in April 6, 2016 to the Multimedia Tools and Applications (MTAP) journal [3] and is currently under review. It englobes the six contributions listed above, all implemented together and evaluated on a set of existing web applications.

1.6 Organization of the dissertation

Chapter 2 positions the work of this thesis within the literature. We focus on the existing multi-screen platforms, the characterization of the multi-screen ecosystem, the mechanisms used for analyzing and segmenting web documents. In addition, we focus on the existing works related to the content and layout adaptation to different devices (i.e., with different capabilities), notably the responsive web design in contrast to adaptive design. Finally, we conduct a study related to the existing methodologies for content synchronization.

In Chapter 3 we introduce the global system we have developed with its main features, its inputs and the common definitions that will be used across the following chapters. In addition, we describe and characterize the dataset that is used throughout this thesis to validate the system parts. Finally, we define the adopted master-slave model for the output multi-screen application. This is done after conducting a study of the tight relations that exist between HTML and JavaScript and after studying the advantages and drawbacks of multiple possibilities.

Chapter 4 describes in detail the process of using the single-screen application to create the multi-screen application. The chapter starts detailing the different segmentation approaches, i.e., pure structural, pure visual and hybrid, that were developed during this PhD in the purpose of separating the application content. Then, it describes the distribution of these content and their mapping to the master and the slave components. Finally, we evaluate the hybrid segmentation relatively to a ground truth manually created.

The basis of the layout refactoring approach in addition to the main challenges and the solutions we propose are described in Chapter 5. On the one hand, we describe the Full-Window Design algorithm that is dedicated for the master component running on large devices. It aims at eliminating blank spaces on the master. On the other hand, we describe the Responsive Web Design algorithm that is dedicated for the

[3] http://www.springer.com/computer/information+systems+and+applications/journal/11042

slave component(s) running on portable devices. It aims at making the slave layout adapt dynamically to device dimensions. For both algorithms, we define our metrics to evaluate the two algorithms.

Chapter 6 presents details about the synchronization mechanism adopted in our system. This mechanism consists in watching the DOM changes and the user interactions respectively on the master and the slave components and in communicating these changes to the concerned component. Finally, we validate this mechanism on a highly dynamic application to conclude about its performance and the system performance as a whole.

Finally in Chapter 7, we summarize the thesis work and we draw some conclusions concerning the validity of our system in real situations. Finally, we present our future plan for further investigations in the topic.

Chapter 2

State of the art

This chapter covers the review of the multiple tracks investigated in this thesis and it links the adopted solutions to the works in the literature. Section 2.1 describes technologies and protocols behind the cross-device and cross-platform communications in the multi-screen platforms.

In general, a platform is a group of technologies that are used as a basis upon which other applications, processes or technologies are developed [1]. For example, the web browser is considered as a platform since it lets plug-ins run as part of it. Operating systems e.g., Mac OS, Windows, Android, etc. are also platforms. In this thesis, a multi-screen platform is an operating environment upon which various multi-screen application can run, independently from the hardware or the software features of the environment.

Aside form the platform, this chapter reviews the proposed solutions for the creation of multi-screen applications in Section 2.2, the environment characterization in Section 2.3, the segmentation techniques for user interface distribution in Section 2.4, layout adaptation to the physical environment in Section 2.5 and finally the synchronization techniques for web applications in Section 2.6.

[1] Platform, `https://www.techopedia.com/definition/3411/platform`

2.1 Technologies and protocols in multi-screen platforms

The main requirements for a multi-screen platform are to provide solutions for devices and web applications 1) to discover each other in the network and 2) to communicate with each other.

Due to the dynamicity of a multi-screen ecosystem, where devices continuously enter and leave the network, the platform should keep track of device and application availability to ensure a continuous user experience. It should ensure the device "invisibility" in the sense that a user does not have to explicitly configure the devices entering to the network and taking part in the multi-screen experience. In addition, the platform needs to allow HTML interfaces loaded from the Internet to access and to interact with multiple devices in the home network.

Multiple technologies and protocols are considered in multi-screen platforms to satisfy the above requirements and challenges. In the following sections, we mainly focus on those adopted by the COLTRAM project and we describe the main contributions brought by the COLTRAM platform to the multi-screen domain in Section 2.1.3.

2.1.1 Web Services

Web services are a web technology that allows clients to communicate with a server in a platform-independent and programming language-independent manner [2]. A web service is accessed via HTTP and executed on a remote system (a web server), hosting the requested service.

A web service is a unit of work handling a specific task described using a context-free declarative language like XML or JSON. These languages can describe any and all data in a platform independent manner for exchange across systems, thus it provides a universal interface. Adopting these universal interfaces is not enough to let a group of web services understand each other. Indeed, they need to agree on the vocabulary and semantics of the data they are going to exchange. In addition, the communication between devices or software requires that they agree on the technology, standards and protocols for the communication and the discovery.

[2] Web Service, `http://www.ibm.com/developerworks/webservices/newto/service.html`

A group of web services interacting together defines a “web service application" in a Service-Oriented Architecture (SOA)[3]. An SOA describes an entire “system of services" dynamically looking around for each other and getting together to perform some application. Adopting the web service technology to design applications, notably multi-screen applications, promotes task-oriented applications.

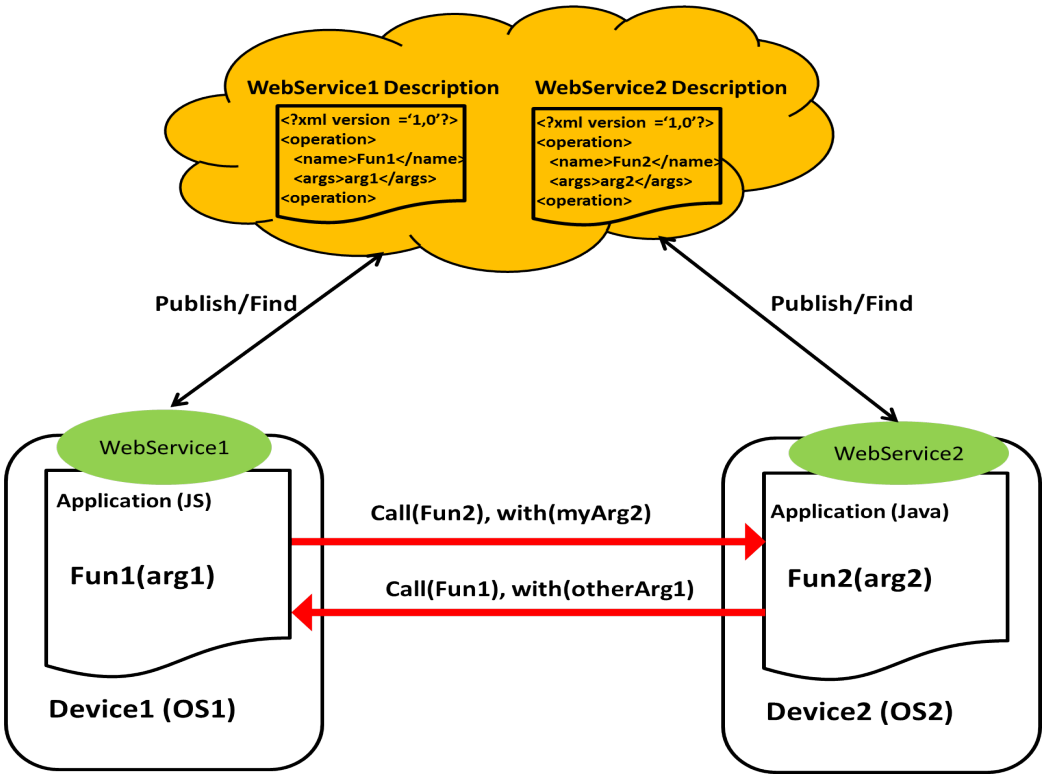


Figure 2-1: Illustrating discovery and communication using Web Service

Figure 2-1 illustrates a web service application made of two web services: WebService 1 and WebService 2. WebService 1 is associated to an application developed in JavaScript and running on Device1. The XML service description comprises a function (i.e., Fun1) that can be called by WebService 2 with one argument (i.e., arg1).

WebService 2 is associated to an application developed in Java and running on Device2. The XML service description comprises a function (i.e., Fun2) that can be called by WebService 1 with one argument (i.e., arg2).

The two descriptions are published and managed on the network. Once webService 1 finds webService 2 and webService 2 finds webService 1, a bidirectional

[3]Service Oriented Architecture`http://www.ibm.com/developerworks/webservices/newto/service.html`

communication channel is established between them.

Once a web service receives an instruction to execute one operation (e.g., Fun2 with myArg2), it maps this instruction to the application code it represents (e.g., a call to the Java function Func2).

Web services use a set of XML/JSON-based formats and protocols in order to:

1. describe 1) the operations to execute or 2) the data to exchange with another web service, 3) the web service location (e.g., using URLs). For instance, Web Services Description Language [4] (WSDL) is an XML format that describes the availability of web services.

2. provide message exchange. The most well-known is Simple Object Access Protocol [5] (SOAP). SOAP is a peer-to-peer protocol for sending and receiving structured XML messages between applications. This protocol encodes messages so they can be delivered over the network using a transport protocol such as HTTP.

3. Advertise and Discover web services: for instance, the Universal Description Discovery and Integration (UDDI) protocol, Simple Service Discovery Protocol (SSDP), multicast Domain Name (mDNS). More information about web service discovery are provided in Section 2.1.2.

The following section focuses on describing in details two discovery systems that include at least an advertising and a discovery protocol.

2.1.2 Service Discovery Systems in multi-device platforms

Discovery is the process by which a client is spontaneously and dynamically notified of the availability of a device or service on the network [55]. In the following, we refer to devices and services as resources.

Resources entering the network register themselves with the discovery system. Depending on the system topology, i.e., using centralized directories or peer-to-peer topology, the registration happens by finding a directory service or by simply sending periodic announcements on the network.

[4] WSDL, `https://www.w3.org/TR/wsdl`
[5] SOAP, `https://www.w3.org/TR/soap/`

During the discovery process, resources provide information that a client will need to access them (e.g.,IP address and port number) and descriptive information such as the resource type (e.g., a gateway device). On the other side, clients provide information about the resource profile they are looking for. The discovery system is responsible for matching the appropriate resources for the client. Examples of discovery systems are Universal Plug and Play (UPnP) [2], Zeroconf [6], Bluetooth, Jini [7], etc.

In the following sections, we focus on the description of UPnP and Zeroconf peer-to-peer systems since they are used in the current sample implementations of Network Service Discovery (NSD)[8] specifications and they are at the basis of the COLTRAM platform.

NSD was proposed to fill the gap related to the absence of a unified way for the web to discover local HTTP-based services. The NSD API enables web pages to discover and to communicate with devices advertising themselves on the network via different discovery protocols in a peer-to-peer configuration. It abstracts away the underlying complexity of Service Discovery protocols and returns back for instance, one or more UPnP or Zeroconf services in the network via a single API call. NSD was investigated in the W3C committee but it is now stopped for fingerprinting concerns.

2.1.2.1 UPnP

Universal Plug and Play is a set of protocols for service discovery defined by a consortium of industrial vendors led by Microsoft.

UPnP builds upon the existing web technologies, such as HTTP, SOAP and XML to provide access to devices and to services in addition to their corresponding descriptions. These web technologies are employed not only to cover Discovery through the use of Simple Service Discovery Protocol (SSDP), but also to cover 5 additional areas, i.e., Addressing, Description, Controlling, Eventing and Presentation [2].

"Addressing“ corresponds to the mechanism for a device to get an IP address. IP addresses are usually provided using DHCP servers. In the absence of such servers, UPnP uses the Auto-IP protocol to automatically generate non-routable IP addresses, especially for home networks.

[6]ZeroConf, `http://www.zeroconf.org/`
[7]Jini, `http://www.sun.com/software/jini/specs/jini1.2html/jini-title.html`
[8]NSD, `https://www.w3.org/TR/discovery-api/`

"Discovery“ is realized by the implementation of the peer-to-peer SSDP protocol defined on top of HTTP. In UPnP, devices and services are characterized using two attributes: type (expressed in the form of URI), and IDs. These attributes are useful for the SSDP discovery as we are going to detail.

SSDP exploits link-local multicast to let clients discover resources directly in a peer-to-peer manner. Once a device joins the network and gets an IP, it sends a presence message in the form of HTTPMU (HTTP over multicast User Datagram Protocol) that is received by all parties listening to the local link. A client sends discovery requests as well over HTTPMU where it specifies its IP address and its port number. Devices respond to this specified address. This response and the presence message both contain: a URI identifying the resource type, its ID, and a URL referring to the device description document.

"Description“ allows the client to get and to investigate the device features (model, serial number, the services it offers, URLs for communication with the device). The device features are obtained from the device description document coded in XML.

The client examines the device description document. If it decides to interact with the services provided by that device, the client sends a control message to that device using SOAP that runs over TCP. The control message contains action names, argument names and variable names specific to the service in question. This step covers the "Controlling“ step.

To be informed of important state changes, UPnP clients subscribe to the event services offered by a device. This mechanism makes up the "Eventing“ portion of the UPnP specification and is handled by the GENA (General Event Notification Architecture) protocol [2]. Eventing consists of notifications of state changes.

Finally, the "Presentation“ aspect of UPnP allows devices to define a presentation URL, which is the location of an HTML document which provides a graphical interface to the device (e.g., a "virtual“ remote control).

2.1.2.2 Bonjour

Bonjour is the Apple implementation of the Zero-configuration networking protocol (Zeroconf) that includes address assignment, host name resolution and service discovery. In contrast to UPnP, Bonjour does neither provide a service description nor propose a means of communication between the discovered services. Thus, in practice it has to be extended to cover these two areas. Bonjour aims at enabling devices to

work together without end-users having to configure the network or having to use IP addresses to refer to machines.

Bonjour combines multicast Domain Name System (mDNS)[9] with DNS Service Discovery (DNS-SD) [10] to leverage existing Internet protocols.

mDNS resolves host names to IP addresses in a small network without the necessity to include a DNS server. Instead, it is the responsibility of the hosts to resolve the names. mDNS defines a new top-level domain, i.e., .local, and it supposes that names ending with .local are analogous to link-local IP addresses (such as 169.254.x.x/16). When an mDNS client needs to resolve a host name, it sends an IP multicast query for a name ending with .local to a special multicast address. Parties that are listening to that address and that can resolve the name respond with their addresses.

To browse the network for services, Bonjour uses DNS-SD that is a set of naming conventions describing how services will be represented in DNS records. Clients use DNS-SD by issuing a request containing the service types they are looking for. As a response, a list of service instance names matching the query is returned. Service names are user-friendly in DNS-SD that considers them as the canonical names for the services, e.g., PrinterFloor5.local. This naming approach is the main difference between DNS-SD and the other discovery systems that only use service IDs to disambiguate services.

2.1.3 The COLTRAM platform

In this thesis, we adopt COLTRAM as a platform under which our multi-screen applications are going to run. We focus our work on the creation and the design of multi-screen applications.

COLTRAM promotes an application platform for Multiscreen services and applications. It provides a unified protocol addressing service discovery, advertising and control. The aim of COLTRAM Discovery and Advertising Framework is to make existing standards and technologies for service discovery available for COLTRAM applications. The network service protocols that are used in COLTRAM are UPnP and Bonjour. In this section, we provide a brief description of the COLTRAM main features and architecture.

In COLTRAM, a multi-screen application consists in multiple components called

[9]mDNS, `http://www.multicastdns.org/`
[10]DNS-SD, `http://www.dns-sd.org/`

atoms. Atoms are able to discover, bind and control remote web services in the local network on the one hand, and can offer and expose web services to other applications or services on the other hand. Web Services are the only interface between atoms and they allow peer-to-peer communication between them.

The particularity in COLTRAM is that services are defined inside the atoms, thus the service discovery and advertising are done at the level of the atoms in the browser using an extended version of the Network Service Discovery API (NSD). Atoms com-

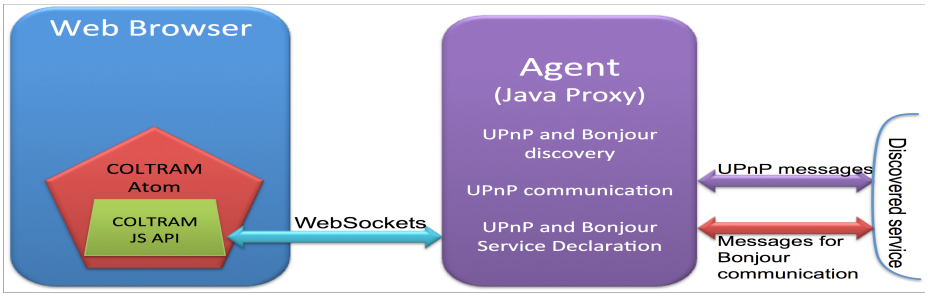


Figure 2-2: COLTRAM Discovery, Communication and architecture

municate with the COLTRAM agent as illustrated in Figure 2-2. Together, they are at the heart of the COLTRAM platform and they provide advertising, discovery, messaging, synchronization, atom distribution, resource sharing, etc.

For each exposed service including Bonjour and UPnP services, COLTRAM generates a unified description as a JSON file. This description abstracts from the underlying service discovery protocols, thus allowing services of different types (UPnP and Bonjour) to communicate.

The COLTRAM descriptions (JSON files) are then sent to the COLTRAM agent using HTML5 WebSockets [11] as Figure 2-2 shows. The COLTRAM agent is an entity that manages the COLTRAM service declaration, discovery and communication.

The webSockets API [45] was developed as part of HTML5 as a JavaScript interface. It defines a bi-directional connection, i.e., a full-duplex single socket connection, between a client and a server. Once the client and the server have opened a WebSocket connection, both endpoints can asynchronously send data to each other at any moment. The WebSocket protocol does not follow the traditional request-response convention, but instead it allows pulling and pushing information between both endpoints. The connection remains open and active until either client or server closes the connection.

[11] WebSockets, `http://www.websocket.org/aboutwebsocket.html`

In COLTRAM, the agent interprets these descriptions and according to the service type, i.e., UPnP or Bonjour, it translates them into the corresponding service description (XML file for UPnP and a url to a JSON file for Bonjour, with the required information). The service discovery and advertising is performed similarly to Sections 2.1.2.2 and 2.1.2.1 between the COLTRAM agent and the network.

Once the COLTRAM agent discovers a service or detects the disappearance of a service, it notifies directly the concerned atoms. The communication between the web services passes necessarily by the COLTRAM agent.

2.2 Frameworks for MSA creation

We have identified multiple approaches in the literature, that propose solutions for the creation of multi-screen applications.

In prior work [25], a 'WebSplitter' was proposed to split web applications, based on a metadata file. This file is unique for each application and determines which application portions 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 splitter architecture is centralized and requires a manual mapping for each element of the application.

In his research, Cheng [13] 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 virtual browser that is a proxy between the web server and the browsers on end-user devices.

Bassbouss et al. [4] outlined how to enable traditional applications to become multi-screen-ready. The application is developed as a single-screen application but 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.

Panelrama [57] is a client-server framework for the construction of distributed user interfaces. The framework includes device categorization, state synchronization and automatic distribution. The content distribution is automatic but it requires

an explicit intervention from the developer on multiple levels to manually divide the graphical user interface. First a developer should select the relevant DOM elements that form a block and then to wrap them in individual templates and second he should characterize the need of each block for device characteristics.

Zorrilla et al. [60] proposed an architecture for distributing a single-screen application over multiple devices while offering users coherent experiences across devices. The architecture decides the best configuration for application visualization through a dynamic set of devices based on some hints provided by the application developer. The applications they consider here should be formed with logic components, i.e., web components [12] and each component should be characterized with hints. These hints provided by the author describe the targeted behavior of the application in a dynamic multi-device environment, but they do not carry information about the context or the targeted devices. It is up to the distributing architecture to map the components to available devices.

In contrast to [13] and [25] our system has a decentralized architecture. Similar to [4] it delivers master-slave applications. The common point for the above works in addition to [57] is that they require a single development environment that facilitates the creation of multi-screen applications. But this means that each single-screen application should be either designed in a modular way [60], or analyzed by the developer to identify the different modules or blocks that form it. There is no an automatic and generic analysis method that can be applied to a large set of existing applications for the application analysis and distribution. Our work provides such method.

2.3 Exploiting the environment

In a multi-screen context, application distribution should be guided by the available devices and their characteristics. To do this, there are two steps: identifying and characterizing the environment and using that characterization.

In this section we present 1) the standardization efforts aiming at describing the environment, i.e., devices, and 2) the works in the literature that exploit the environment description for application distribution.

[12] Web components, `http://webcomponents.org/`

2.3.1 Standards for device description

Standardization efforts were conducted by the W3C forum and the WAP Forum to define common interfaces for device features. The purpose behind device description is to provide customized content delivery, i.e., depending on the device features a different content is sent. These efforts are specific for web applications in general.

W3C proposed Composite Capability/Preferences Profiles (CC/PP) [13] and WAP created the User Agent Profile (UAProf) [14] that is a concrete implementation of CC/PP but dedicated for Wireless Application Protocol (WAP) phones.

CC/PP defines a way to specify the capabilities of a mobile user agent and the user preferences as a collection of Uniform Resource Identifiers and Resource Description Framework (RDF) [15] descriptions. RDF provides a model for describing resources that have properties or attributes and characteristics. RDF itself uses XML for its syntax and for its serialization across networks. In another term, RDF provides the vocabulary while CC/PP provides a common structure for any vocabulary. The mobile vendors are responsible for authoring these reference profiles that are stored on their servers.

Among the identified features, there are the device hardware characteristics (i.e., type, model number, display size, input/output methods, etc.), the device operating system, the network infrastructure, the browser software, etc.

In addition, CC/PP uses a mechanism for sending the device profile to the content server along with a web page request. The client browser adds to the HTTP request a header containing a URL for the device profile. When a content server receives an HTTP request containing a CC/PP reference, it processes the request, follows the URL to the profile, and uses the information it finds there to format content that is suited to the device and to the user preferences, all automatically.

2.3.2 Processing the environment features

Zorrilla et al. [60] use a set of rules to distribute application user interfaces based on device features. The main rules are as follows. In general, dimensions, screen size, means of interaction and the content type are the main factors that influence the distribution. Concerning the TV, they consider that the amount of textual content

[13] CC/PP, `https://www.w3.org/TR/CCPP-struct-vocab/`
[14] Wireless Application Forum, `http://www.wapforum.org/`
[15] RDF,`https://www.w3.org/RDF/`

should be limited; application parts that are not suitable for TV should be avoided, because of the limited interactivity of a remote control; content to be shared among multiple users should be presented on the TV. Concerning mobile devices, they consider that the essential interactive parts should be rendered there to control media elements on remote devices; personal information should remain on personal devices.

Yang and Wigdor [57] model devices and the application parts (or blocks of contents) by considering first the device characteristics, i.e., device resolution, pixel density, physical screen size and means of interaction, where some of the characteristics need to be hand-coded since it is difficult to detect them automatically. Second, a score is provided by the developer to express the relative importance of a device characteristic for the usability of a block of content. Then, a cost matrix is calculated to relate devices to blocks and a linear optimization function is then applied to get the optimal block distribution model. On the appearance or removal of a new device or a new block, the matrix is re-calculated and re-optimized for the new distribution. While this approach looks optimal, it requires that the developer knows in advance the application blocks, that he characterizes each block and describes the block requirements in terms of device features using scores.

Similar to [60], we mainly focus on the device characteristics that affect the usability of a block of content. We also focus on keeping the interactive content on mobile devices and media content on the TV. To this end, we start by analyzing and characterizing the application content to assist first the identification of different parts of the single-screen application and to the content distribution in a second step.

2.4 Web application analysis methods

To identify the different blocks that form an application, that are self-contained and independent from each other, segmentation techniques are used in the literature to segment the user interface of an application. These blocks will be then distributed across devices discovered in the network.

Multiple approaches exist for segmenting a web page. Structure-based segmentation approaches [54][28][24][34] consider only the DOM tree. These approaches suffer from limitations since they do not consider the visual aspects of the application and that the DOM tree does not necessarily correspond to what is rendered by the browser. On the contrary, visual approaches [43][41][9][58] consider the rendered page without necessary considering the DOM tree. Usually, these approaches are ex-

pensive in terms of processing because they require rendering the web page and they often require processing the document as an image using signal processing techniques. There are also content-based segmentation approaches [30][11] where only the textual information is considered to segment the page into sections of different semantics. The problem here is that the employed techniques are not effective on pages with limited textual content.

Hybrid approaches combine multiple criteria to analyze and to segment an application. For instance the hybrid VIPS [8], based on the joint DOM and visual analysis, utilizes both structural information in the DOM tree and visual cues to semantically segment a page. The hybrid Block-o-Matic (BoM) platform [46], based also on the joint DOM and visual analysis, additionally abstracts the segmentation from the DOM tree and works at higher levels, called logical trees without pixel rendering. This abstraction facilitates the understanding and the processing of the page structure. Though the processing of BoM is automatic, its configuration with a granularity parameter (pG) is manual and has to be tailored for each page. Configuring BoM with an inadequate pG value leads to a page not correctly segmented, and applying BoM with the same pG value on a heterogeneous page does not always create coherent blocks similarly on the whole page.

After identifying the application blocks, some segmentation works in the literature proceed with a characterization of these blocks. Sanoja and Gançarski [46] assign labels for each resulting block, that are not relevant for our multi-screen environment, e.g., header, content, image, logo, etc.[46]. A function-based object model (FOM) for website adaptation is introduced by Chen et al. [12]. The segmentation model defines a block as a set of information that have a specific function, i.e., information, navigation, interaction, decoration or others. In FOM, even if a function reflects the intention of the author for using this object, it does not reflect the type of interaction with the end-user.

In our work, we reuse the hybrid approach and the abstraction model proposed by Sanoja and Gançarski [46] but we adapt the segmentation to make it completely automatic and multiscreen-oriented. Additionally, our approach reuses the idea of identifying the block functions from the page content as in FOM, but we define functions from the end-user perspective and not from the author perspective.

2.5 Adaptive web applications and Responsive Web Design

Changing the context-of-use for an application towards the multi-screen environment can cause an unpleasant experience for the end-user especially at the level of the graphical user interface. This is mainly due to the fact that single-screen applications are designed for a specific amount of content and for a specific device at a time.

In a multi-screen environment, there are multiple elements that make the layout design of an application a challenging task. First, depending on the available devices in the network we cannot know in advance the device on which the application is going to run. Second, with the migration service [20] provided in a multi-screen platform, an application can move seamlessly from one device to another. This requires maintaining or adapting the layout to the new target device. Third, the amount of content that is associated to each device is not predictable during the design process and it highly depends on the segmentation results. Thus, there is a need for a layout design that adapts to the dynamicity of the multi-screen environment, i.e., new devices appearing or disappearing in the network, and that adapts to the abrupt change in the amount of content on the web page.

2.5.1 Adaptive Design Features

Adaptive Design is a global concept which aims at designing web applications that adapt in terms of form, function and accessibility to the environment changes. The environment includes the application context-of-use (i.e., device features, browser features, time, bandwidth, user preferences etc.) and the end-user disabilities (e.g., deaf, blind, etc.). The particularity of Adaptive Design is that it is paranoid about the environment capabilities and it makes no assumptions. During runtime, the environment is first detected and characterized. Depending on the environment the adequate design, layout and/or content are sent to the client. The main principles of the Adaptive Design are to the ensure ubiquity, flexibility, performance and enhancement for web applications.

In this section, we present our analysis for the main features/approaches that characterize the different strategies adopted in Adaptive Design. These features answers the following questions: “what to adapt?", “where and how does the adaptation happen?" , “how to adapt?".

2.5.1.1 What to adapt?

The Adaptive Design (AD) includes not only the layout adaptation, but the content adaptation as well.

The layout adaptation consists in adjusting the content dimensions and positions to fit the device screen size, to highlight the most relevant content on a web page or to make these content readable and accessible even on small-screen devices. For instance, considering an airline website that can be used to search and buy plane tickets, to check user reservations, to do the online check-in, etc. On the desktop, all these utilities are present on the website. But a user, having a smart phone, will be mostly interested in checking her reservations or doing the online check-in on-the-go. One strategy here consists in hiding the search and buying utilities from smart phone users. A complementary strategy can be to rotate the horizontal menu to get a vertical visualization in the aim of enhancing the readability and the accessibility for each of the reservation checking and the online check-in utilities.

The content adaptation, including multimedia content, consists in selective content delivery based on the user profile, on the device profile and on the browser. A user profile is related to the user interests and preferences. It can be built based on collecting demographic information, information interests, browsing history (e.g., recently visited web sites, visiting frequency, etc.), access privilege referring to the information a user with little or more right can access, etc. [59]. For instance, we consider a user that uses the airline website to search for a ticket from Peru to Lebanon for summer vacations in August. If two days later she opens the same website on the same device, the website recognizes the user and exploits her browsing history and by default proposes the same journey.

The device profile as defined in 2.3.1 is used to support device-related adaptation. It specifies the MIME types and physical characteristics of a device including color depth, screen size, memory size, operating system, as well as supported markup languages [59].

For instance, we consider that our airline company designed two different website versions: one full-version for desktop users and one optimized version for mobile users. When loading the airline website on a smart phone, e.g., iPhone 6S Plus, the server identifies the device type and in consequence sends the mobile version to the iPhone.

The adaptation of multimedia content (for example, images, audio and video) considers the network bandwidth, the decoding power of a device, the supported

formats (e.g., png, jpeg, esp, etc. for images, ogv, gif, avi, mp4, etc. for videos) and the memory capacities (in particular for video). To this end, the device profile and the network state are tested to choose the suitable parameters for encoding the multimedia content.

In this thesis, we are mostly concerned about the layout adaptation to the browser window size. We do not want to generate new content for our multi-screen applications especially given that our proposed solution is a client-side system.

2.5.1.2 How does the adaptation take place?

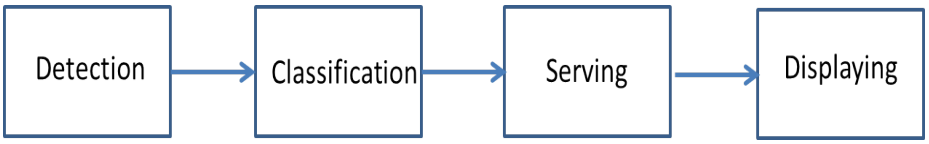


Figure 2-3: Adaptive Design Process

A typical adaptation process consists of four main phases as shown in Figure 2-3.

It starts with detecting or constructing the user profile and the device profile. Usually this phase is called Browser sniffing or Device Detection. Then, the profile is analyzed and classified in order to select or generate the suitable content and layout. The Device Detection, the classification and the serving can happen on either the client-side or on the server-side or on both sides.

In this section, we focus on the different approaches for device and browser detection. There are two approaches: client-side and server-side. In general the client-side approach is based on scripts or on native methods offered by the user agent. The server-side approach can use server-side scripts or it can exploit the header of the HTTP requests.

Opening a web page in the browser consists in the browser sending an HTTP request to the server where the application main document resides. Inside the header of the HTTP request, the browser communicates some information about its identity (i.e., in the User-Agent string) and its capabilities (i.e., the supported MIME types, the accepted character set, the accepted languages). Table 2.1 presents the main HTTP request header fields and their descriptions, in addition to an emergent header, i.e., the HTTP Client Hints that is still a draft in the HTTP working group [16]. Its main objective is to deliver optimized content for each device.

[16]HTTP Client Hints, `http://httpwg.org/http-extensions/client-hints.html`

HTTP header field	Description
User-Agent	Name of browser, user agent or platform from which the request originates
Accept	Comma separated values (CSV) which are MIME types that are supported by user agent
Accept-Charset	Specifies the character set that supports the user agent
Accept-Language	Contains information about the supported languages in the user agent
X-wap-profile and Profile	Provides information about the UAProf (User Agent Profile) XML file that is unique to each device
HTTP Client Hints	Indicates a list of device and agent specific preferences

Table 2.1: Description of the HTTP header fields reflecting the browser profile

Additionally, the header may contain inside the X-wap-profile a link to an XML file that is unique to each device and that describes the device on which the browser is running. Further details about the device descriptions are found in Section 2.3.1.

Additionally, the server can be augmented with Device Description Repositories (DDR) APIs and databases that take the responsibility of getting accurate information about the browser and the device. An example of a DDR is Wireless Universal Resource FiLe (WURFL) [17], DeviceAtlas, DetectRight, etc. As a result, they return more relevant information about the device features comparing to the classic detection using only the HTTP header [14]. DDRs are efficient and accurate, but they require the database to be up-to-date.

In addition to the standardization efforts described in Section 2.3.1, many client-side JavaScript libraries have been released that enable developers to determine properties of the browser with a simple JavaScript API. For instance, the 'Modernizr' API [18] that consists of a set of tests run on the application load and that provides fallbacks. The problem here is that these libraries cannot determine some aspects of the underlying device, e.g., its operating system.

Some browsers implement a set of device detection methods that can be accessed using DOM objects, e.g., screen.width, screen.pixelDepth, navigator.userAgent, navigator.cookieEnabled. In addition, media-queries [1] are another client-side strategy for adapting the application design based on the device or display dimension, orienta-

[17] WURFL, `http://wurfl.sourceforge.net/`
[18] Mdernizr, `https://modernizr.com/`

tion, resolution, etc. Using these strategies, decisions related to the design adaptation can be taken immediately without referring to the server (as it is the case for Responsive Web Design).

We are mostly interested in the client-side adaptation since our solution is a pure client-side solution. In the rest of this section, we focus on the layout adaptation to browser window size.

2.5.1.3 How to adapt the layout?

In this section we focus on the methodologies and on the multiple techniques that exist to adapt the layout to the browser window. There are two main layout techniques: static and fluid layouts that are the basis for two other techniques: adaptive and responsive layouts.

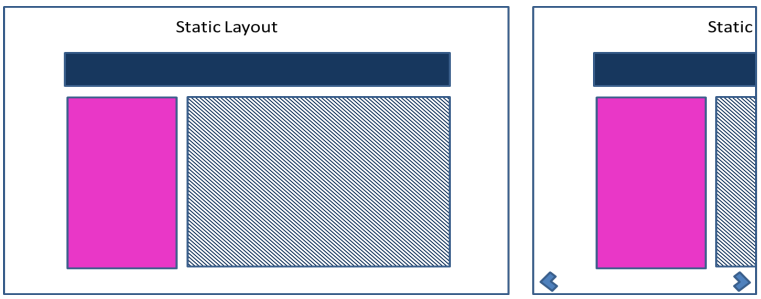


Figure 2-4: Static Layout on large and small window sizes

The static layout (also called fixed layout) is designed with preset sizes for its elements, for instance using the pixel units. As a consequence, the layout does not change based on the browser dimension. Figure 2-4 shows a static layout displayed on a large and a small window. The drawback here is that horizontal scrolling is required to see the complete content on the browser window.

The fluid layout (also called liquid layout) uses relative units (i.e., percentage, em) for content dimensions and positions instead of pixel units. Dimensions/positions with percentage values are calculated on-the-fly, relative to a reference context (e.g., a parent element). The higher reference context is the browser window. Thus, a change in the browser window will trigger changes to related elements. The fluid layout fills the entire width of the browser window while adapting either to the varying size of the screen from one device to another or to the resized browser window on the same device.

Figure 2-5 compares the fluid layout presentation on a large and small window.

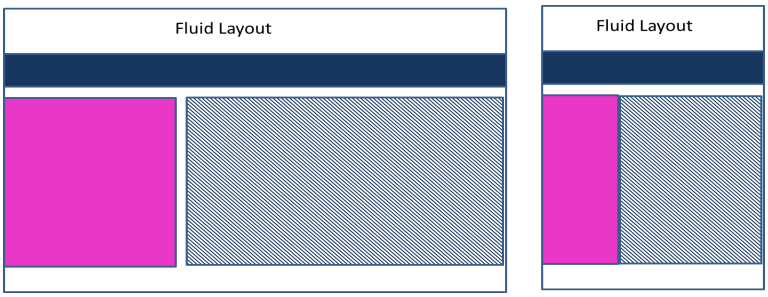


Figure 2-5: Fluid Layout on large and small window sizes

The layout is not optimal when moving from a wide window to a narrow window especially if it results in very small content that makes the text illegible and the interactions inaccessible.

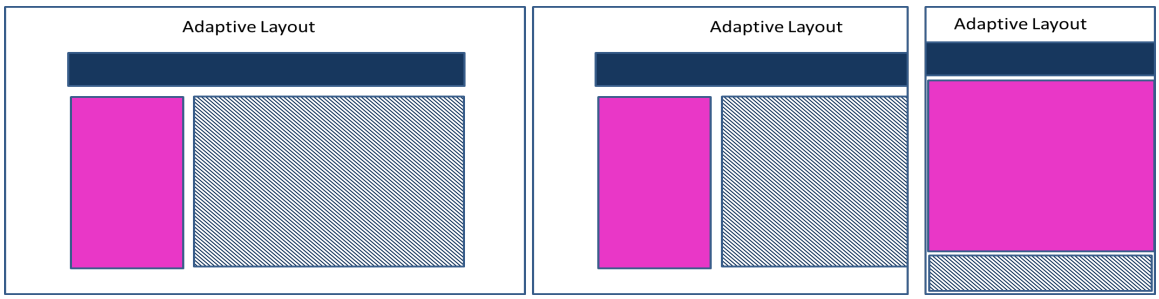


Figure 2-6: Adaptive Layout on large, medium and small window sizes

To avoid the fluid layout issue with the very small content, one can define multiple static layouts one for each relevant window size called breakpoint. The adaptive layout adopts this approach and switches between a set of static layouts at these breakpoints. During runtime, the switching happens promptly due to using CSS media-queries. Figure 2-6 illustrates an example of adaptive layout on a large, medium and small windows. We consider here only one breakpoint. The switching happens when moving from a “large window" to a “small" window, or vice versa. The problem here is specifically on the “medium" window where the layout behaves as a fixed layout and the horizontal scrolling is required again. The layout adaptation is discontinuous.

The responsive layout overcomes the drawback of the adaptive layout by adopting the fluid layout. As the browser window size changes, the responsive layout will flex just like the fluid layout. However, once the window size reaches and detects one of the set breakpoints using CSS3 media-queries [1], it switches the layout. Figure 2-7 is an example of a responsive layout with one breakpoint on three different window sizes. The layout only switches when the window size passes from “medium" and “large" to “small" or vice versa. But, the layout continues adapting itself between the

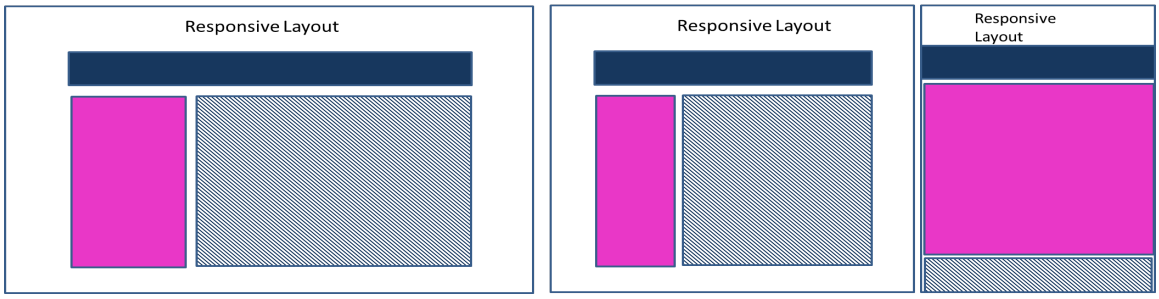


Figure 2-7: Responsive Layout on large, medium and small window sizes

medium windows as Figure 2-7 shows.

In summary, the responsive layout, also called Responsive Web Design (RWD) changes its layout at any point while the adaptive design adapts to the browser window width at specific points.

For either the fluid, adaptive or responsive layouts, the application URL is always the same. The only difference is how the content dimensions are re-evaluated and when the re-evaluation takes place.

2.5.2 Adaptive Design Strategies

Among the client-side approaches for layout adaptation, we cite the adaptive layout [40] and the Responsive Web Design (RWD) [17] that adapt the layout to the browser window size, orientation, etc. RWD makes use of three technical items, i.e., fluid grids instead of static layouts, flexible images and CSS3 media queries. The goal of the RWD is to build all-in-one web pages that detect user screen and orientation and that automatically and dynamically re-organize the content for an optimal experience without adapting the content itself and without requesting the server for additional layouts. Multiple transformations on the user interface elements can take place, e.g., re-arrangement, re-sizing, replacing, changing visibility, moving elements across pages, etc.

One strategy for pure content adaptation is using multiple versions that are saved on the server. Based on the device and browser features detected on the server, the appropriate version is sent. In this case, the application URL is not the same on all devices.

For hybrid approaches, i.e., that include both content and layout adaptations, there is an approach called “RWD proxy sites" that consists in designing multiple

RWD versions for the same site. The only difference with the classic multi-version site, is that the layouts are responsive to the viewport dimensions.

Finally, REsponsive + Server Side Components (RESS) [19] is another hybrid approach that, in contrast to RWD proxy sites, serves the content from one URL. It was proposed by Luke Wroblewski in 2011. The concept uses RWD but supplements it with server-side detection feature to serve modified content when required. The cornerstone of RESS is that the application components are associated dynamically on the server side. For each component, multiple versions exist on the server. Depending on the device detection and the server classification results, the corresponding component version is selected.

The importance of RESS, compared to RWD, is that it serves smaller images on smaller screen or when bandwidth is limited. In contrast, RWD downloads the image and then scale it in the browser. Using RESS, the browser features can be considered in the content delivery (e.g., serving video elements only if the browser supports HTML5, avoid serving Flash games on iOS, fallsback to png when SVG is not supported, etc.).

Strategies	Content Adaptation	Layout Adaptation	Server -Side Detection	Client -Side Detection	Fixed Layout	Fluid Layout	One URL for all devices
Adaptive Layout	-	+	-	+	+	-	+
RWD	-	+	-	+	-	+	+
Multi-version Sites	+	-	+	-	+	-	-
RWD multi-version Sites	+	+	+	+	+	+	-
RESS	+	+	+	+	+	+	+

Table 2.2: Adaptive Design Approaches and Techniques

Table 2.2 summarizes this study and maps the above features to the common AD strategies.

[19] RESS, `http://www.lukew.com/ff/entry.asp?1392`

2.5.3 Frameworks for Responsive Web Design

In this thesis, we focus specifically on the RWD due to its flexible model and since as previously denoted our requirement is to just modify the layout and not the content.

There are several front-end frameworks that support the development of RWD applications. For instance, Bootstrap [53] and Foundation [62] are used to develop responsive mobile-first web applications. They include a grid system to scale the layout as the device or viewport size changes. While these frameworks are powerful, they only address the design of responsive layout from scratch. In this thesis, we develop our re-factoring system to approach the automatic and dynamic re-design of non-RWD application to make them responsive to device screen width.

One identified limitation for the above frameworks is that they do not adapt the application layout based on the dynamicity of the application content nor based on the available empty spaces on the screen. To this end, a variant of media queries, i.e., the element-queries polyfills [20], was developed by Marc J. Schmidt as a proof-of-concept. It triggers changes to an application layout based on the dimension of some DOM elements instead of the screen dimensions. This new concept, though not yet standardized, pushes the layout design one step forward towards the dynamic layout adaptation to the application content. Element-queries can be used as part of our layout refactoring solution to trigger a layout adaption to the amount of visible content on each side of the multi-screen application. More details are found in the Chapter 5.

2.6 State persistence for multi-screen

In the multi-screen context, sequential and complementary usages for multi-screen applications require state persistence across devices. In the first case, moving an application from one device to another is called migration [5]. In addition to changing the device, the migration requires preserving the application view and data. In the second case, the components of a complementary application share a shared context that is distributed between them. This context is continuously changing especially for dynamic applications that evolve with time, and for interactive applications with which a user interacts. The shared context has to be up-to-date to ensure the correct functioning of the application.

[20] Element-queries, `https://github.com/marcj/css-element-queries`

State persistence was addressed in the literature for collaborative multi-user applications where multiple users are collaborating on the same application, e.g., [51], [3], etc.

We have identified two global approaches for state persistence. A first approach requires supporting state persistence by design, using toolkits for distributed user interfaces [36] or using existing APIs, e.g., ShareJS [19] that is capable of synchronizing simple string objects and JSON documents. The problem in this approach is that it requires scattered and verbose source code changes.

Using these tools or libraries for automatically refactoring the code of an existing application is a tedious task, especially because of the complexity of the JavaScript language and the tight relations between the documents of a web application. More details are found in Section 3.4.1. In addition they impose constraints on the authoring technique to developing multi-screen applications.

A second approach consists in transparently adapting the logic to support state monitoring. Using the transparent adaptation approach [51] does not require a change in the application main source code. Instead, it requires extending the code with an adaptation layer that is responsible of listening to changes, recording them and redirecting them. In our work, we use the transparent adaptation approach.

Below, we review a set of existing transparent adaptation mechanisms from the literature.

Imagen [33] proposed an automated generic and transparent approach for persisting and migrating an application state of JavaScript applications. The application state here includes not only the state of the DOM tree but also the state of the JavaScript code. They face the complexities of the JavaScript language, e.g., closures and event-handlers and the complex HTML5 media objects that can be captured and serialized. Quan et al. [44] proposed to collect user parameters into an object called user interface continuation. Programs can create UI continuations by specifying what information has to be collected from the user and supplying a callback to be notified with the collected information. Ghiani et al. [21] focus only on managing the state of forms on web applications.

Mutation-summary [56] is a JavaScript library that can be used to ensure the transparent adaptation and to synchronize elements from a DOM tree. It only requires to be configured to watch specific type of changes that may happen on a DOM tree. Mutation-summary does not allow the synchronization of the JavaScript data and variables. In contrast, ColADA [27] is a front-end framework that extends the

Knockout[21] library and that allows the synchronization of the application model and view. The main objective here is to make easier the development of collaborative MVC applications by limiting the developer responsibilities to simply annotating the application source code. During run-time, these annotations are processed and expanded to blocks of code that ensure the document synchronization.

Panelrama [57] already mentioned in Section 2.2, achieves the "block“ [57] synchronization through a client-server architecture. The developer should first select the state variables that need to be synchronized among the distributed blocks, and then should change the logic of the application. Once detected, the block changes are propagated to a central page to be after redirected to the device concerned in this change to update its view.

HydraScope [26] is a framework that distributes and orchestrates the distributed web-based user interfaces that run in parallel through a central synchronization server. The orchestration consists in synchronizing mainly the application view without having to access the application logic, based on DOM sensing, inspections and event injection methods. HydraScope can be extended with libraries to support the data view synchronization following some rules hard-coded by the developer or provided as options for end-user.

In this thesis, we focus mainly on the view synchronization and we use decentralized mechanism for the synchronization.

2.7 Conclusion

This chapter focused on presenting the technological and scientific advances in the domain of multi-screen application development and the corresponding platform support. Web services is the adopted technology that makes the cross-platform and cross-language communication a reality. The protocol stacks that orchestrate the web service behavior were also presented.

The COLTRAM research project exploits web service technologies and protocol stacks to build a multi-screen platform that harmonizes the interfacing between UPnP and Bonjour services. The COLTRAM platform does not include authoring COLTRAM-enabled multi-screen applications, while it is required.

In the literature, multiple multi-screen application frameworks were proposed us-

[21] KnockOut, `https://github.com/knockout/knockout`

ing either a client-server topology or a centralized client topology. Most of these frameworks reuse existing applications to create multi-screen applications following their topology, but they all require the developer to decide on the content distribution either by manual annotation or using web components. In contrast, we propose an automatic content distribution for video-centric applications based on device features.

Concerning the content distribution, segmentation approaches are developed in the literature with multiple applications in mind. We found that the hybrid segmentation approach is useful for our content distribution requirements. After studying the existing hybrid techniques for segmentation, we found that the abstraction model provided by BOM [46] facilitates the segmentation task, and the application re-design. By adopting its abstraction model and by making the multi-screen environment controls the segmentation results, we propose a new approach for application segmentation for content distribution.

We investigated as well the theory and the basis of the web application design, specifically the adaptive web design that is capable of adapting the content and the layout to the environment. We found that client-side approaches for layout design are the solution for our requirements to enhance the multi-screen applications delivered by our system. This is due to the fact we have no control over the domains from which the applications are served.

Finally, we checked in the literature the existing approaches for providing state persistence. While rewriting and intercepting the JavaScript logic looks very limiting and complex to be done automatically, the transparent adaptation technique seemed more feasible. In addition, we reviewed some of the logic transforming works that were used to ensure the synchronization between distributed content, with a focus on view synchronization.

Chapter 3

Overview of the Refactoring System

This chapter is a global introduction to the refactoring system. The aim is to present the whole picture and to give an overview the system basis before detailing them in the following chapters. The chapter covers the system architecture, the input and output characterization, the extrinsic factors influencing the system, and finally the multi-screen application model that it assumes.

As part of the input characterization, we also characterize the application datasets that are used in the following chapters to validate the system components.

3.1 System Global Description and Architecture

Our work operates on multiple aspects to provide an end-to-end refactoring system that delivers functional multi-screen applications.

The first aspect is the exploitation of the multi-screen environment and the characterization of web applications. More details are found in Section 3.3.

The second aspect is the distribution of the application user interfaces and their adaptation to the multi-screen environment.

A user interface is considered as the association 1) of the information structure, 2) of the interaction design that defines how people can manipulate and contribute to that information and finally 3) of the visual design [1]. For web applications, the user interface structure and the visual design are mainly created using HTML and CSS documents. The interaction design designates the logic behind the elements of

[1] See `http://www.freshtilledsoil.com/what-is-user-interface-design/`

the user interface and it is usually implemented in JavaScript.

In the context of multi-screen applications, a single User Interface (UI) is substituted with Distributed User Interfaces (DUI) that can be distributed geographically, among multiple devices, among graphical displays and means of interaction [15].

The third aspect is the state distribution and the preservation of the application functionality during runtime and the runtime support for the application dynamicity.

Section 3.4 introduces and arguments the adopted model for the user interface distribution.

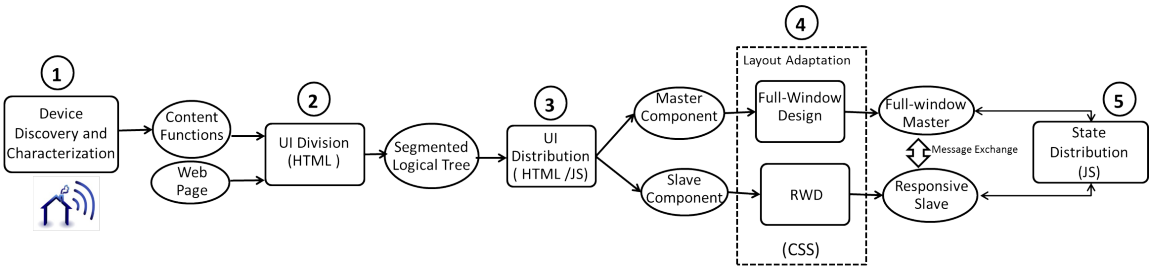


Figure 3-1: The architecture of the refactoring system

Each of the above three aspects are implemented in one or more components in our system architecture presented in Figure 8-1.

The first aspect is represented with the Device Discovery and Characterization component (number 1). The second aspect covers components number 2, 3 and 4. UI Division and UI Distribution deal with the processing of the HTML and the JavaScript documents to deliver the functional multi-screen application. Layout Adaptation processes the application CSS document to adapt the multi-screen application layouts to the device types. The third aspect is represented with the State Distribution component (number 5) that is mainly concerned in JavaScript.

3.1.1 System Features

The refactoring system is dedicated mainly for end-users and aims at alleviating the user tasks during a multi-screen experience. The system is automatic and it is designed to not require a configuration from the end-user. The end-user only needs to provide the application she wants to split, to connect multiple devices to the network and finally to command the system after selecting the devices of her interest.

Our current implementation targets specifically multimedia web applications, and

more precisely video-centric applications that have at least one video element. Our choice for supporting the video-centric application is motivated by the increase of the online video consumption across the globe. In 2015, a survey related to the online video consumption [2] forecasts that the average amount of time people spend consuming online video each on a daily basis will grow by 23.3 % by 2015 and 19.8% by 2016. In an article published in June 2016, Cisco forecasts that the IP video traffic will be 82% of the consumer traffic by 2020, up from 70% in 2015 [3]. The most notable in these forecasts is that the mobile video consumption is growing at roughly 5 times the rate of non-mobile devices (i.e., smartTV and desktops), and that mobile consumption will form 52.7% of the total consumption in 2016.

Based on these facts and following our requirement, a dataset was selected among existing single-screen applications and it is described in Section 3.2. The dataset is used throughout the thesis to validate most of the system components.

Enlarging the system scope to cover different types of applications is possible since the system is extensible. The key point here is to characterize these applications and to identify the common features that can be used by the system to automatically separate and distribute the application contents. We note that the system is designed in a modular manner that facilitates its reuse and its extension to support different application types.

From a practical point of view, the complementary multi-screen usage is mostly relevant between two devices. Based on this, the system implementation focuses on a distribution between two devices, but the system is also valid for more than two devices. Throughout the following chapters, we will validate quantitatively our system on two devices. In addition, a first investigation for our system with three devices is also conducted in Chapter 6.

Figure 3-2 illustrates a layered and a global presentation of the system runtime environment. The system runs on top of the browser, specifically the JavaScript engine, and the COLTRAM platform. It defines multiple APIs that operate on the three documents (i.e., HTML, JS and CSS) of the single-screen application.

Situating the system between the JavaScript engine and the application is required since the system conducts a dynamic analysis on the application documents and thus it requires the access to the DOM tree, and it needs also to filter browser

[2]Online video consumption growth forecasts, `http://www.zenithoptimedia.com/mobile-drive-19-8-increase-online-video-consumption-2016/`

[3]Cisco survey, `http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/complete-white-paper-c11-481360.html`

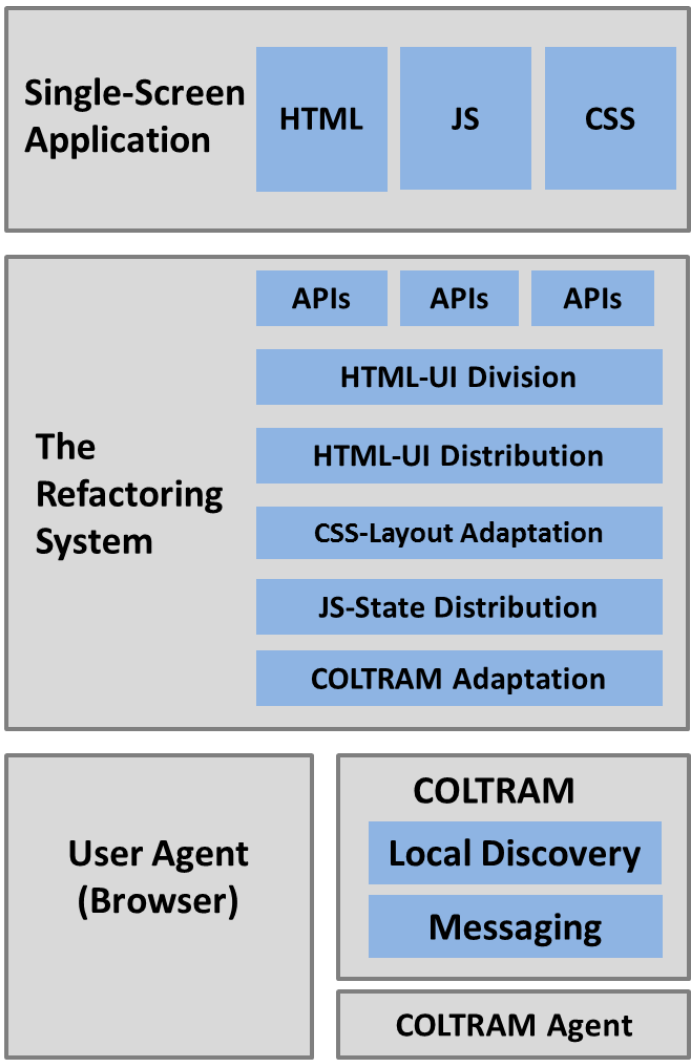


Figure 3-2: The refactoring system between the user agent and the application

events. We note here that the execution context of the refactoring system and the execution context of the MSA application are completely separated. The interest of this separation is to avoid the incompatibility issues in the presence of multiple development environments of web applications, i.e., using external libraries such as jQuery [4], prototype [5], RequireJS [6].

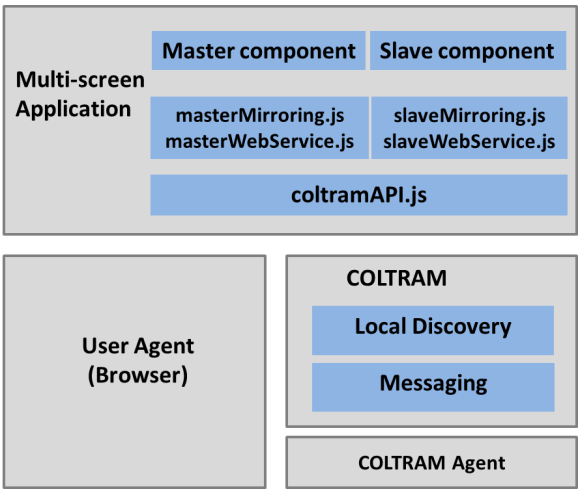


Figure 3-3: The delivered multi-screen application

Once the end-user commands the system, the system output is a multi-screen application that is not tied to the refactoring system. It runs on top of the browser and the COLTRAM platform as shown in Figure 3-3. This feature makes the multi-screen application portable (or migratable) across connected devices in the COLTRAM platform.

Note that the system is not tied to the COLTRAM platform, and other discovery and communication solutions (e.g., postMessage [7], webRTC [8]) are also applicable. In a preliminary work reported in [47], we used postMessage as a means of communication between the multi-screen application components running on the same device. This last aspect is a limitation for postMessage since it does not support the cross-device communication.

[4] jQuery, `https://jquery.com/`
[5] Prototype, `http://prototypejs.org/`
[6] RequireJS, `http://requirejs.org/`
[7] postMessage API, `https://developer.mozilla.org/en-US/docs/Web/API/Window/postMessage`
[8] WebRTC, `https://webrtc.org/`

3.2 System Input and Dataset Characterization

We believe that characterizing the system inputs and the selected dataset at first helps in sensitizing the reader to our system scope and to the application features that are relevant for the processing of the five system components.

We build a dataset of web applications containing at least one video. Most of these applications are designed for the desktop and are not responsive. Our dataset can be split into three categories as follows:

1. Full-featured video-centric applications: here, we select the top 11 most popular video websites based on the ranking of eBizMBA Rank [9] by March 2016.
2. Basic video player applications: we select 7 basic applications provided by different video libraries, i.e., mediaElement [10], videojs [11], jplayer [12].
3. the semantic video application: a highly dynamic video application developed by Mozilla.

Table 3.1 contains the exhaustive list of our 19 applications. We characterize these applications by identifying mostly the aspects related to the graphical user interface. For instance, the number of links with which a user can interact, the presence of video elements (HTML5, Flash, etc.), the number of images, the word count that gives an idea about the website type and the page height when rendered on a 1920*1080px desktop. Finally, we conducted some manual tests to check whether the selected sites are responsive. One limitation here is that we do not provide an automatic mechanism to detect whether an application is responsive or not.

It is important to note that values are reported in Table 3.1 reflect the state of the application just after the page has loaded, i.e., before running the video and without considering any advertisements on the page. These numbers might change during runtime after running the application. These changes characterize the dynamic behavior of an application. This is in particular very relevant for the semantic video application.

For each application, Table 3.1 indicates different characteristics of these applications. It indicates the number of DOM elements each page contains ranging from 113

[9] See `http://www.ebizmba.com/articles/video-websites`
[10] See `http://mediaelementjs.com/`
[11] See `http://www.videojs.com/`
[12] See `http://jplayer.org/`

Table 3.1: Dataset characterization on a 1920*1080px viewport

Apps/Count	DOM nodes	Video elem.	Inter-active elem.	Link elem.	Image elem.	Word	Doc. Height	RWD
Viewster [a]	684	1	480	89	31	333	2285	Yes
Vimeopro[b]	113	1	73	24	5	83	1465	No
Vimeo[c]	1083	1	572	166	34	864	3169	Yes
Youtube[d]	2800	1	1633	254	56	878	4831	Yes
Dailymotion[e]	1172	1	572	55	17	441	2506	Yes
Yahooscreen[f]	1284	1	596	159	2	170	5881	Yes
Twitch[g]	809	1	403	209	60	440	3981	No
Liveleak [h]	631	2	308	111	21	392	1971	No
Ustream [i]	1051	1	436	91	4	403	2015	No
Break [j]	3411	1	1401	538	106	1495	7750	No
Metacafe [k]	449	1	225	46	6	181	1663	No
VideoJS [l]	175	1	124	1	0	37	1099	No
Jplayer demo1 [m]	31	1	8	1	0	15	1099	No
Jplayer demo2 [n]	36	1	8	2	0	16	1099	No
JWPlayer demo1 [o]	19	1	2	2	0	2	1099	No
JWPlayer demo2 [p]	324	1	284	5	18	13	1099	No
JWPlayer demo3 [q]	1091	1	3	2	0	10077	1099	No
MediaElements [r]	104	2	18	3	1	138	1306	No
Video Player Pages (average)	254	1.14	64	2	3	1471	1129	No
Semantic Video[s]	528	1	312	41	39	119	1090	No
Average	831	1,1	376	90	20	878	2371	-

[a] Viewster, `http://bit.ly/1RiOBqA`, [b] Vimeopro, `http://bit.ly/29uf3xQ`,
[c] Vimeo, `http://bit.ly/1QsT7lm`, [d] Youtube, `http://bit.ly/1OVFaLO`,
[e] Dailymotion, `http://bit.ly/25nuLka`, [f] Yahooscreen, `http://bit.ly/1T7nQqk`,
[g] Twitch, `http://bit.ly/18uyRaC`, [h] Liveleak, `http://bit.ly/1pJG8CV`,
[i] Ustream, `http://bit.ly/1XPrpkl`, [j] Break, `http://bit.ly/1RiOBqA`,
[k] Metacafe, `http://bit.ly/1RncU5m`, [l] VideoJS, `http://bit.ly/29y2ceE`,
[m] Jplayer demo1, `http://bit.ly/29J6r90`, [n] Jplayer demo2, `http://bit.ly/29nULaY`,
[o] JWPlayer demo1, `http://bit.ly/29p3eXN`, [p] JWPlayer demo2, `http://bit.ly/29nUR2o`,
[q] JWPlayer demo3, `http://bit.ly/29uezrJ`, [r] MediaElements, `http://bit.ly/29Fnwhg`,
[s] Semantic video, `http://bit.ly/29CwVpN`

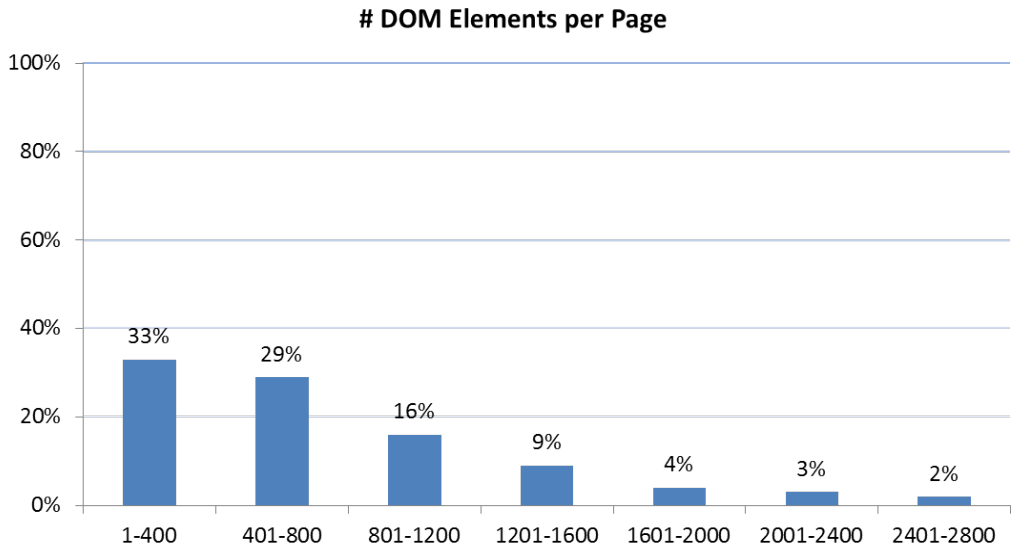


Figure 3-4: Average number of DOM nodes per page

to 2800, on average 831 nodes.

This shows that our dataset is representative of typical websites as illustrated in Figure 3-4[13]. We have 8 applications with less than 400 nodes, 4 applications between 401 and 800 nodes, 5 applications between 801 and 1200 and 2 applications with more than 3000 nodes. Among these nodes, on average only one element is a video except for the Liveleak page and for one mediaElement application where there are 2 videos. On average 46% of these nodes are interactive following our definition given in Section 4.3, among which 24% are links. These percentages confirm the presence of interactivity in our dataset. Youtube, Dailymotion, Vimeo and Viewster applications are responsive, thus they will not be used to test our responsive re-design approach. The average word count is 878 words that is distributed between the video title, the link text, user comments, short description about the video, or video-linked information (or transcript) as it is the case for the semanticVideo application and JWPlayer demo3.

For a window with a width of 1920 px, the height of these pages varies between 1090 and 7750 pixels, on average 2371 pixels. This helps us to identify whether the height of a page can influence the performance of our system.

[13]HTTPArchive statistics, `http://httparchive.org/interesting.php`

3.3 The content-device duality

As mentioned earlier in this chapter, an important feature of our system is that it is automatic and does not require any external assistance from a user. It uses the environment diversity in the multi-screen ecosystem to assist the distribution of the application content. In this section, we study the relations that exist between devices and applications with the intention of providing on each device a 'well-fit' portion of the application, based on device features and capabilities.

In the context of multi-screen applications, a Google study [23] conducted a user-study in 2012 to understand the user behavior in a cross-platform. It identified four main device combinations for complementary interactions between two devices, with their corresponding usability percentages:

1. TV and Tablet, 40%
2. PC and Smartphone, 36%
3. TV and Smartphone, 35%
4. TV and PC, 32%

In most of the above combinations, the main device is a TV or a PC that both have a large screen allowing a user to be placed relatively far from it. The second-screen device, i.e., tablet or smart phone, has a smaller screen size, it is a portable device and it is simple to use and to interact with its touch screen.

Functions / Devices	Video Rendering	Text Visualization	User Interaction
TV	++	+	–
PC	++	++	+
Smart Phone	+	+	++
Tablet	+	++	++

Table 3.2: Characterizing devices with features related to the context of usage

Based on these observations, we propose in Table 8.1 a characterization for the four device types, i.e., TV, PC, Smart phone and tablet, with functions, i.e., video rendering, text visualization and user interaction. A function refers to the type of media that a device can render, i.e., audio, video, or to the type of interaction between

an end-user and a device, i.e., to display content or to listen to user data and user requests. The list of functions is not exhaustive and can be extended and refined to consider more device features, such as checking the support for media formats.

We associate a score for each combination of a device and a function. This score varies between '-' at its minimum utility, '+' at its medium utility and '++' at its maximum utility. It reflects how much a device is suitable for a certain function. For instance, Table 8.1 states that TV and PCs are more suitable for video rendering than smart phones and tablets in the context of home networks and complementary usage.

TVs, with a negative interaction score, should be avoided for non-trivial user interactions. Cesar et al.[10] and Bernhaupt et al. [6] report on the complexity of using a remote control to interact with a TV especially because its buttons are limited to arrows, channel numbers and a validation button. For example, navigating an EPG requires extra effort comparing to using a mouse on a PC, or touch screen on smart phones. On the other side, smart phones and tablets are the most suitable for user interaction.

For text visualization, PC is considered better than TV though both have a large screen, but in contrast a TV is usually situated far from the user while the PC is at a shorter distance, thus allowing a better visualization for large texts. If we compare the text visualization on a smart phone and on a tablet, they are both good at displaying large text content and they are both destined for private use, but a tablet with its wider screen requires less effort to read the text, e.g., the amount of text displayed at once on the tablet is larger than the one on a smart phone with a smaller screen size, in addition the text can be better readable on a tablet.

The functions characterizing devices in Table 8.1 also characterize the content of the applications. For example, a media part of an application can be identified by the presence of a video or audio element or an embedded Flash object. Similarly, an interactive part of an application can be identified by the presence of input elements, or event listeners that listen to user interactions. In consequence, these functions can be used to associate the application content to devices.

A multimedia application is an application that uses a collection of various media sources, e.g., text graphics, audio, animation and/or video. Comparing these media sources to the functions identified above, a multimedia application can be viewed as formed with multiple blocks where each block is characterized with a function. A block here is defined as a set of application elements that are visible on the screen

and that form an entity that is independent from the rest of the blocks.

In our work, we do not duplicate visible blocks between the primary device and the secondary device(s) on the one hand. In consequence the number of devices among which we distribute the application is smaller or equal to the number of the identified blocks. But visible blocks could be duplicated among multiple secondary devices resulting in cloned components. In this case, there would be no constraint on the number of secondary devices. We assume that audio and video content are associated to a large device, i.e., TV or a PC as a first choice. We call these content 'multimedia' content. Interactive content are associated to smart phones or tablets as a first choice together with large text content. We call them the 'interactive' content.

As previously stated, our system is responsible for guiding automatically the splitting of the applications based on the environment. A user can at any time launch her single-screen application on a device, connect another device and then query the system to distribute the application on the two devices. The selection of two devices among multiple devices on the network is out-of-scope in this thesis. This can be done automatically, for instance, using the cost matrix approach proposed by Yang et al. [57]. In our work, we assume there are always no more than two devices at a time.

The device characteristics on which we focus are: (1) the number of screens, (2) the screen size, i.e., large or small display, (3) the means of interaction, i.e., touch input, keyboard, mouse or non-interactive, (4) the type, i.e., TV, PC, tablet or smart phones. The COLTRAM platform includes a web service that collects the features listed above. Then, for each device we identify its dominant feature that we consider as the function of the device based on Table 8.1 explained in the previous section. For instance, in the case where a user selects two identical large devices, i.e., 2 PCs, we first compare their screen sizes. We consider that the PC with a bigger screen is more suitable for displaying 'multimedia' content than the other PC.

We recall that the two functions we consider in this work are 'interactive' and 'multimedia', but this list is not exhaustive and could be extended to include for example an 'information' function for large textual content.

The identified content functions will guide the application analysis done inside the components number 2 and 3 in Figure 8-1. More details are provided in Chapter 4.

3.4 Multi-Screen Application model

As mentioned earlier, distributing an application user interface is equivalent to distributing its associated documents, i.e., notably HTML, JavaScript and CSS.

Multiple questions arise when trying to find an application model that is based on splitting the HTML, CSS and the JavaScript documents. The questions are related to 1) the features of each language, 2) the difficulties of their dynamic analysis during runtime, 3) their dependence from each other (tight links) and 4) the distribution cost on the system performance.

Sections 3.4.1, 3.4.2 and 3.4.3 are dedicated to answer these questions from a technical point of view. The resulting multi-screen application model is detailed in Section 3.4.4.

3.4.1 Splitting the HTML document

The JavaScript representation of HTML documents, CSS properties and the event system is given by the W3C Document Object Model (DOM) [14]. In another word, the DOM tree is a dynamic representation of the HTML document, where parent and sibling relations among HTML nodes are explicitly expressed and where the HTML elements and attributes are accessible and manipulable dynamically using JavaScript. In addition, using the DOM tree more information concerning the application state can be captured compared to the HTML document. For instance, multiple event listeners can be set simultaneously on one HTML element and this can only be seen in the DOM tree.

By analyzing the DOM tree dynamically, it is possible to partition and to distribute the corresponding HTML document. Figure 3-5 illustrates a simple DOM tree (a) where elements have two patterns. To split this tree, elements having the yellow pattern created DOM tree 1 (b) and elements having the blue pattern created DOM tree 2 (c). The body element is at the head of every DOM tree, so it is duplicated.

On one side, this splitting methodology reduces the size (i.e., in terms of number of nodes) and the complexity of the DOM tree compared to the initial DOM tree. In consequence, it reduces the loading time of the distributed application especially on devices with limited connectivity and resources. In this example, the DOM tree

[14] DOM, `http://www.w3.org/DOM`

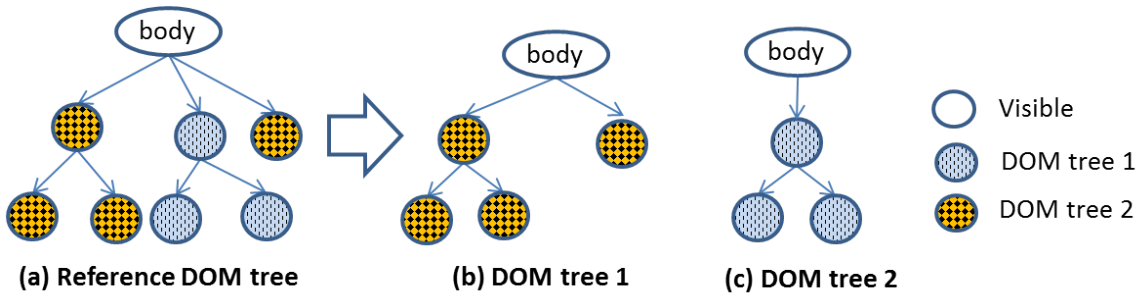


Figure 3-5: Default DOM Splitting illusttration

is small but existing web applications have a more complex DOM trees as shown in Table 3.1, and the reduction of the number of DOM nodes is more remarkable and more relevant than this simple example.

On the other side, this radical splitting is harsh especially because parent and sibling relations are not preserved. The body element had three children in the reference DOM tree (a), but has two on DOM tree 1 and only one on DOM tree 2. In addition, this radical splitting can have consequences on the main logic interpretation and correctness since there are tight relations between the DOM tree and the JavaScript code, as it is going to be detailed in the next Section 3.4.2.

3.4.2 HTML/JavaScript links

Jensen et. al. [29] states that the JavaScript code in general cannot be analyzed separately from the HTML code. This is because the JavaScript logic is usually added to the browser to manipulate the HTML documents dynamically during runtime and to allow end-user to interact with the application. The JavaScript engine on the browser is responsible for these tasks, i.e., dealing with the application user interface, in addition to dealing with the server communication and other tasks.

Using the DOM interfaces defined in the browser API, the JavaScript logic can concretely:

1. access DOM elements directly, e.g., using “document.getElementById", or through exploiting parent relations using “.firstElementChild", “.parentElement", “.nextSibling", etc.
2. add DOM elements, e.g., using “document.appendChild"
3. add and edit the attributes of DOM elements, e.g., using “.setAttribute"

4. delete DOM elements or their HTML attributes, e.g., using “.removeChild", “removeAttribute"

5. add and delete event listeners on DOM elements, using HTML attributes, e.g., the “onclick" attribute, or using “.addEventListener"

Using the DOM interface methods to manipulate the DOM tree usually implies a perceivable change on the HTML document. Thus, these methods preserve a coherence between the state of the HTML document and the DOM tree object. Some of the above changes to the DOM tree, especially those related to HTML attributes and event listeners, can be done using built-in setters, i.e., JavaScript properties. Examples for these properties are: “.id", “.selected", “.value", “.style", etc. Correspondingly, some of them imply an update to the HTML document while some others do not (e.g., .selected, .value, etc).

In addition, the JavaScript program execution is driven by events in the browser, e.g., the page is first loaded, the user interacts using the mouse or the keyboard, timeouts occur, responses to HTTP requests are received, etc. The event handler code reacts by modifying the program state and the HTML page via its DOM tree and by interacting with the browser API. The event handler code can be defined separately in a JavaScript document or in a script element or as a value for an HTML attribute. Here again Jensen et al. [29] statement applies.

Running the JavaScript code separately from the HTML document, for instance in NodeJS [15], will not run or will break because 1) its execution is triggered following events on the HTML elements and 2) it continuously interacts with the DOM tree.

This means that the option of isolating the JavaScript document from the application DOM tree is not a possible solution.

3.4.3 To Split or Not To Split the JavaScript

From the previous section, the JavaScript logic should always be on the side of the DOM tree. But if the distributed application has two DOM trees, is it necessary to assign a logic for each of them?

In this section, we are going to answer this question throughout two scenarios and we are going to study their feasibility and their challenges. Scenario 1 aims at

[15]NodeJS, `https://nodejs.org/en/`

splitting the JavaScript document. Scenario 2 is to keep the main JavaScript logic intact next to each of DOM tree 1 and DOM tree 2 of Figure 3-5.

In the beginning of this thesis, we adopted by default Scenario 1 and we developed an algorithm to split the JavaScript code according to the split DOM tree. This first trial helped us in identifying concretely the challenges and the complexities of this scenario. Then, it pushed us to think about Scenario 2 and to conclude finally about the drawback and the advantages of each of them.

3.4.3.1 Scenario 1: To Split the JavaScript

We recall that Scenario 1 consists of splitting the DOM tree and splitting the JavaScript document as well.

The basis for the JavaScript splitting is to try to separate the code related to elements in the DOM tree 1 from the code related to elements in the DOM tree 2. The following section focuses on the challenges and the cost of splitting the JavaScript code on the performance of the application and the system.

The first obstacle we face is the JavaScript language itself and its dynamic aspects. JavaScript has higher-order functions and closures, exceptions, extensive type coercion rules, and a flexible object model where methods and fields can be added or change types and inheritance relations can be modified during execution [29]. The complexity of the JavaScript language is also expressed in the complex interactions with the browser and with the HTML DOM as stated earlier in Section 3.4.2.

Talking about browsers, the browser environment gives rise to additional challenges. In fact, they do not follow the ECMAScript language specification as it was reported by Maffeis et al. [35] in their trial to understand the JavaScript language. In addition, browsers provide non-standard functionality. Incompatibilities in the browser environments are a major concern when it comes to modeling the interactions between JavaScript and browsers.

More challenges were identified in the literature [35], [29] and static analysis approaches were proposed to deal with them. But, these methods are heavy since they consist in constructing data flow graphs and call graphs on one side. On the other side, they require a preliminary phase, running the complete application to illustrate all its states. Thus, such a method is not suitable for our system objectives concerning refactoring any application at any moment.

For the moment, let us assume that we could split the JavaScript document to

study the consequences on the performance of the application and of the system.

Listing 3.1: Example of a JS document

```
1 var f = function (){
2   var v = document.getElementById('DOM1element').value;
3   document.getElementById('DOM2element').value = v * 10;
4 };
```

A simple function in Listing 3.1 is defined. It reads the value of an element belonging to DOM tree 1 in line 2 and updates accordingly the value of an element belonging to DOM tree 2, in line 3.

Listing 3.2: Split JS document

```
1 //code associated to DOM tree 1
2 var f1 = function (){
3   var v = document.getElementById('DOM1element').value;
4   stub ({'call': 'f2', 'args': v});
5 };
```

Listing 3.3: Split JS document

```
1 //code associated to DOM tree 2
2 var f2 = function( remoteValue ){
3   document.getElementById('DOM2element').value = remoteValue *
        10;
4 };
```

The splitting of the function f is straightforward here as shown in Listings 3.2 and 3.3. But it is much more complicated in the presence of loops, global variables, anonymous functions, etc.

But apart from conceiving and developing a runtime method to split the code instructions, additional tasks are required to preserve the initial execution order and to preserve the data flow between the distributed application. This is illustrated in line 4 of f1 where a stub function is called after line 3 has executed. The stub function is responsible for collecting the necessary arguments, i.e., the remote function to be called (f2) and the required arguments to make f2 executes as expected. In addition, the stub function is responsible for calling f2 with these arguments.

Practically, Listing 3.2 is associated to DOM tree 1 that is rendered on the browser of a first device. And Listing 3.3 is associated to DOM tree 2 that is rendered on the browser of a second device. In this case, the stub manages as well the messaging and

the communication between both sides. The drawback of using stub functions is that they can overwhelm the network with excessive and non-relevant messaging between both devices.

These challenges make the dynamic analysis of the executable JavaScript code a very complex and heavy task especially during the application runtime. For these reasons, we dropped the idea of analyzing the JavaScript code and we decided to abstract from the JavaScript code.

3.4.3.2 Scenario 2: Not To Split the JavaScript

We recall that Scenario 2 consists in splitting radically the DOM tree and keeping an unmodified version of the JavaScript document next to DOM tree 1 and DOM tree 2.

The JavaScript *element lookup* mechanism, i.e., using for example *getElementById*, *getElementsByTagname* and related functions, searches for elements based on their id, tag name, class names, etc., or using the JavaScript properties. These methods are applied to a DOM element object and return null, one or more elements.

A null element is returned 1) if the DOM tree does not have at least one element that satisfies the search criteria or 2) if the search criteria is not valid. In addition, calling a method or getting/setting properties on a null element breaks the application logic. This exactly can happen if trying to access the omitted elements in Figure 3-5(b) or (c).

In some other cases, the logic will not break but instead the application behavior may change.

Listing 3.4: Example of a HTML document

```
1 if ( document.body.children.length == 3)
2      call f();
3 else
4      call g();
```

For instance, the JavaScript in code Listing 3.4 is associated to the reference DOM tree in Figure 3-5(a), and the expected behavior is to call function f(). Applying this code on DOM tree 1/2 of Figure 3-5 , results in executing function g() since the body element has 2 children now on DOM tree 1 and one child on DOM tree 2.

To avoid the above problems, there are two solutions:

- to modify the main logic and adapt it to the distributed DOM tree. But we dropped it in Section 3.4.3.1.
- to preserve both the JavaScript logic and the main logic on one side.

3.4.4 Summary and Our MSA application model

Reverse engineering logic is a heavy and a complex task that is complicated to be done dynamically as previously shown. Our solution operates transparently on the original source code. It does not change it but instead it adds a layer between the browser and the application at runtime.

The challenges in transparently modifying the logic are first that the script logic of a web application cannot be valid if it is separated from the DOM tree, especially if the code is used to read and manipulate the DOM tree during run-time. Thus, having the whole script logic with a subset of the main DOM tree does not ensure that the code will not break.

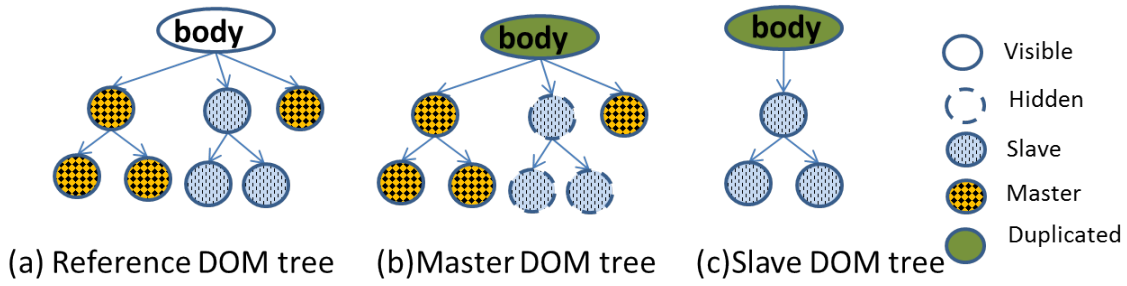


Figure 3-6: The adopted multi-screen application model

In consequence, we decided to keep the main logic intact and to assign it to only one component of the distributed application.

Our multi-screen application model consists of a master and a slave component as shown in Figure 3-6. The master component is a rich application that holds all the application logic. It has a DOM tree that is a slightly modified copy of the main application DOM tree. Elements that belong to the slave component are hidden on the master as illustrated in Figure 3-6 with dotted borders. These hidden elements are considered as a shortcut to the slave component whenever the logic requires the access to the slave DOM tree. Thus, the strategy of keeping the original tree on the master reduces and limits the communications between both components to only exchanging changes and updates.

The slave component has a DOM tree that is a subset of the main application DOM tree. It has no logic, but it is dependent on the master component.

Using this application model, our system has an additional responsibility to provide an interface between the master and the slave components to let them ensure the complementary usage and the state coherence especially for the slave component.

The complementary usage implies a dependency between the components that form the multi-screen application, especially if the application is dynamic and if during the run-time a change on one component triggers a change on the other component(s). Thus, additional cross-device operations are required during run-time and need to be provided by our system.

The main two run-time operations that interest us are the synchronization and the redirection, to which we refer by state distribution. The synchronization consists in ensuring that the states of the application components are coherent at any moment during run-time. The term state in this thesis is limited to the state of the DOM tree. The redirection refers to the mechanism of propagating the state changes towards the concerned component. To integrate these operations at the level of each component, we add an additional layer of logic that detects state changes and sends them when necessary.

To make the slave capable of propagating user requests and data input to the master component where they will be processed, and in order to make it capable of integrating the updates received from the master, we enrich the slave with a logic, independent from the original application logic. On the master component, updates concerning the slave will be monitored, captured and serialized to the slave. Further details are provided in Chapter 6

3.5 Conclusion

This chapter aimed at previewing the complete system and at presenting its main features and requirements based on technical and theoretical studies.

The architecture of the described system consists in five principal blocks: the environment exploitation, the UI division, the UI distribution, the layout adaptation and finally the state distribution.

As an input, the system takes a video-centric application that contains at least one video element and delivers a multi-screen service application at the output.

The refactoring process is influenced by the device features. This was the result of the identified content-device duality that exists between a multimedia application and device features.

The model of the output application was decided after investigating multiple solutions and after conducting technical studies on the HTML and JavaScript codes. The adopted model consists in a master-slave model. The master is a rich application having the main application logic on its side and having a slightly modified version of the main DOM tree. The master relies on the slave to receive user data and queries.

The slave has a DOM tree that is an extract from the main DOM tree and it relies on the master to update the state of its DOM tree.

Chapter 4

Creating Multi-screen Applications

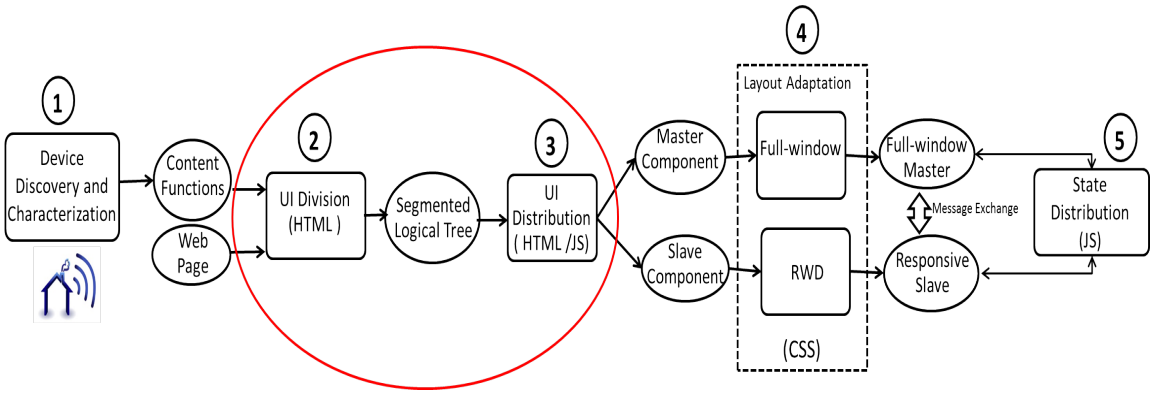


Figure 4-1: UI Division and UI Distribution phases in our refactoring system

This chapter describes the part of our refactoring system responsible for the application distribution including: the User Interface (UI) division and the UI distribution components as illustrated in step 2 and step 3 in Figure 4-1. While UI division only selects/identifies the elements to distribute, the UI distribution creates the user interface of the master and slave components. The UI distribution approach described in Section 4.4 is independent of the UI division methods. As a result of the application distribution, a multi-screen application is created following the application model described in Chapter 3.

The chapter describes first our three different tentatives to divide a user interface. First, we have developed a visual tool in Section 4.1, that requires the user to divide visually the user interface by selecting a region of interest. Second, we have developed a DOM-based analysis method described in Section 4.2, to automatically divide the user interface structure, from the environment features.

Finally, we combined the visual analysis of the first method and the structural analysis of the second method to obtain a hybrid analysis method using a segmentation technique described in Section 4.3. This combination circumvents the limitations of each of the two methods and adapts them to our system requirements while resulting in the expected user interface division. The conformance of the resulting user interface to the expected user interface is studied and evaluated in Section 4.6.

4.1 Screen-Region Selection Method

4.1.1 Principles

The screen-region selection method is based solely on the application visual rendering. It requires that an end-user selects a rectangular region of the browser window during runtime. The method consists in replicating the selected region on the slave component. The remaining regions belong to the master component. It works by analyzing the geometry of the DOM elements to identify those that fall within the selected region.

The challenges here are mostly related to the proper mapping of the selected region to the corresponding DOM elements, based on their geometrical properties. If the mapping is not correctly done, unwanted elements are sent to the slave component while they should remain on the master component or vice versa. There are multiple causes for these problems, as follows:

- the region selection might lack of precision and in consequence unwanted elements are considered as selected DOM elements (Problem 1) or vice versa.
- there might be an incoherence between the calculated geometry of some DOM elements and the effective space they occupy on the screen. This can happen for elements whose geometry does not encompass those of its descendants, or for the block-level elements that take the available full-width independently of their content size (for example, the H1-H6, P, DIV elements) (Problem 2)

The screen-region selection method is illustrated in Figure 4-2 in the form of a program flow. The algorithm runs once an end-user selects a region on the screen using a selection tool, e.g., RectMarquee [1]. The geometry of the selected region is provided as an input to the algorithm in addition to the DOM tree.

[1] rectMarquee Tool, `https://github.com/mfaber/quexf/tree/master/js`

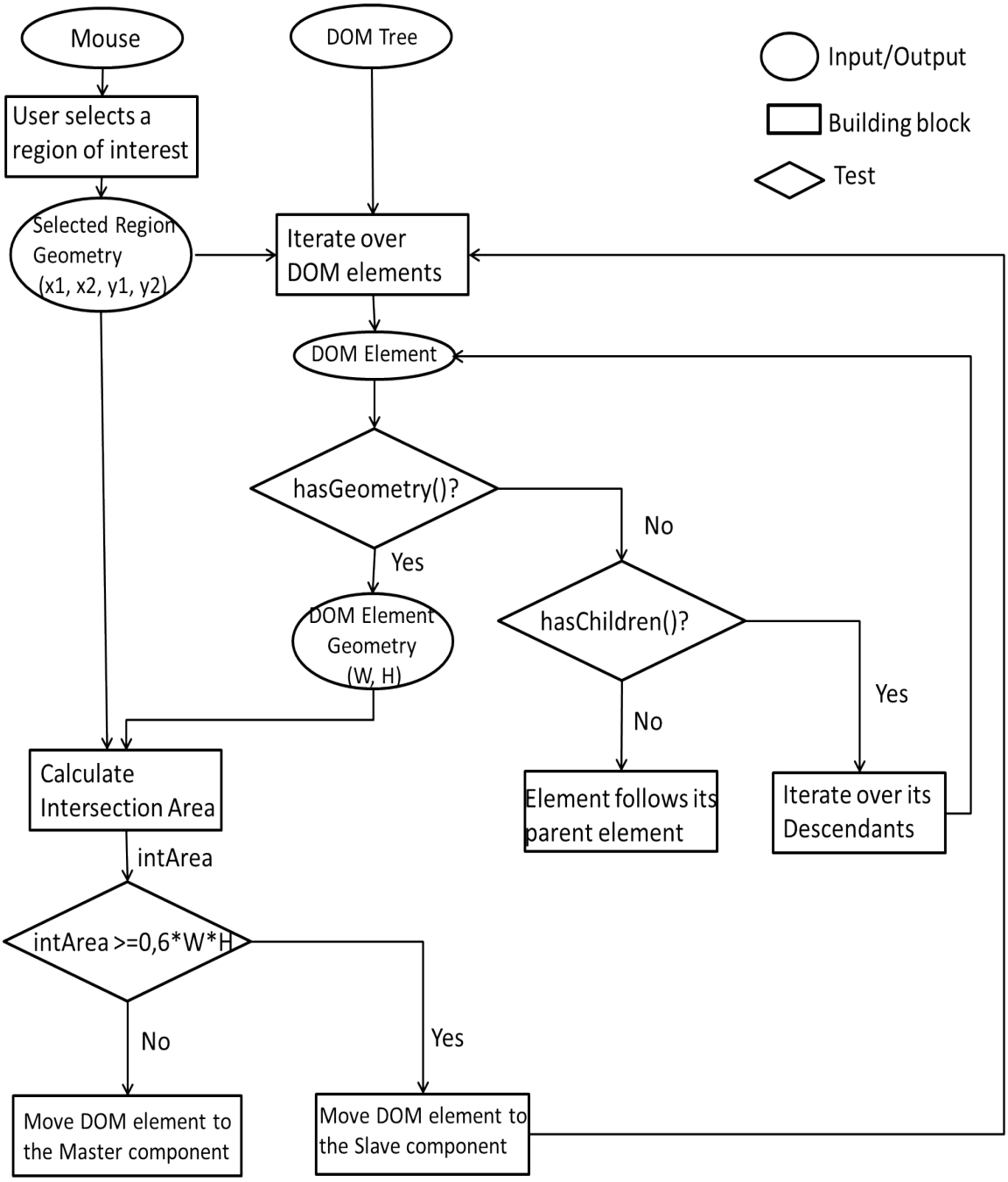


Figure 4-2: Screen-Region Selection Method

The algorithm main building blocks are: 1) traversing the DOM tree in pre-order, 2) checking each element geometry and 3) computing the intersection between the area corresponding to the DOM element and the area of the selected region.

If an element has no geometry, i.e., his height is null, the algorithm still processes the element descendants for the reasons described in Problem 2. If the element is a

leaf node, i.e., do not have a child, it is moved by default to the same component as its parent element. Otherwise, depending on the component(s) to which the descendants belong, the element follows the same component.

If an element has a geometry, the algorithm detects if the element belongs partially, completely or not at all to the selected region. To explain how the algorithm detects the intersection, we illustrate the different cases in Figure 4-3. The four blue blocks enumerated from 1 to 4 represent four DOM elements and the pink region is the selected region.

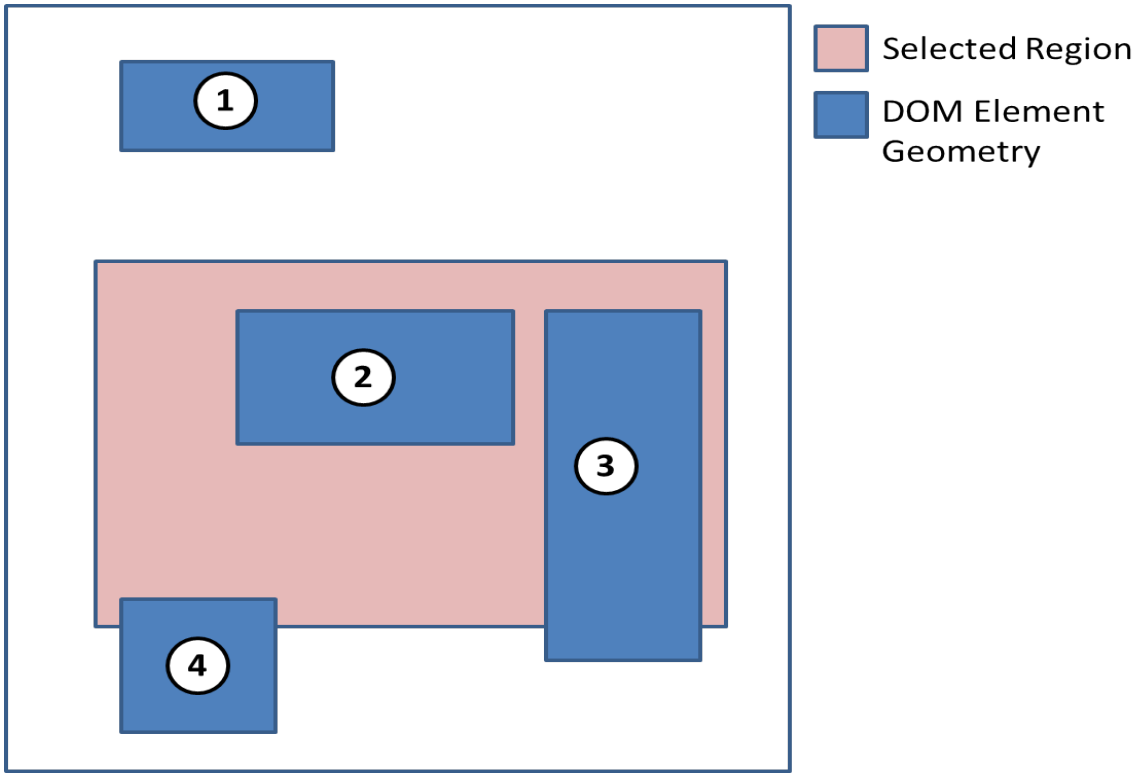
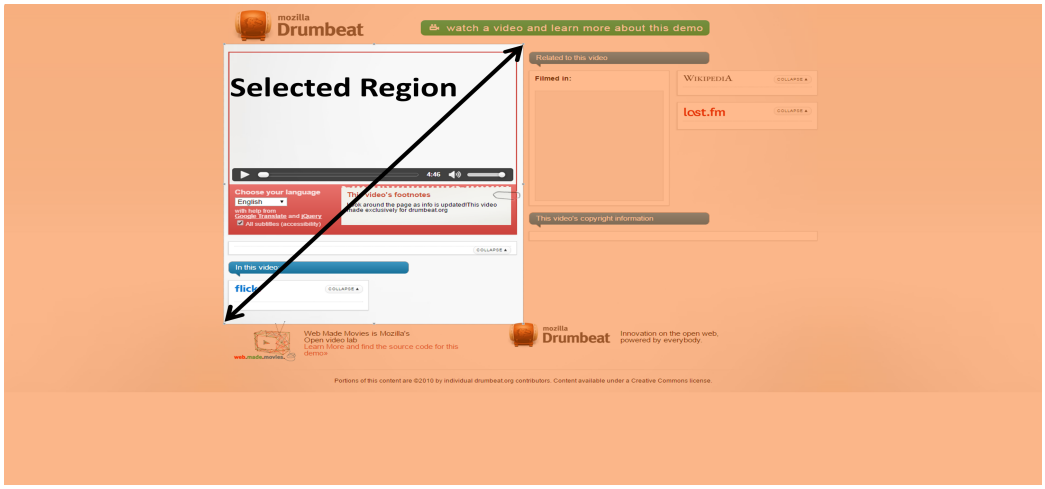
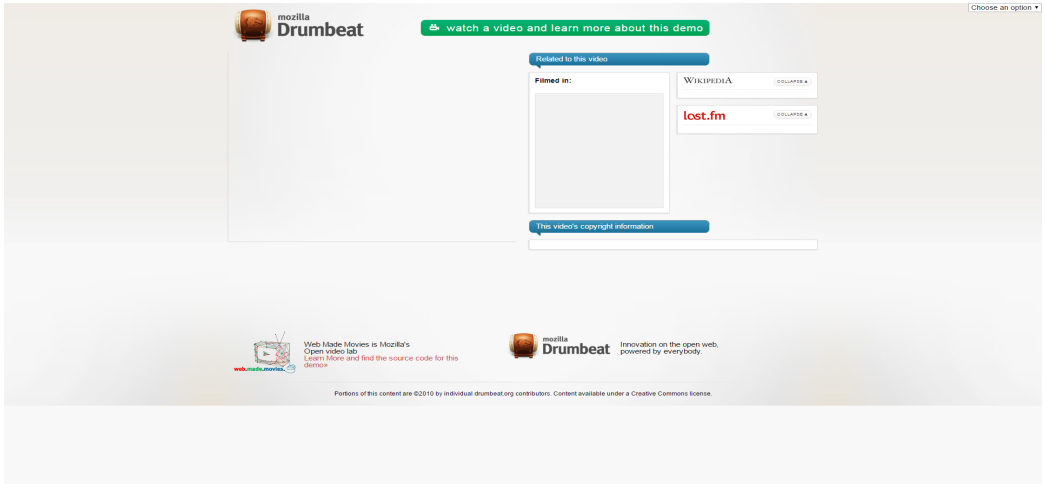


Figure 4-3: Calculating intersections between the selected region and DOM elements

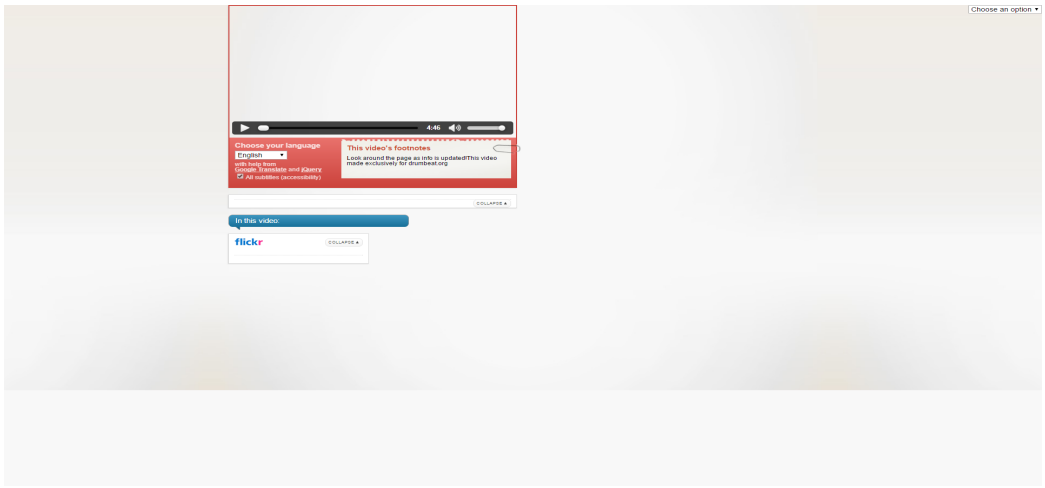
Block 1 is completely outside the pink region (intArea = 0), then it belongs to the master component. Block 2 is completely inside the pink region (intArea = 100%), then it belongs to the slave component. For those cases, the algorithm is straightforward. The complexity comes with elements that are partially inside the pink region, i.e., blocks 3 and 4. These cases are usually the result of the lack of precision when selecting the region of interest (Problem 1). To deal with this, the algorithm checks if the intArea covers more than 60% (a heuristic value based on our experience) of the DOM element area. If it is the case, as for block 3, then the element belongs to the slave component. Otherwise, it belongs to the master component, as it is the case for block 4.



(a)



(b)



(c)

Figure 4-4: (a) Selecting the right side of the VideoSemantic application (b) The resulting master component (c) The resulting slave component

Figure 4-4 illustrates an example for dividing the VideoSemantic application using the marquee tool for screen-region selection. Figure 4-4(a) shows the selected region containing the video element, its footer and the flickr box. Figure 4-4(b) shows the master component and Figure 4-4(c) shows the content of the selected region on the slave component.

4.1.2 Method limitations

This visual method has the advantage of abstracting from the complexity of the DOM structure to partition the user interface. It also follows the user preferences. But, it does neither consider the application content nor the environment features since it does not check whether an element is suitable for a certain device or not.

The selection is manual and requires the user interaction. The issue here is that on some devices, especially on a TV, the screen-region selection is not practical using a remote control. In consequence, this can limit the usability of the overall system.

To circumvent the two limitations related to the environment and to the system usability, we developed a primitive DOM-based analysis method that first characterizes the DOM elements and then partitions the DOM tree between the slave and the master components. Further details are found in Section 4.2.

4.2 DOM-based Division Method

4.2.1 Principles

The DOM-based division method is an automatic method that focuses only on the static and dynamic analysis of the structure and the content of the DOM tree. The purpose is to find the DOM elements that correspond to our two functions defined in Section 3.3.

The method requires first a study of the HTML5 elements as defined in the W3C specification to determine their roles and how they could be classified for the purpose of the application division.

WHATWG [2], W3C [3]and Mozilla [4] all categorize the HTML elements according

[2]WHATWG, `https://whatwg.org/`
[3]W3C Content models, `https://www.w3.org/TR/html5/dom.html#content-models`
[4]Mozilla content categories, `https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/`

to their content model. The content model refers here to a description of the element expected contents, i.e., the element children in the DOM tree. Examples for content models are Metadata content, Flow content, Sectioning content, Phrasing content, Interactive content, Paragraphs, Transparent content models, etc. Notice that one HTML5 element may belong to one or more content models. Among these content models, the two most relevant for our requirements concerning the content-device duality are: the Interactive and the Embedded contents. Interactive content is specifically intended for user interaction and it includes the following HTML5 elements: a, button, input, keygen, label, select, textarea, in addition to audio, video (with controls), embed, and img and iframe.

The Interactive content list as defined in W3C specification does not fit exactly into our "interactive" function. This is because the audio, video and embed elements are more "multimedia" than "interactive". An image and iframe are not "interactive" in our classification if no event listener is set on them.

"Embedded content", following the W3C specification is content that imports other resource into the document, or content from another vocabulary that is inserted into the document [5]. The associated element list includes but is not limited to the audio, video, object and embed elements that correspond exactly to our "multimedia" function. The remaining elements, i.e., canvas, iframe, svg, img and math, do not fit into any of our functions.

In consequence, we defined our own categories for visual elements as follows:

- **interactive** elements (e.g., a, area, button, datalist, form, input, keygen, textarea, nav, optgroup, option, output, select),
- **multimedia** elements (e.g., video, audio, object, source, track),
- **non-interactive, non-multimedia leaf** elements (e.g., caption, dialog, figcaption, h1 to h6, hgroup, img, kbd, label, legend, p, progress, span) and that may contain text nodes.
- **other grouping** elements that include the remaining HTML5 elements without the Metadata content and the Script-supporting elements (e.g., table, div, header, footer, article, etc.).

This categorization is a prerequisite for our runtime DOM-based analysis. If

`Content_categories`

[5] Embedded Content, `https://www.w3.org/TR/html5/dom.html#embedded-content-2`

one would want to extend our system, more refined categories could be added to correspond to other functions, such as “information".

The analysis of the DOM tree includes 1) a static analysis of the DOM element tag names and 2) a dynamic analysis of the DOM element behavior to check if event listeners are set to make them belong to the “interactive" category.

The algorithm iterates over each DOM element in the DOM tree and compares its tag name to those present in the “interactive" or in the “multimedia" categories.

- If the element falls within the first category, then it belongs to the slave component.
- If the element falls within the second category, then it belongs to the master component.
- If the element falls within the third category, the element is undecidable.
- If the element falls within the forth category, the algorithm iterates over its children first. If the all children belong to the same component, the element follows its children. Otherwise, if the children are divided between both components, then the element is duplicated on the master and the slave. If none of the children belong to any component, the element remains undecidable. If most children belong to one component and there are some undecidable children, the element follows the majority of its children.

Then, the algorithm detects any change in the element basic role by checking statically its attributes, mainly declarative event listeners (e.g., ‘onclick’). Then, it checks dynamically whether an event listener is added using JavaScript. If it is the case, the element belongs to the slave component.

4.2.2 Limitations

At the end of this analysis, only part of the DOM tree could be divided between the master and the slave component. The remaining elements, especially some leaf nodes, remain undecidable while they usually take part in the application graphical user interface, notably the non-interactive non-multimedia elements.

One solution would be to cluster or aggregate DOM elements following the parenthood and sibling relationships, but this can yield to an unsatisfactory UI division.

This is explained by the fact that these structural relationships do not necessary imply visual/geometrical relationships if CSS is used to layout elements. For instance, we consider two siblings in the DOM tree; using CSS, we make one element float to the top left of the window and the other to the bottom right of the window. Structurally they can be aggregated since they are siblings but visually they should not.

Adopting the structural analysis makes it easier to consider the functions derived from the environment features but it is very limiting for the above mentioned reasons. Thus there is a need to consider the element geometries and their visual effects to divide the application user interface.

After implementing and experiencing with the two above methods, we decided to abstract the analysis from the DOM tree and the visual block concept that were introduced in the screen-region selection method. In addition, we kept our HTML5 content categorization that was introduced in the DOM-based analysis method. By combining these ideas and concepts, we designed the hybrid segmentation method described in the following section.

4.3 The hybrid segmentation method

4.3.1 Principles, challenges and overview

The segmentation we consider here is an automatic and a hybrid segmentation method guided by the multi-screen environment based on 1) the visual analysis using the rendering tree that is associated to the DOM tree, 2) the structural analysis of the DOM tree and 3) the content function analysis. We recall that the content functions, derived by the environment, reflect the type of interaction between an end-user and a block of content.

The two main requirements here are 1) to identify in the application structure the elements that correspond to the content functions provided at the input, following Section 4.2 and 2) to build visual and semantic blocks around these elements.

Generally, a segmentation process assumes that a big block is decomposed into sub-blocks of smaller sizes. In our approach segmenting an application based on its structural analysis works the opposite way. The analysis starts from the particles, i.e., DOM elements, that we try to aggregate into bigger blocks.

Two main challenges appear here. First, how to determine the point at which we

should stop the elements aggregation. Second, the number of DOM elements that we have to deal with varies and can be large, as web page statistics indicate[6]. In addition, not all DOM elements are visible or relevant to our segmentation.

We decided first to simplify the DOM tree to only keep the graphical DOM elements while giving special attention to those that are representative of our functions, provided at the input. Afterwards, the segmentation takes place on the simplified tree to identify the function blocks. Finally we project back the segmentation result to the DOM tree. Both steps are represented in Figure 8-2 and will be described in the following sections.

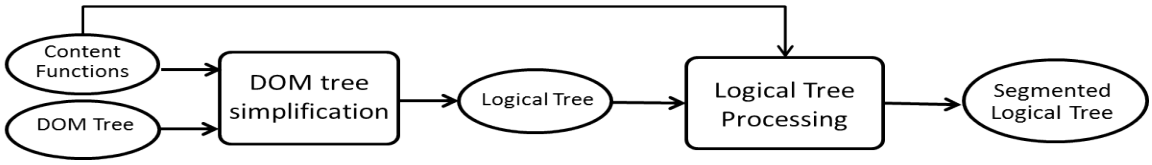


Figure 4-5: Building blocks for the segmentation algorithm

4.3.2 DOM tree simplification and labeling

The aim of simplifying the DOM tree is to keep only the elements that form the user interface. By doing so, the number of elements we are dealing with is reduced. The simplification process is inspired by BoM [46]. It takes the DOM tree as an entry and delivers a logical tree representing the user interface structure that is partially labeled with functions. A pre-requisite for this phase is having the application rendered on the browser, not necessarily full-rendering (pixels). In this case, the browser computes automatically the content geometry based on CSS and it creates a rendering tree. Using this information provided by the browser, we can focus on the segmentation process.

Table 4.1: DOM node types and associated logical nodes

DOM node Types	Visibility	Function	Associated Logical node	Label
Relevant	yes	yes	yes	Function
Visible	yes	no	yes/no	no
Non-relevant	no	no	no	no

[6]See `http://httparchive.org`

We assume that in general a DOM tree consists of three types of nodes, i.e., “relevant elements", “visible elements" and “non-relevant elements", as shown in Table 4.1. Each type can contain any DOM node no matter its position in the DOM tree, i.e., leaf node or grouping node. We characterize these types as follows:

1. **“relevant elements"** correspond to nodes that have a visual effect (e.g., visible) and have a function, e.g., a video element or one of the interactive element types.
2. **“visible elements"** correspond to nodes that have a visual effect but have no function, e.g., a visible paragraph containing text.
3. **“non-relevant elements"** correspond to nodes that have no visual effect, e.g., hidden nodes.

The first step for the simplification process is thus to classify the DOM tree elements of the current application. The classification is based on 1) a visual and geometrical analysis, 2) a static and a dynamic analysis of the DOM elements behavior. The visual and geometrical analysis checks if a DOM element is visible on the screen and has dimensions. The static and the dynamic analyses are the same of Section 4.2 and aim at identifying the function of the relevant elements.

A first traversal of the DOM tree is conducted to identify the DOM elements that have by default a function following our categories in Section 4.3.1. This is done based on the static and dynamic analysis of the DOM elements. As a result, the DOM tree is partially annotated because not all elements belong to our two main categories, i.e., interactive and multimedia.

Similar to BoM [46], a second pre-order traversal of the DOM tree is required to start the simplification process and to create an intermediate structure between the DOM tree and the logical tree. We call this structure the geometric tree. This second traversal could be merged with the first traversal, but for separation of concerns we preferred to separate them. Another classification is applied here to eliminate DOM nodes that are “non-relevant" whether because they are comment nodes, metadata, style, scripting tags, text nodes with white spaces, carriage return tags, etc., or whether they are nodes with no geometry (i.e., their area is null). This classification is described in Table 4.1. We note that we took the liberty to adjust these categories especially because BoM [46] bypasses the HTML img and video tags from its categories, while their presence is necessary for our approach. As a result, the geometric tree represents both the visible DOM nodes and their corresponding geometry in one

structure. The geometric tree is more representative of the DOM tree than the logical tree is, as we are going to see below. Each geometrical node contains information about its content category, including our 'interactive' and 'multimedia' categories, and the content geometry.

The geometric tree is then used to create the logical tree. The geometric tree is traversed in pre-order starting from the root. For every geometric element, depending on its category and its visibility, the algorithm decides on the creation of a logical node as follows and as shown in Table 4.1.

- A "relevant element" is retained to form a logical node labeled with its corresponding function. In this case, the corresponding logical node is a leaf node since the algorithm does not iterate over the descendants of its geometric node.
- A "visible element" forms a non-labeled logical node. Similar to BoM [46], one exception happens for a visible element that does not have siblings and that only has one children. In this case, we do not create an associated logical node.

  The corresponding geometric subtree is then processed iteratively.
- A "non-relevant" element does not form a logical node.

At the end of the simplification process, the logical tree contains a reduced number of elements compared to the DOM tree and it is partially labeled with functions. The efficiency of the simplification is evaluated quantitatively later in Section 4.6.1.

Every logical node is characterized with an id, a pointer to the corresponding DOM node, layout information and the associated function if any.

Some of the logical leaf nodes are labeled, their number is big as shown in Figure 4-6. In addition, they form geometrically small blocks in most cases.

To reduce their number, the logical tree is optimized again to form geometrically bigger labeled blocks whenever it is possible. The optimization procedure traverses the tree from the root to the leaf nodes in a breadth-first manner, and acts as follows:

- If a node is labeled, the algorithm checks whether its next sibling is labeled with the same function. If positive, the algorithm merges them to form geometrically a bigger node. The result of the merging is one logical node with updated information, i.e., an updated list of DOM nodes to include all DOM nodes of both logical objects, updated function if necessary and updated layout information. After analyzing all siblings, if only one labeled child remains, we

propagate its label to its parent. If negative, i.e., the sibling has no similar function, the algorithm continues to the next sibling.

- If a node is not labeled, we move down into its subtree to search for a possible optimization.

At the end, the output of this optimization is a logical tree with a smaller number of nodes but with bigger geometry. It should be noted that some leaf nodes may still be non-labeled, and grouping nodes are not labeled.

4.3.3 Segmentation: Processing the simplified tree

The segmentation phase consists in producing labeled blocks from the partially-labeled logical tree. In order to understand the difficulty of the segmentation we can consider two extreme cases. A segmentation that produces one block from each

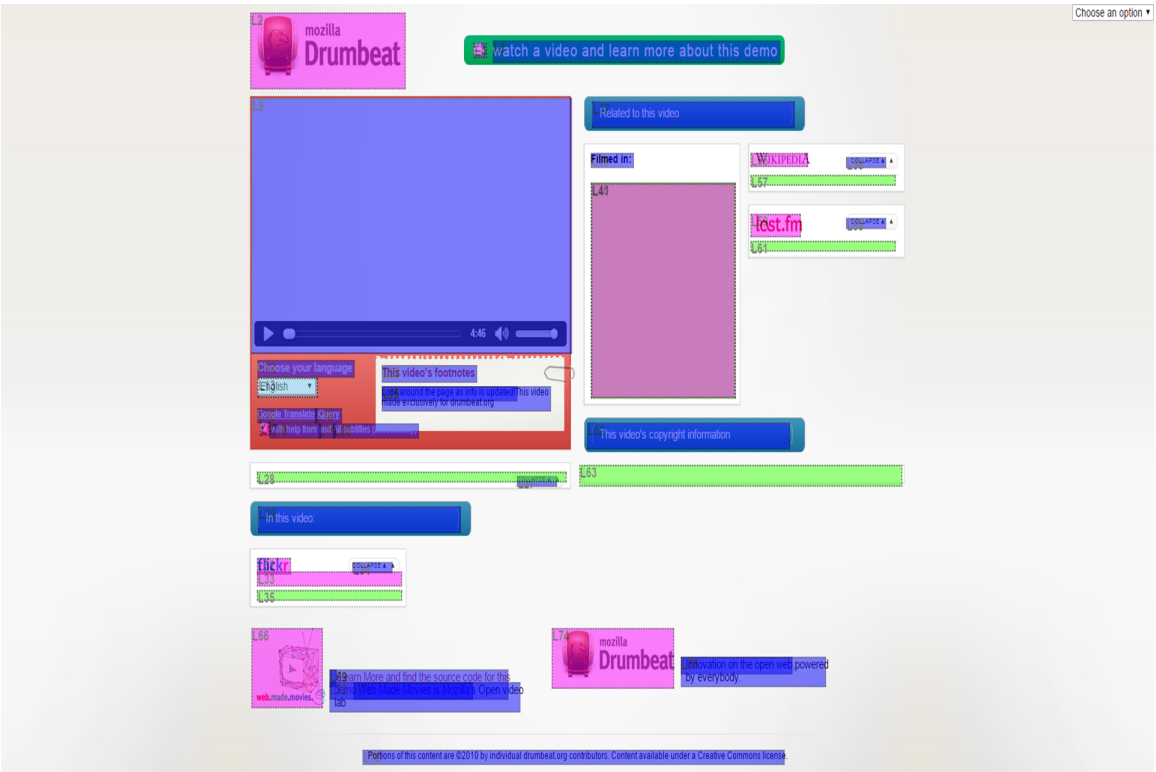


Figure 4-6: Trivial Segmentation leading to an excessive number of blocks

logical leaf node, results in creating an excessive number of blocks as Figure 4-6 shows for the VideoSemantic application. A trivial segmentation without constraint,

that produces one block from all the logical tree, is not useful for the user interface distribution between two components.

To get a better segmentation, we impose our own constraints for the processing of the logical tree as follows.

Two logical nodes are part of one independent block (1) if they are siblings in the logical tree, (2) if they satisfy Gestalt laws that "prescribe for us what we are to recognize as one thing" [42] and that are based on the proximity, similarity, closure and simplicity and (3) if they are neither labeled with more than one function, nor their descendants are.

Similar to BoM [46], we adopt the notion of granularity parameter (pG). The granularity parameter determines the area under which a logical node can be considered as a final block. This can be applied only if its descendants verify condition (3). In contrast to BoM where the pG value is a constant value set by the user, our approach consists of calculating the pG values automatically and continuously during the logical tree processing based on the dimensions of the labeled logical nodes. This continuous update of the pG value adapts the processing of the node subtree to its content. In our work, we consider the notions of global and local pGs. The global pG is set before starting the segmentation based on the labeled nodes of the entire tree. The local pG is updated during runtime, as described in the next paragraph, to adapt the segmentation of the node subtree to its content.

Both the global and the local pG are computed by considering the geometry of the labeled descendants respectively in the entire logical tree and in the local subtree as follows: For all the labeled nodes, we compute the ratio of their areas to the relevant page area. We define the relevant page area as the rectangular area defined by the top-left corner of the page, a width equal to the page width, and a height set to the minimum between the page height and five times (i.e., a heuristic value based on our experience) the screen height.

The pG value of a subtree corresponds to the biggest calculated ratio in the subtree, or to the global pG if the subtree does not have labeled descendants. Intuitively, the bigger the pG, the fewer final blocks are produced and the better the segmentation results are as shows Figure 4-8. Figure 4-7 presents a page with an illustration of all its logical nodes and their corresponding areas expressed in percentage, relatively to the complete page. Figure 4-8(a) presents the segmentation results of the reference page with a big pG value, i.e., 60%. We consider that the green and the pink colors correspond to two different functions. The segmentation resulted in two large

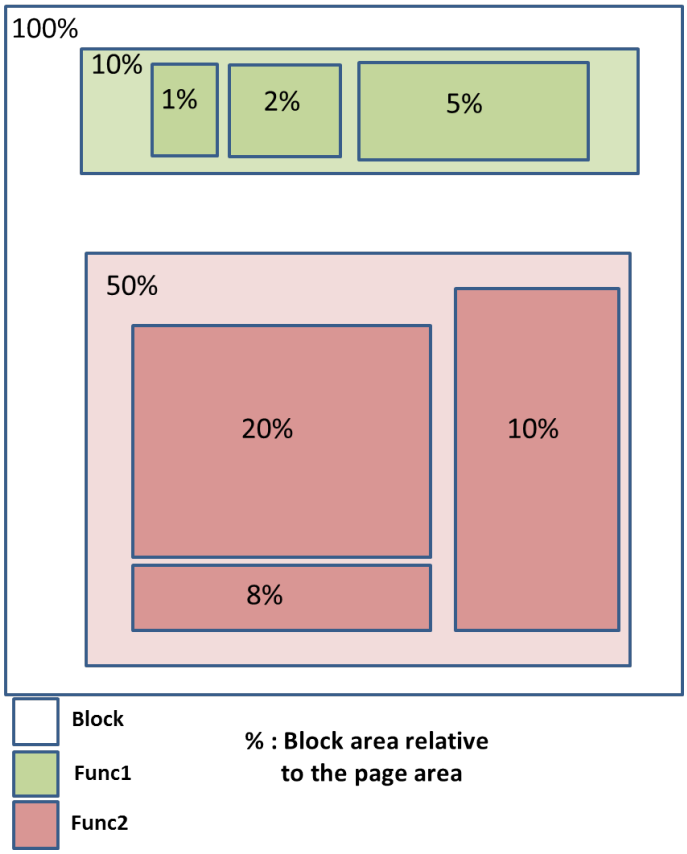
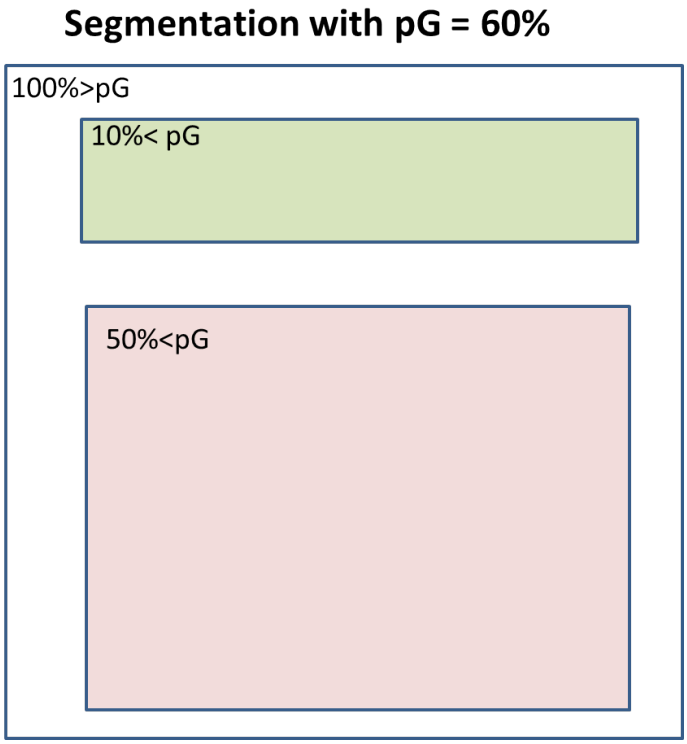
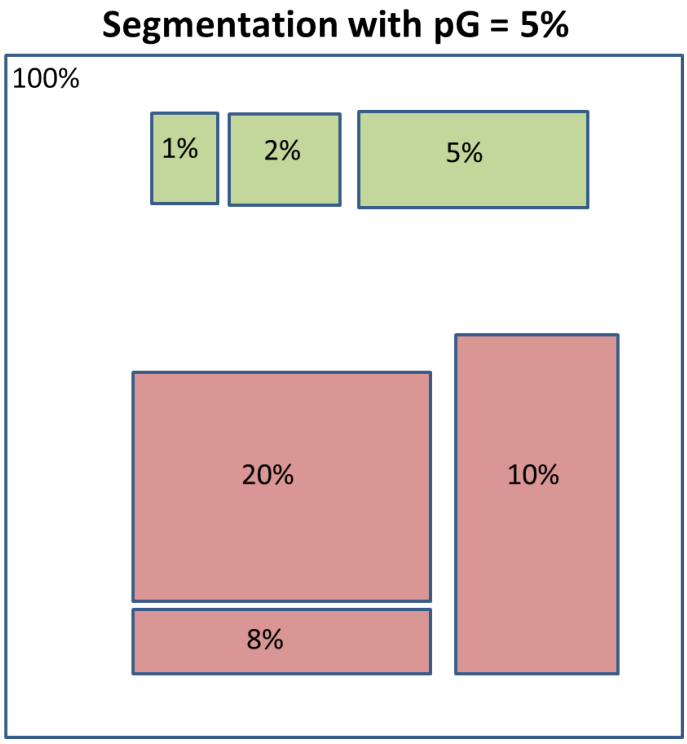


Figure 4-7: A reference web page with a representation of all its logical nodes and their corresponding relative areas.



(a)



(b)

Figure 4-8: (a) Segmentation results with a big pG value, i.e., 60% (b) Segmentation results with a small pG value, i.e., 5%

blocks, both blocks with an area smaller than pG. In Figure 4-8(b), the pG value is very small, i.e., 5%. Thus, the segmentation lead to segment the biggest green block and the biggest pink blocks, resulting in 6 smaller blocks comparing to Figure 4-8(a). Note that this example is a simple case where the blocks with different functions are not mixed.

Then, we proceed with the processing of the logical tree starting from the root node and using the global pG value, as follows: (1) If a node is labeled, we try to merge it with its siblings, as described below. (2) If a node is non-labeled and its descendants have different labels, we process its subtree first. (3) If a node is non-labeled, its descendants have only one function and its relative area is bigger than the pG, then we process its subtree; Otherwise, if its relative area is smaller than the pG, we investigate the possibility of merging it with its siblings.

We try to merge a node with its next siblings, as follows: if the node does not have any sibling, it produces a block. Otherwise, for each sibling, if one of the functions of the sibling descendants is different from the current node function, the nodes are not merged even if the Gestalt laws and geometrical conditions are satisfied, and the current node produces a block. Otherwise, if the functions are the same, the merging of nodes is tested using the Gestalt laws and the geometrical conditions, as in literature [46]. At the end of the merging, at least one labeled block is produced.

At the end of the processing, the leaves of the processed logical tree represent the final blocks that constitute the application user interface. All leave nodes are labeled and ready for the distribution in contrast to the grouping nodes that remain non-labeled.

4.4 User Interface distribution: The DOM Distribution

The user interface distribution represents the time at which the single-screen application turns into a functional multi-screen application with a distributed user interface between the master and the slave components as Figure 4-9 shows. More precisely, this phase has two objectives: to build the HTML documents for each component and to prepare them for the runtime environment.

This section focuses on the first objective since this chapter is dedicated to the work on the DOM tree. The second objective is treated separately in Chapter 6.

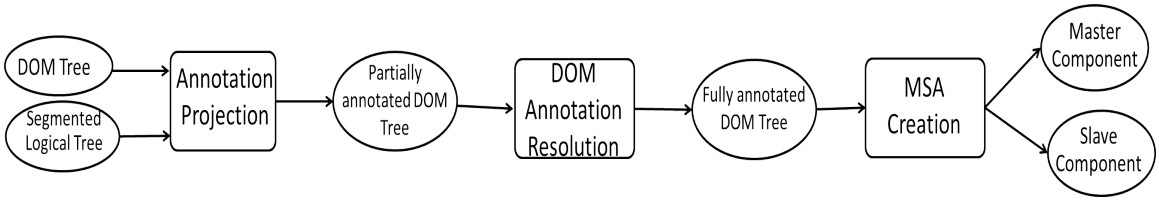


Figure 4-9: Building blocks for the UI Distribution phase of the refactoring system

Depending on the UI division method that is used, the UI distribution part takes either two lists of DOM elements to be associated to each component or a segmented and labeled logical tree.

In the first case, the DOM distribution is straightforward and starts with the DOM annotation resolution in Section 4.4.2. In the second case, an intermediate step, described in Section 4.4.1, for label projection is required to move the work from the logical tree to the DOM tree. This projection determines for each DOM element in the DOM tree whether it is part of the slave or the master HTML document, and whether it should be monitored for changes during run-time.

In both cases, the DOM distribution basis is the same; the leaves that share the same label, i.e., 'interactive' or 'multimedia', create respectively the slave and the master user interfaces.

For the remaining sections, we only consider the hybrid approach for dividing the user interface since it takes part in the final version of the refactoring system.

4.4.1 Annotation Projection from logical tree to DOM tree

Listing 4.1: Example of a HTML document

```
1  <body>
2    <header  id="header">
3      <a href="link1.html"> this is link1 </a>
4      <img id="logo" src="img.ext"></img>
5    </header>
6    <div id="container">
7      <video  src="videosource.ext" style="width:530px;height:299px
           "></video>
8      <div id="videocontrols" onclick="playpause()"></div>
9      <div id="subtitles"  style="
           position:absolute;width:530px;top:370px;left:417px">
10       <div id="subtitle-0" style="display:none;"></div>
11       <div id="subtitle-1" style="display:none;"></div>
```

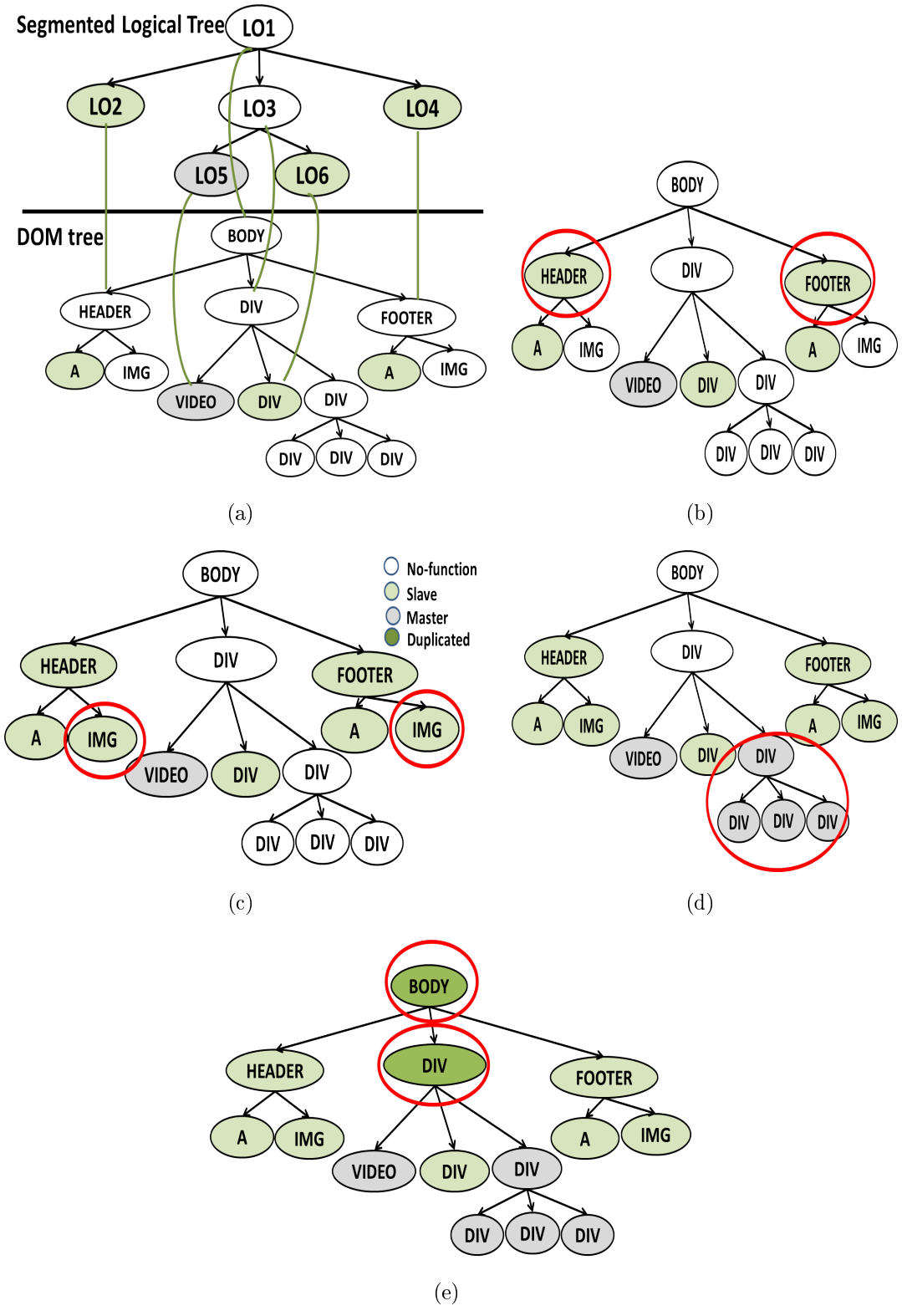


Figure 4-10: (a) DOM simplification into logical tree (b) Function projection from logical tree into the DOM tree (c) Function resolution of DOM descendants (d) Function resolution of DOM siblings (e) Function resolution of DOM antecedents

```
12          <div id="subtitle-2" style="display:none;"></div>
13        </div>
14      </div>
15      <footer  id="footer">
16        <a href="link2.html"> this is link2 </a>
17        <img id="copyright" src="rights.ext"></img>
18      </footer>
19  </body>
```

The projection algorithm uses the fact that each logical node has an explicit correspondence with a DOM node in the form of a property illustrated with green lines in Figure 4-10(a). Figure 4-10(a) illustrates the DOM tree associated to the HTML document in Listing 4.1 as well as the resulting segmented logical tree on the top of the figure.

The projection starts from the leaves of the logical tree in a depth-first post-traversal manner. We set three straightforward rules for the function projection on the DOM tree, as follows:

1. DOM elements associated to logical nodes with a function 'multimedia', represented in Figure 4-10 with the grey color belong to the master (e.g., the video element)

2. DOM elements associated to logical nodes with a function 'interactive', represented in Figure 4-10 with the light green color, belongs to the slave, (e.g., the header and the footer elements with all their descendants)

3. DOM elements whose direct children correspond to logical nodes of different functions are shared between the master and the slave components. These are colored in dark-green and surrounded with red circles in Figure 4-10(e).

At the end of the projection, we have the DOM tree of the main application partially annotated and the annotated elements are sparse in the DOM tree. This is because not all the DOM nodes are represented in the logical tree and because among those that are represented not all of them are linked to a leaf logical node that has a function. Figure 4-10(b) shows that the header and footer are annotated as a result of the projection. By opposition to the annotation of the logical tree, after this step any element (either a leaf or a grouping node) may be annotated.

4.4.2 DOM Annotation Resolution

For the rest of this section we focus on the DOM tree and specifically on the annotated elements that are the key to resolving the DOM annotations. For each element having initially a function, referred to as the 'center', we apply the function resolutions in three steps. We first deal with descendants, then with siblings and finally with parents respectively as follows:

- resolving the annotation of its sub-tree,
- resolving the annotation of its geometrical siblings first, especially because the segmentation approach does not cover the case of overlapping elements. Second, the algorithm resolves the annotation of its structural siblings to cover all the DOM tree, and
- resolving the annotation of its antecedents if possible.

The 'center' descendants inherit by default their parent function as shown in Figure 4-10(c) since they belong to the same user interface block as it is the case for the descendants (i.e., img) of the header and the footer elements represented in Figure 4-10(c).

For elements that do not have a function, the algorithm iterates over each of its siblings and it first checks if the sibling geometrically overlaps the 'center'. [7] If it does, the element gets the same function as 'center'. In Figure 4-10(d), the div element with 'id = subtitle' overlaps the video element and as a consequence it obtains the 'multimedia' function that it passes as well to its descendants.

If the sibling does not overlap any element, then it gets by default the function of the first element to its left in the tree. [8]

The parent of the 'center' iteratively creates a record of its children functions. Once all his children get a function, the parent resolves its own as follows: if all children have the same function, the parent adopts it. If there are multiple functions, then the parent is shared between the master and the slave. For instance in Figure 4-10(e), the circled 'container' div (resp. body) is a shared element since it has descendants of different functions.

[7] Note that this test is not valid for the pure structural UI Division method.

[8] Note that if an element has no 'center' sibling, then the algorithm moves upwards to resolve the function of the parent element.

4.4.3 Creating the master and the slave components

The third step is the production of the master and the slave HTML documents following the application model introduced in Section 3.4.4. To produce the master application, the system works on the main application and makes the 'interactive' DOM elements 'hidden' using CSS 'visibility:hidden' property-value pair.

To create the slave application, elements having the 'interactive' function and those that are common with the master component are extracted and serialized to the new slave component running on the secondary device [9].

To this level, we have produced the master and the slave HTML documents. But, the master and slave can neither communicate with each other, nor work in complementary fashion. This will be explained later in Chapter 6.

4.5 Summary

Table 4.2: Complexity of each algorithm in the UI Division and UI Distribution phases

Algorithms	Number of Tree traversals		
	DOM	Geometrical	Logical
DOM tree simplification and labeling	2	1	1
Processing the logical tree	-	-	2
Projection	-	-	1
Propagation	1	-	-
MSA Creation	1	-	-

In Table 4.2, we summarize the main steps of the hybrid UI Division and the UI Distribution phase. We focus on showing for each step the concerned trees and the number of traversals for each of them.

The DOM tree simplification requires traversing twice the DOM tree: once to hook the content elements that correspond to our functions, and another time to create the geometric tree. The geometric tree is then traversed once to create the

[9]Note: Our system proposes as well an option to download each of the slave and the master HTML documents. This is useful if the system is used by a multi-screen application developer, and also for testing purposes.

non-optimized logical tree. This latter is also traversed once to optimize the logical tree in order to prepare it for the segmentation.

Processing the logical tree, i.e., the segmentation, requires three traversals of the logical tree: one to calculate the relative areas of all logical nodes in order to set the global pG value, one to determine if a node has descendants with different funcitons, and another traversal is required to aggregate the logical nodes while annotating them with our functions.

The projection step consists in projecting the annotations from the logical tree to the DOM tree. Thus, it requires one traversal for the logical tree.

The propagation works only on the DOM tree and aims at fully annotating it while traversing it once. Finally, MSA creation requires one traversal of the DOM tree in order to serialize the slave (resp. master) content to the slave (resp. master) component.

In our work, the 'interactive' contents are always assigned to the slave and the 'multimedia' contents are always assigned to the master component that contains the main logic. The latter assumption is crucial if the state of an element depends on more than what is stored in the DOM tree (e.g., canvas, videos with buffered data, etc.). An example could be an application that has its contents dependent from the state of the video element (e.g., the video current time). For an application like Youtube, where the page content is not updated accordingly to the video element, the master could have either multimedia or interactive content.

No matter which video player (HTML5 video, flash, or other players) is used in an application, the video state is managed natively inside the browser and exposed partially in JavaScript. Every video player offers a JavaScript object representing the video element and its state. This state is not expressed inside the DOM tree.

The problem of moving the video to the component that does not have the main logic (neither the video object), is that it is not possible to track its state changes. In consequence, the linked content on the other components will never receive any update. One solution for HTML5 videos is to extend the native methods to capture the video state and redirect it to the other component.

4.6 Segmentation Evaluation

In this section, we evaluate quantitatively the two phases of the hybrid segmentation method, i.e., DOM tree simplification and the logical tree processing.

The first objective is to check how efficient the simplification is by measuring the number of nodes and the tree depth. To this end, some statistics are collected from the tested DOM trees and from the logical trees before and after the segmentation of our dataset applications (Section 3.2). Section 4.6.1 presents the statistics.

The second objective is to check whether the segmentation results respect and satisfy our constraints. For this reason, we compare the segmentation results to a ground truth and to one segmentation method from the literature. Sections 4.6.2 and 4.6.3 describe the methodologies and the comparison results.

4.6.1 Efficiency of the simplification method

Tables 4.3 and 4.4 present the statistics obtained on our dataset of existing sites after applying the DOM tree simplification and after segmenting the logical tree. Table 4.3 contains the average number of nodes (resp. the depth) of each of the DOM, the geometric, the logical and the segmented logical trees. Table 4.4 presents the reduction percentages 1) of the DOM tree relatively to the geometric tree, 2) of the DOM tree relatively to the main logical tree and 3) of the logical tree relatively to the segmented logical tree.

The number of DOM nodes varies between 254 and 3411 nodes with an average of 1133 nodes. The number of geometric nodes varies between 9 and 2203 nodes with an average of 622 nodes. It is notable that in the case of JWPlayer demo3, the number of geometrical nodes exceeds the number of DOM nodes. This is because during the creation of the geometric tree, text nodes are wrapped into HTML span elements in order to compute their geometry. This leads to the increase in the number of visible DOM nodes, thus the increase in the number of geometric nodes and finally the one-level increase in the tree depth. The number of logical nodes varies between 17 and 346 nodes with an average of 90.5 nodes.

Table 4.4 shows that the average reduction rate of node count is only 27% when moving from the DOM tree to the geometrical tree. The average reduction rate of node count is of 86% when moving from the DOM tree to the initial logical tree. Comparing these two reduction rates, we remark that the geometric tree is closer to

Table 4.3: Results of the simplification algorithm on the geometrical and logical trees in terms of node and depth count

Applications	DOM tree		Geometrical tree		Logical tree		Segmented Logical Tree	
	Nodes	Depth	Nodes	Depth	Nodes	Depth	Nodes	Depth
Viewster	684	27	576	27	39	9	29	3
Vimeopro	113	9	103	9	17	3	13	2
Vimeo	1083	18	950	19	346	9	78	5
Youtube	3263	23	2203	23	71	9	39	6
Dailymotion	1172	17	794	15	147	9	74	8
Yahooscreen	1284	21	768	20	46	7	27	6
Twitch	809	10	528	11	71	6	42	4
Liveleak	631	17	521	17	109	6	38	3
Ustream	1051	17	264	14	80	8	37	5
Break	3411	20	1974	20	47	6	42	6
Metacafe	449	16	348	12	127	8	27	4
VideoJS	175	8	62	8	18	3	14	1
Jplayer demo1	31	8	34	8	13	4	7	1
Jplayer demo2	36	8	37	8	14	4	8	1
JWPlayer demo1	19	4	9	3	8	2	8	2
JWPlayer demo2	324	12	178	11	8	2	8	2
JWPlayer demo3	1091	5	2145	6	1080	4	8	2
Media Elements	104	7	70	5	25	3	17	2
Video player pages	254	7	362	7	166	3	10	1.6
Video Semantic	528	14	112	11	52	8	29	5
Average	1133	16.6	622	13	90.5	7	37.8	4.5

Table 4.4: Reduction rates for DOM tree, geometric tree and logical tree

Applications	DOM-Geom Reduction(%)		DOM-Log Reduction(%)		Log-Log Reduction(%)	
	Nodes	Depth	Nodes	Depth	Nodes	Depth
Viewster	16	0	94	67	26	67
Vimeopro	9	0	85	67	24	33
Vimeo	12	-6	68	50	77	44
Youtube	32	12	98	61	45	33
Dailymotion	32	12	87	47	50	11
Yahooscreen	40	5	96	66	41	14
Twitch	35	-10	91	40	41	33
Liveleak	17	0	83	65	65	50
Ustream	75	18	92	53	54	38
Break	42	0	99	70	11	0
Metacafe	22	25	72	50	79	50
VideoJS	65	0	90	63	22	67
Jplayer demo1	-10	0	58	50	46	75
Jplayer demo2	-3	0	61	50	43	75
JWPlayer demo1	53	25	58	50	0	0
JWPlayer demo2	45	8	98	83	0	0
JWPlayer demo3	-96	-20	1	20	99	50
Media Elements	33	29	76	57	35	43
Video player pages	12	6	63	53	35	43
Video Semantic	78	21	90	45	44	37.5
Average	27	6	86	56.5	46	35

the DOM tree than to the logical tree. This means that on average only 14% of the DOM elements are sufficient to represent the user interface. Reducing the number of elements of the main application refines the segmentation phase since only relevant elements will be considered for the segmentation.

The number of logical nodes at the end of the segmentation varies between 7 and 74 with an average of 37.8 nodes (see Table 4.3), leading in Table 4.4 to an average reduction rate of 46% of logical nodes comparing to the initial logical tree. This means that on average only 3.3% of the DOM elements are represented in the segmented logical tree.

The DOM tree depth varies between 7 and 27 levels with an average of 16.6 levels. The geometric tree depth varies between 9 and 27 levels with an average of 13 levels. The logical tree depth varies between 3 and 9 levels with an average of 7 levels. The DOM-Geom depth reduction is 6% on average. The DOM-Log depth reduction is 56.5% on average. Reducing the hierarchical structure of an application approaches us to what an end-user perceives exactly in the application since the leaves of the logical tree represent the application content.

The logical tree depth at the end of the segmentation varies between 2 and 8 with an average of 4.5 levels, leading to an average reduction rate of 35% of the tree depth. With a total reduction of 80% in the tree depth, the logical tree approaches from the application user interface.

4.6.2 Qualitative evaluation: Comparing to BoM

In this Section, we compare the results of our segmentation method to that of BoM [46].

The methodology consists in applying the two segmentation methods on the same application and in illustrating the results in the form of visual blocks on top of the application content. We selected a Youtube page [10] that is a very complex application in terms of number of DOM nodes and in terms of page size as shown in Table 4.3 with 3263 DOM nodes and 23 levels of hierarchy.

Figures 4-11 and 4-12 present segmentation results of the YouTube page, respectively using BoM and our hybrid method. Note that we cropped the comments section for a better illustration.

[10] http://bit.ly/1eue6i3

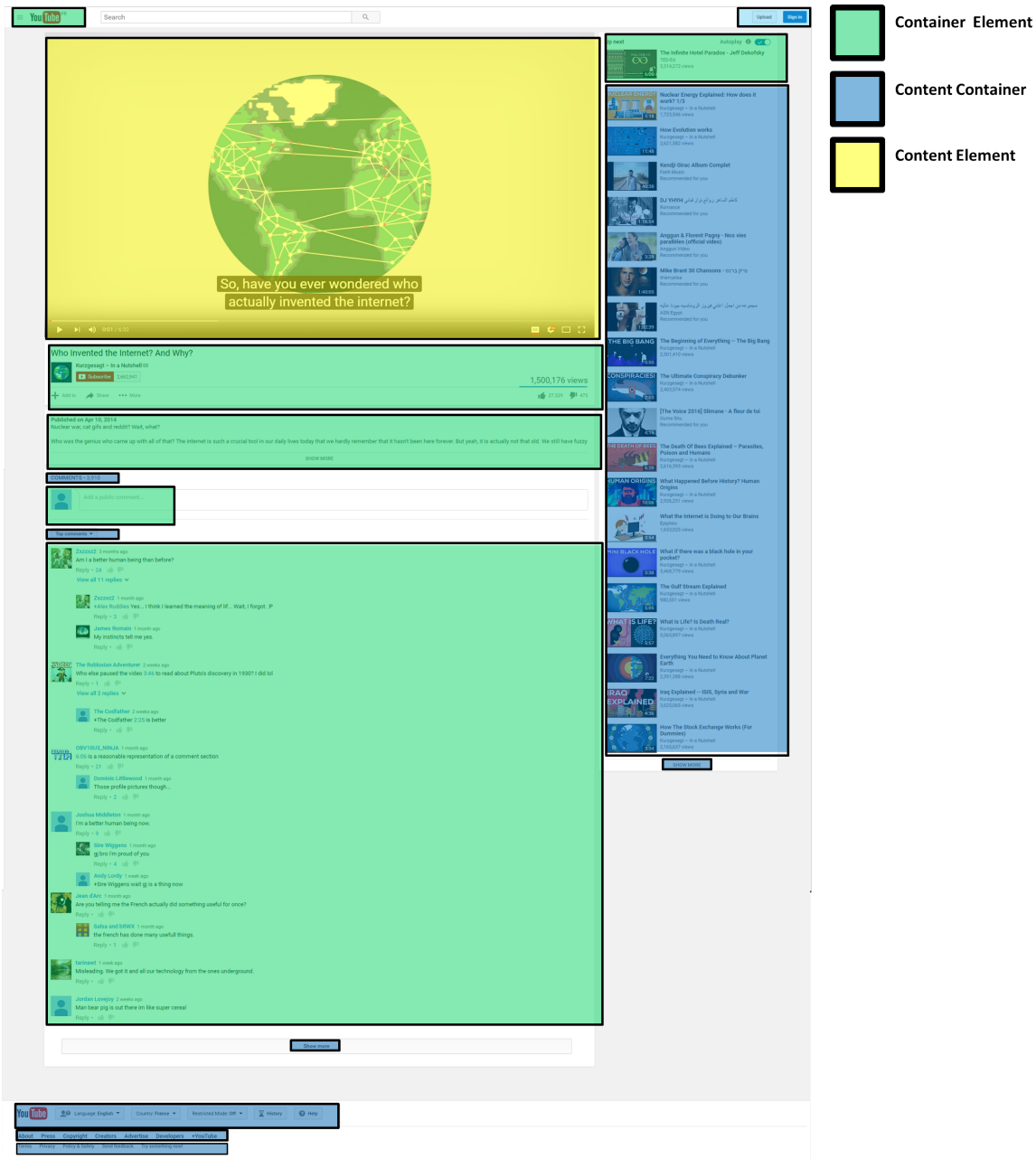


Figure 4-11: Segmentation results on a YouTube page: BoM with pG = 0.31

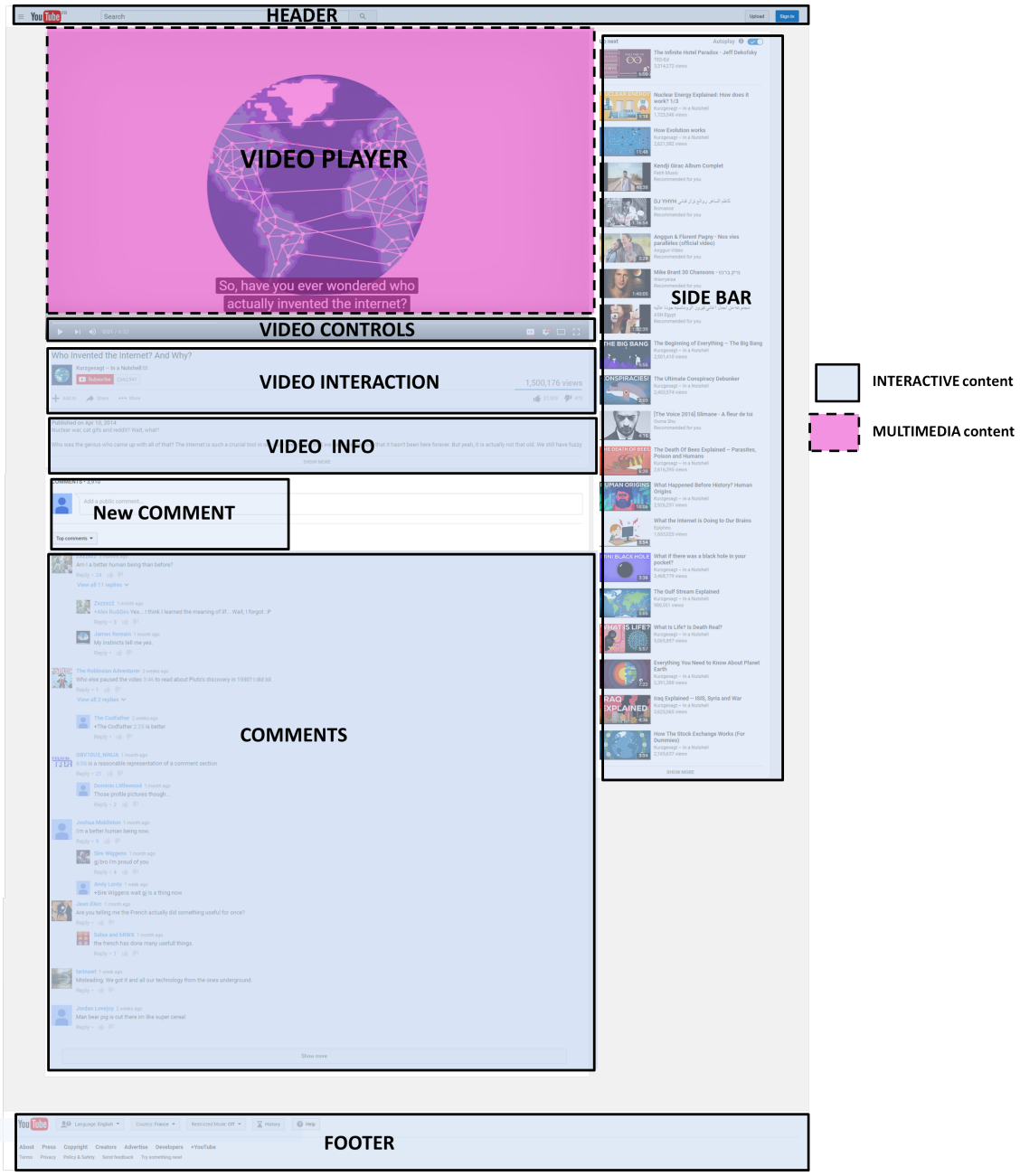


Figure 4-12: Segmentation results on a YouTube page: MSoS with pG = 0.31 and 0.36

Figure 4-11 represents the results segmentation using BoM with a pG value set manually to 0.31. Note here that in Figure 4-11 the block colors are specific for BoM and they express its classification for DOM elements. The yellow color reflects content elements (e.g., span, a, li, h1-5, video etc.), the green color represents the container elements (e.g., ul, p, table, section, header, footer, etc.) and the blue color represents content container. Following BoM, a content container is a container element that all its descendants are content elements. The video element was not considered initially in BOM classification, but we added it to the list of content elements.

Figure 4-12 represents the results using our hybrid segmentation. The light-blue blocks refer to interactive blocks and the unique purple block refers to the multimedia content. During the segmentation, only two different pG values were computed: 0.36 (global and local) and 0.31 (local). Most of the logical nodes were processed with the 0.31 value, this is why we decided to configure BoM with this value.

Using BoM, the segmentation generated 16 blocks while using our method the segmentation generated 9 blocks. The HEADER block in Figure 4-12 corresponds to two blocks in Figure 4-11. The SIDE BAR block corresponds also to two blocks in BoM results. The NEW COMMENT block corresponds to three blocks in BoM results. In contrast, the VIDEO block with the VIDEO CONTROLS block correspond to one block in BoM. This shows that content functions were taken into consideration during our segmentation. Note that video subtitles are judged as multimedia content since they overlap the video element. The COMMENTS block corresponds to two blocks in BoM results. Finally, the FOOTER block corresponds to three blocks in BoM results.

Our segmentation method resulted in a reduced number of blocks and most of these blocks are self-descriptive since they contain all the content.

In this example, we see that our method ensured the separation of blocks with different functionality, thus facilitating the content mapping to the 'best-match' device in the context of multi-screen environment.

4.6.3 Quantitative Evaluation: Comparison to a ground truth

In this section, we evaluate quantitatively the quality of the segmentation method by comparing the segmented applications from our dataset to a ground truth.

In the context of our work, the segmentation quality depends on two aspects. The first aspect is related to the visual coherence of the resulting blocks. The visual

coherence means that every resulting block corresponds to a block that is naturally identified by the human eye.

The second aspect is related to the correctness of the function attributed to each block.

In the following Section 4.6.3.1, we quantify the segmentation quality using some metrics and we briefly describe how the ground truth was created. The comparison results and their interpretation are presented in Section 4.6.3.

Note that ground truth and results are accessible from our site [11].

4.6.3.1 Creating the ground truth and our metrics

The ground truth (GT) was created manually, where coherent blocks were determined and assigned a function between 'multimedia' and 'interactive'. Afterwards, we compare our segmentation results to this GT. We provide the comparison results in Table 8.2 in the form of precision and recall metrics. We define the precision and recall metrics as follows:

$$Precision = \frac{Nb\ of\ Matching\ Blocks}{Nb\ of\ Resulting\ Blocks} \tag{4.1}$$

$$Recall = \frac{Nb\ of\ Matching\ Blocks}{Nb\ of\ GT\ Blocks} \tag{4.2}$$

Recall is equal to one if the segmentation algorithm could identify correctly all the blocks of the GT. Precision is equal to one if our segmentation algorithm did not produce any non-matching block. The non-matching column refers to the number of blocks that: 1) are over-segmented by the segmentation algorithm, i.e., when 1 block in the GT corresponds to multiple blocks in our results, 2) have no correspondence with any block in the GT (i.e., extra blocks or concatenated blocks) or they are not correctly labeled.

In our case, the over-segmentation does not present a problem as long as all the resulting blocks have the same function. Only the block function assignment affects the distribution of the graphical user interface, i.e., the percentage of non-related blocks from Table 8.2. Moreover, this category covers the absence of a function, which is not even a big issue in our work especially if it is related to a content that

[11] See `http://download.tsi.telecom-paristech.fr/gpac/MSoS`

is neither multimedia, nor interactive. This is because during the distribution phase, our system is capable of resolving the lack of functions as described in Section 4.4.2.

4.6.3.2 Results and interpretation

Applications	Precision	Recall	Non-Matching	
			Over-Segmented	Non-Related
Viewster	0.4	0.8	0.2	0
Vimeopro	1	0.71	0	0
Vimeo	0.13	0.33	0.5	0.03
Youtube	0.86	0.83	0.125	0.125
Dailymotion	0.174	0.8	0.2	0.22
Yahooscreen	0.38	0.83	0.17	0.23
Twitch	0.67	0.89	0.11	0.04
Liveleak	0.875	0.7	0	0.06
Ustream	0.38	0.625	0.25	0.155
Break	0.33	0.75	0.25	0.055
Metacafe	0.36	0.37	0.33	0.09
Social pages	0.6	0.7	0.2	0.08
VideoJS	1	1	0	0
Jplayer demo1	1	1	0	0
Jplayer demo2	0.5	0.5	0	0.25
JWPlayer demo1	0.5	0.5	0	0.25
JWPlayer demo2	0.67	1	0	0.165
JWPlayer demo3	1	1	0	0
Media Elements	0.44	0.67	0.33	0.11
Video player pages	0.73	0.81	0.05	0.11
Semantic Video	0.71	0.83	0.07	0.035
Average	0.63	0.75	0.12	0.09

Table 4.5: Evaluation of the segmentation approach

Looking at Table 8.2, we can see that our system performs quite well as the percentage of non-related blocks is between 0 and 0.25, on average 0.09. The value

of 0.25 for the video player pages mostly corresponds to blocks for which no label is assigned in applications where there was a small number of blocks.

Table 8.2 also shows that the calculated metrics are coherent for the three sets of applications, independently from the application height or the number of DOM nodes. The precision rate is the lowest for the social applications with 0.6 comparing to the video player and to the semantic video applications with respectively 0.71 and 0.73.

The Vimeo application has the lowest precision and recall values (resp. 0.13 and 0.33), but it also has the lowest rate of non-related blocks (0.03). The problem here is that the computed pG was small enough to lead to a high over-segmentation (0.5).

Most applications from the video player category are simple and mostly composed of a video element, a custom control bar and in some cases a subtitle area, etc. The segmentation results of these applications show high precision and recall values (resp. 0.73 and 0.81). This indicates that our algorithm is capable of separating the control bar from the video rendering part.

For the video semantic application, the precision rate is 0.71 indicating that most of the GT blocks were identified by our algorithm, even for this complex application.

Though the average number of the over-segmented blocks is small for the video player pages and for the video semantic application (resp. 0.03 and 0.05), its value is important for social pages (0.2). The over-segmentation is the drawback of calculating the value of the granularity parameter according only to the labeled nodes, see Section 4.3.3.

4.7 Conclusion

The user interface division and distribution that are described in this chapter are the key components of our refactoring system.

The interface division decides on the content that belongs to each of the master and the slave components. While multiple approaches were tested in this chapter to divide the DOM tree, only the hybrid approach provided a complete solution. The complete solution includes 1) the usage of the environment features to divide the DOM tree and 2) the abstraction from the complexity of the DOM structure and conducting the segmentation following the visual and geometric aspects.

The performance evaluation of this hybrid approach included:

- the comparison of the segmentation results on a Youtube page to those of a segmentation method from the literature, and
- the comparison of the segmentation results on our dataset applications to a ground truth manually created.

In both cases, the evaluation shows that our segmentation method respects the system requirements described in Chapter 3 and that the resulting blocks are visually coherent in most of the cases. This does not preclude the production of extra non-relevant blocks, but their number is small.

Our content classification and division does not take into consideration container elements that can embed external elements (e.g., iframes) or elements that mix different languages (e.g., svg elements). To consider iframes in our division algorithm, we have to analyze the nested document inside the iframe. But for security reasons, Cross-Origin problems appear if the main HTML element and the nested document belong to different domains. In the other case were both documents belong to the same domain, we have observed that in existing applications iframes are used to embed advertisement content or to embed a video player. In these cases, there is no need to divide the iframe document as long as we identify its function in the page (e.g., advertisement, video, etc.). In addition, elements used to draw graphics on-the-fly, as it is the case with canvas are challenging for division. The canvas contents are not part of the DOM tree and a canvas has no DOM descendants. This means if we want to split a canvas, we should think about analyzing its corresponding JavaScript object. This can be an interesting future work, i.e., splitting canvas elements.

In this chapter, the distribution of the user interface is limited to the DOM tree distribution. The distribution includes as well the transparent transformation of the logic and the adaptation to the COLTRAM platform that are described later in Chapter 6. At the end of the complete distribution, each of the master and the slave components is created and run on its corresponding device.

Running on a new environment, the two components will confront layout anomalies especially given that the CSS documents are not distributed. As a solution, the component layouts are also refactored to adapt to the screen dimensions, as it is described in the following Chapter 5.

Chapter 5

Layout Refactoring

This chapter focuses on the Layout Adaptation phase in our system architecture in Figure 8-1.

The chapter starts in Section 5.1 with an overview of the layout problems that are caused by the application distribution and relates these problems to each of the master and the slave components.

In Section 5.2, we describe globally the basis of our layout refactoring solution that aims at resolving the identified layout anomalies.

Section 5.3 explains the Full-Window Design solution that is a first implementation of the global solution for the master component. It aims at making the master component exploit the large display of the master device by making its content stretch to occupy the whole display.

Section 5.4 is another implementation of the global solution, but it is dedicated for the slave component. In contrast to the master component, the slave runs on a small device with a small screen size. The aim of this implementation is to optimize the user experience to avoid the horizontal scrolling in favor of the vertical scrolling.

For each of the two layout problems, a set of relevant metrics is identified and used to evaluate the two solutions for the master and the slave components.

Finally, a conclusion is provided in Section 5.5.

5.1 Effects of content distribution on the application layout

We have identified two factors that influence the result of the application layout in a distributed environment.

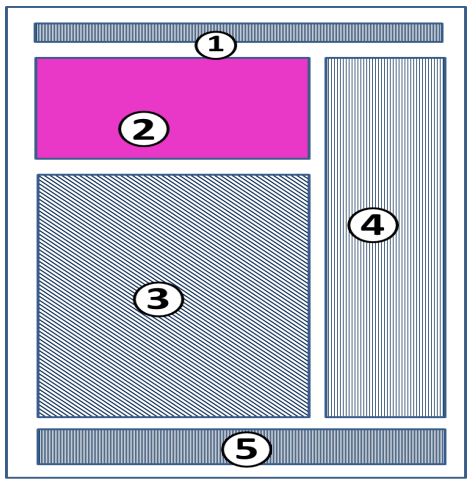
The first factor is related to the environment and to the multiple devices on which the application blocks are going to be laid out. This is true especially if the main application was intended for one specific type of devices; desktop applications are good for large screens but not good enough for small screens.

The second factor is related to the content distribution itself irrespective of the device characteristics. Indeed, distributing an application results in two or more applications, each having fewer blocks than the single-screen application.

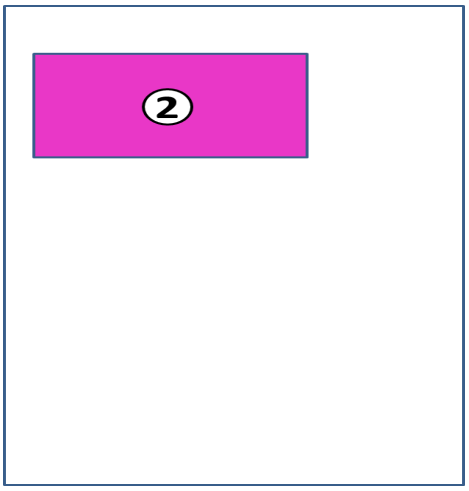
Depending on how blocks were initially positioned relative to each other in the main application, those selected for each candidate device can suffer from layout anomalies once rendered alone on that device. Examples of layout anomalies are: 1) layout discontinuity, 2) misplacement or wrong re-arrangement of blocks, 3) horizontal scrolling. Figure 5-1 illustrates these anomalies on a reference layout of a single-screen application. As depicted in Figure 5-1(a), the application has five blocks enumerated from 1 to 5. This numbering follows the reading order, as defined by Faraday [16], moving from left to right, and from top to bottom. The importance of the reading order is that it expresses how a user perceives the different parts of an application. Layout anomalies are detailed hereafter.

5.1.1 Layout Discontinuity

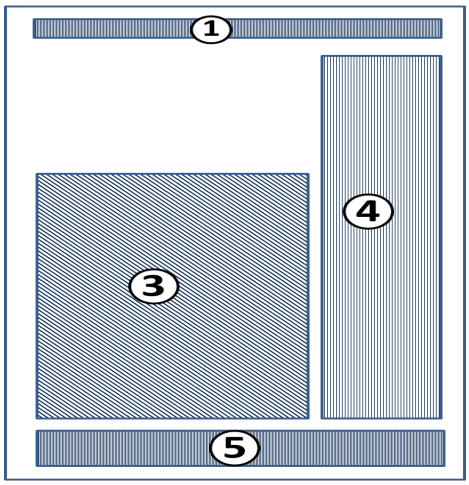
A layout discontinuity corresponds to large empty blocks between content blocks or around a unique block in the page. We define a “blank space" as a region on the screen that does not correspond to any foreground element in the DOM tree. For instance, the region that separates the visible blocks from each other is a “blank space". This blank space is caused by the padding or margin properties of the DOM elements, floating elements, hidden elements using the CSS ’visibility’ property, etc. It can also be the result of rendering a content that has a fixed and static layout on a large-screen device. In such a case, the content fails to utilize the available space. The blank space can be represented by a white space, or it can have a background color or a background image.



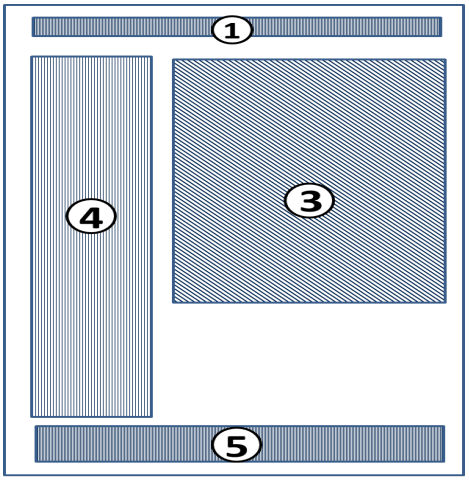
(a)



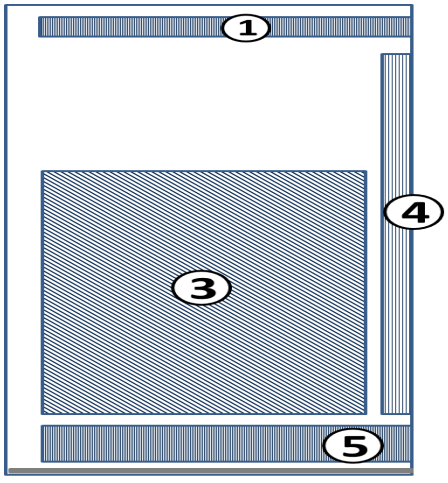
(b)



(c)



(d)



(e)

Figure 5-1: Layout Anomalies: (a) Main Application (b) Master Layout Discontinuity (c) Slave Layout Discontinuity (d) Misplacement (e) Horizontal Scrolling

Figure 5-1(b) represents block number 2 displayed alone on the page and surrounded with blank spaces. Figure 5-1(c) represents all the blocks of the main application except block number 2. This last one is replaced with a blank space in the center of the page. In both cases, the blocks do not occupy all the available space of the browser window. This can be the result of the split described in the application model in Section 3.4. In order to propose a better rendering and a more acceptable layout, we want to re-dimension and re-position the blocks to optimize the screen occupancy and to better exploit the totality of the device screen. Inspired by the full-screen API [1], Section 5.3 presents our solution for this layout discontinuity problem.

5.1.2 Invalid content re-arrangement

A wrong or invalid re-arrangement of blocks appears when at least two blocks of content are mapped to a sub-application and when their new dispositions break the original reading order. Figure 5-1(d) illustrates this anomaly where block number 4 is placed before block number 3, thus breaking the reading order.

This anomaly normally appears when the block positions are not set by the application designer as fixed positions using absolute units (pixels or points), but instead when they depend on the disposition of previous blocks or next blocks which is often the case in applications using CSS.

These blocks, once rendered alone, should respect the reading order of the main application to ensure the application understandability even after distribution. The challenge here is to first identify the reading order and then to respect it during the layout re-design. The reading order identification is part of the block identification mechanism described in Section 4.3. Sections 5.3 and 5.4 represent our solutions to avoid the wrong content re-arrangement.

5.1.3 Horizontal scrolling

Another problem can appear when large and wide blocks are attributed to devices with a small screen size and when the layout is not capable of adapting to the screen size, i.e., case of non-responsive or non-adapted applications. This is illustrated in Figure 5-1(e) where blocks 1 and 5 are partially shown on the page and block 4 is

[1] Full-screen API, `https://developer.mozilla.org/en-US/docs/Web/API/Fullscreen_API`

almost absent from the application view.

This results in a user experience that requires continuously the horizontal scrolling in order to see the whole content. Based on user tests, Zorrilla et al. [61] claim that horizontal scrolling causes an unpleasant user experience. Therefore, we want to make sure that the blocks of content adapt their positions and dimensions to the screen size while always having the application content readable, especially on small screens.

The challenge is to dynamically adapt the layout of an application that targets one device, to multiple devices. Section 5.4 presents our approach that is based on the dynamic design of responsive web applications.

5.1.4 Summary

In this chapter, we propose to solve the problems induced by the content distribution and by the multitude of devices on which the components are going to run.

The solution consists in refactoring the layout for a better rendering experience. We consider two types of layout refactoring: one that avoids the horizontal scrolling mainly on small devices, i.e., second-screen devices, while being responsive; and another that optimizes the content occupancy especially on primary devices. In both cases, the content disposition always follows the application reading order.

In the following section, we present the common building blocks for the refactoring solutions, including the inputs, output and the common logic.

5.2 Overview of Layout Re-factoring

The layout refactoring consists in adjusting the content disposition without affecting neither the content itself, nor its functionality, nor its reading order.

In general, a layout defines the placement and the dimensions of DOM elements on a web page using cascading style sheets. But, as stated earlier, working with the DOM tree is challenging especially given that not all of its nodes are visible. As we have seen in Section 4.3, we have the logical tree representation, which is an abstraction of the DOM tree and which contains information about the reading order of the content. However, this representation is not sufficient to work on the layout for two reasons:

- its structure is not close enough to the DOM tree, in terms of number of DOM nodes and depth of the tree as it was shown in Table 4.3. Figure 5-2 shows an example of this problem on the DOM tree of Listing 5.1. The DOM tree consists of three visible nodes on three level of hierarchy. Similarly the geometric tree has three geometric objects in contrast to the logical tree that has only two nodes on two levels hierarchy. We recall that this is one rule in the construction of the logical tree, that consists of eliminating an intermediate parent in the case where the number of nodes is equal to the number of the hierarchy levels in the geometric tree.

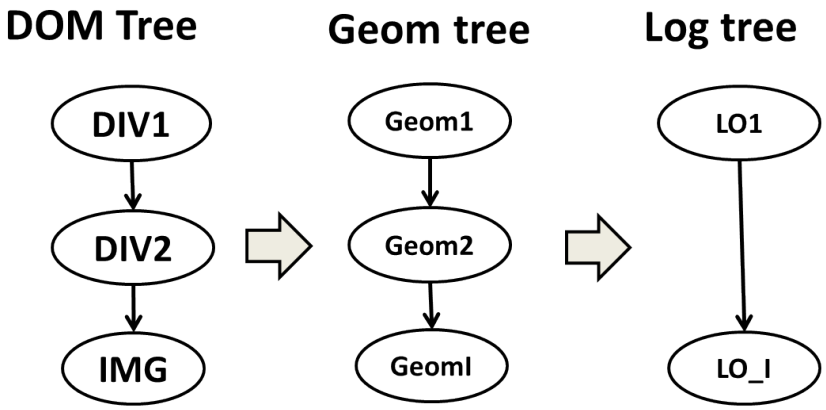


Figure 5-2: Simple example illustrating the drawbacks of the logical tree comparing to the geometric tree

Listing 5.1: CSS styling rules

```
<style>
  #div1 {
    width:500px;
  }

  #div2  {
    width:80%;
  }

  img{
    width:50%;
  }
</style>
<div id='div1'>
  <div id='div2'>
    <img id='icon' src='icon.png'></img>
  </div>
```

```
18 </div>
```

- the parent-children relationships in the DOM tree need to be preserved; otherwise the designed style will not give the expected layout. Listing 5.1 is an example illustrating the importance of respecting the parent-children relationships. Div1 has a width of 500px. Div2 has a width expressed in percentage, i.e., 80%. In CSS, percentages define sizes in terms of parent objects, e.g., Div1 width. Thus, the width of Div2 is equal to 400px. Div2 contains an icon image with a width equal to 50%, i.e., 50%*400px = 200px.

As mentioned in [46], the logical tree construction also produces an intermediate structure called the geometrical tree, whose structure is very close to the DOM tree. For each visible DOM element, a geometric object is associated. To preserve the reading order and to respect the parent-children relationships during the re-design, we use both structures, the logical and the geometrical trees, as inputs for the layout refactoring phase.

Figure 5-3: Common building blocks for layout refactoring algorithms

The refactoring consists of two main phases: resetting and re-design, as shown in Figure 5-3. The layout resetting presented in Section 5.2.1 ensures that the new layout will take effect on the application and will not be inhibited by the old layout style sheets. The layout re-design whose basis is described in Section 5.2.2 automatically identifies the layout changes that should take place on the normalized application depending on the re-design objectives.

These changes are then applied to the normalized application and as a result a re-designed application is delivered.

5.2.1 Layout resetting principles

The layout resetting aims at eliminating or neutralizing the CSS properties that can throw off the redesign algorithm. Without this preliminary phase, the CSS styling rules that are dynamically created at the end of the layout refactoring process could

be inhibited. This is caused by two mechanisms specific to the rendering engine of a web browser: the style sheet cascade order and the specificity.

The first mechanism is the style sheet cascade order. Declarations for style properties can appear in the form of (1) styling rules and (2) simple properties defined inside the HTML document. Styling rules can be defined explicitly in the HTML document in the form of HTML style tags, or implicitly with links to external CSS files. The simple properties can be only defined inside the HTML style attributes for a specific HTML element. These declarations can appear simultaneously in several style sheets or several times inside a style sheet. The cascade order determines the order of applying the style properties depending on their origin and their precedence.

The second mechanism computes the specificity that determines which styling rule is going to apply in the case of several declared styles. It consists of assigning a weight for each utilized CSS selector. By comparing these weights, the CSS rule with the highest weight is applied. CSS properties defined as inline styles have the highest specificity.

Dealing with these two mechanisms is a complex task. To avoid this complexity, we filter out the styling rules after we have created the logical tree and we only select the relevant properties (i.e., to which our algorithm is going to assign a new value). We classify the relevant properties as follows:

- The sizing set that includes only the 'width' property and that indicates the width of an HTML element.
- The offset set that includes 'margin', 'margin-left' and 'margin-right' properties that indicate the amount of white space around the box model of an HTML element.
- The position set that includes 'position', 'float', 'left' and 'right' properties that indicate the position of an HTML element and its floating type and amount.

Then for each filtered property we normalize it by setting its value to the empty value. An empty value lets the browser attribute the initial/inherited value for each property. By doing this, we ensure that our new layout 1) will be applied and 2) the tasks for the rendering engine after applying the new rules will be alleviated [7] since there will be only one CSS declaration defined in one style sheet for the above listed properties.

An alternative for normalizing the layout would be to remove radically all styling

rules and all properties and to produce our new design. This implies more effort since we need to filter out the layout to backup the properties that have no relation with element geometries, to save them and then create new styling rules. This work is error prone and the produced CSS styling rules may not be optimized if they are created automatically.

5.2.2 Layout re-design principles

As previously mentioned, we implemented two solutions for the layout re-design that fulfill different objectives but share a common logic. In this section we focus on this common logic.

In the early days of HTML/CSS design, HTML tables have been misused by application designers to design layouts using HTML. Tables organize their content inside cells, thus resulting in a good looking display. Recently CSS3 came along and provided new means of organizing the content on the page in the form of grids without using HTML elements. In either case, the spatial distribution of the content can be represented in the form of a grid that contributes to optimizing the space occupied by content.

We use this to automatically re-design the application layouts following a grid-based template. In our approach, grids are dynamically derived from the content geometry as described below.

The identification of a grid follows a set of rules or constraints. A grid cell can contain at maximum one geometric block, but a geometric block can span horizontally and vertically multiple cells. This means that there might be some empty cells in the grid, that are useful for the master Full-Window Design (FWD) presented in Section 5.3. In addition, cells can have non-uniform dimensions.

The grid identification problem can be simplified as the identification of the horizontal and the vertical separators between geometric blocks and to positioning these blocks in the grid.

To identify the separators, we can equally start with the horizontal or the vertical separators. For horizontal separators, we iterate over all the application blocks in the order of their appearance on the screen, i.e., from top to bottom and from left to right, and we test for each of them whether they are horizontally separable from the remaining blocks. Two blocks are horizontally separated if the biggest ordinate of one block is smaller or equal to the smallest ordinate of the other, and vice versa.

If this condition is satisfied, we consider the maximal ordinate between the top left corner of each block as the separator.

If not, we try to identify a vertical separator between them. The same reasoning is applied to identify the vertical separators. The number of columns in the grid is equal to the number of vertical separators augmented by 1. The number of rows in the grid is equal to the number of horizontal separators augmented by 1.

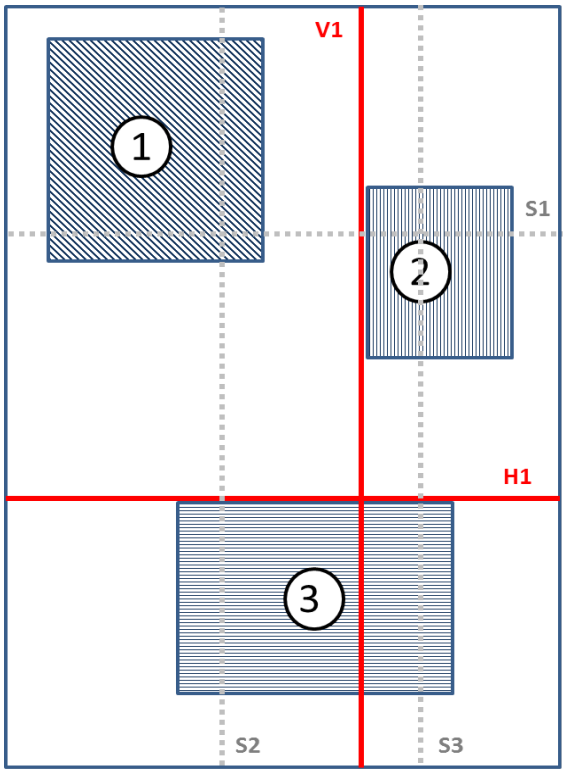


Figure 5-4: Vertical and Horizontal separators

Figure 5-4 represents 3 blocks for which we identify the horizontal and vertical separators. Blocks 1 and 2 are both horizontally separable from block 3, thus H1 is considered as a horizontal separator. In contrast, blocks 1 and 2 are not horizontally separable since there is at least one line, e.g., S1 that passes simultaneously through both blocks.

Blocks 1 and 2 are vertically separable and the corresponding vertical separator is V1. In contrast, none of block 1 or block 2 is vertically separable from block 3 since line S2(3) passes simultaneously through block 1(2) and block 3.

We position each object within its cell, relatively to the vertical and horizontal separators. As an output, we get a grid representing the spatial distribution of the application blocks as well as blank spaces representation. Some cells are empty and

there may be blank space around the blocks of occupied cells. In our example, we identified a 2*2 grid. Block1 and Block2 occupy one cell each in the first row, while block3 occupies 2 cells in the second row.

After identifying a grid and depending on the re-design objectives i.e., eliminate the blank spaces (Section 5.3) or eliminate the horizontal scrolling (Section 5.4), a set of re-design rules are applied on the application to reconfigure its layout.

5.3 Master Adaptation: Full-Window Design for large devices

5.3.1 Overview

The Full-Window Design approach aims at exploiting the available blank spaces on the master application, that runs on large-screen devices.

To remedy the excess of blank spaces, our solution takes the set of blocks intended for the master application, i.e., logical objects labeled with a 'multimedia' function and the associated geometric tree. The main requirement here is to deliver a layout that fully covers the window, without altering the structure of the DOM tree.

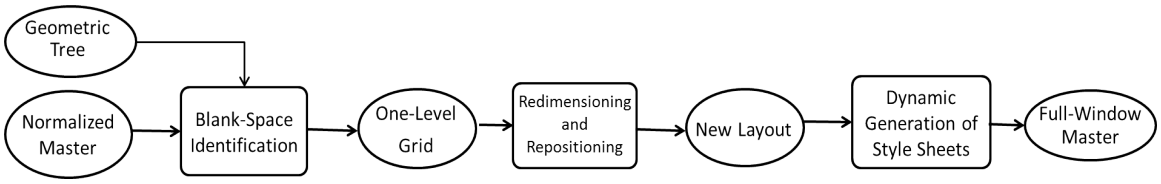


Figure 5-5: Building blocks for the Full-Window design algorithm on the master application

As Figure 5-5 shows, the Full-Window Design algorithm consists of three main phases: the blank-space identification, the block re-dimensioning and re-positioning and the application of the new layout on the application DOM tree. The blank space identification is based on the grid identification. Once the blank spaces are identified, we look for possibilities to occupy them with content in the second step. Finally, the new layout is applied on the application DOM tree.

5.3.2 Blank-Space Identification based solely on geometrical features

To identify the blank spaces, the algorithm described in Section 5.2.2 is applied on the set of the master blocks and as result a grid is obtained.

Among the grid cells, those that are empty correspond to the blank spaces that need to be eliminated.

We note here that the grid identification is pure geometrical in contrast to the grid identification for RWD as we are going to see in Section 5.4.2.

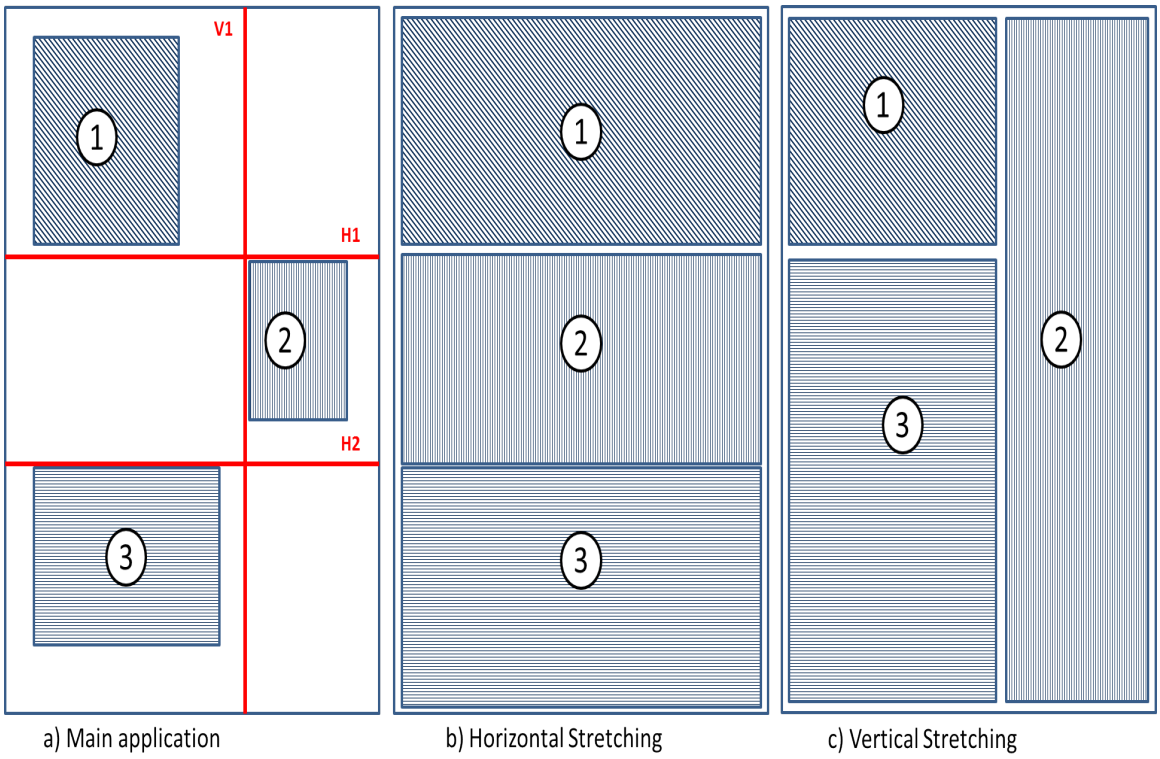


Figure 5-6: Full-Window Design: horizontal vs vertical stretching

5.3.3 Block re-dimensioning and re-positioning

This phase takes as an input the identified grid and exploits the empty cells to re-position and re-dimension the blocks by stretching them to occupy the whole space on the browser window.

To follow the reading order of the blocks, we start first checking the possibility for a horizontal stretching mainly from left to right, and then we check the possibility of

a vertical stretching mainly from top to bottom. We took this decision based on the block matrix for web applications as defined by Song et al. [50]. Song et al. state that the regions containing information in a web page do not have equal importance. Usually the most important blocks reside in the most visible part of the page. Thus the block importance is correlated to the spatial features of a web page. Song et al. conclude that a highest importance is assigned to blocks found in the middle of a web page, and a lower importance to blocks outside this area. During the stretching we want that the blocks at the middle of the page stay in the middle and those in the bottom (to the top resp.) stay at the bottom (to the top resp.) of the page.

Stretching rules may be broken in some case. Exceptions may happen for the last block in a column (e.g., block2 in Figure 5-6(c) where the vertical stretching happened to the bottom in addition to the top) and for the first block in a row (e.g., block1 and block3 in Figure 5-6(b) where the horizontal stretching happened to the right instead of to the left).

A stretching is only possible if the cell that is next to the concerned block is an empty cell. If it is the case, we update the grid occupancy and we continue checking the stretching possibility of the same block in the updated grid. Note that if a block spans multiple rows (columns resp.), it is possible to extend it only if the above condition applies for each of its rows (columns resp.).

Exploring first the vertical possibilities of stretching the blocks is also possible but the final result can be very different without necessary breaking the reading order as illustrated in Figure 5-6 where three blocks are present with their reading order. The horizontal stretching led to a one-column layout. The vertical stretching led to a pavement layout formed by two columns and where the first column contains two blocks and the second contains only one block.

The algorithm iterates over the blocks as they are positioned in the grid and following the reading order. For each block, it checks the stretching possibility. Once there are no more possibilities, we update the position of each block to let it start at the top left corner of the first cell it occupies and ends at the bottom right corner of the last cell it occupies to remove the padding and margin spaces.

Depending on the position of the empty spaces around a block, the stretching happens as follows.

1. A block is re-positionned to the left and then stretched to the right if the blank space is to the left.

2. It is stretched to the right if the blank space is to the right.

3. It is re-positionned to the top and then stretched to the bottom if the blank space is to the top.

4. It is simply stretched to the bottom if the blank space is to the bottom.

Additional constraints can be added here 1) especially to respect the aspect ratios of each block, i.e., the ratio of its width to its height, 2) to respect the relative sizes between the different blocks, or 3) to consider the block position and the importance matrix to decide on the stretching direction. These two constraints are left for future work but they can be easily integrated.

The block new dimensions are then computed based on the cell dimensions and positions. These new values are used to create CSS styling rules and they are applied on the DOM element associated to each block.

5.3.4 Dynamic generation of style sheets

The intent of this process is to create CSS rules for re-designing the master component.

As mentioned earlier, the Full-Window Design should produce a layout that overlays the master application layout.

The reason is that the master application is a modified version of the single-screen application where parts of it are hidden using the CSS style property-value pair 'visibility: hidden'. The effect of this property on the master application is the creation of large blank spaces that stand between blocks.

Another property-value pair that could have been used for hiding HTML elements is the 'display: none'. In this case, the blank spaces are eliminated since the 'display: none' works by setting the element height to zero. The issue with 'display:none' is that the corresponding element is ignored during the construction of the rendering tree [2] in the browser. This can cause a dysfunctioning as it is the case with the Google maps that check the height of the map container before loading the map images. For this reason, we replaced it with 'visibility: hidden'.

To overlap blocks on top of an application, a set of style rules are applied to their corresponding DOM elements and to their antecedents.

[2] Render Tree Construction, `http://www.html5rocks.com/en/tutorials/internals/howbrowserswork/#Render_tree_construction`

Listing 5.2: Simplified Full-window CSS styling rules

```
/* Global CSS style rule for antecedents of FWD element */
.parent {
  z-index: auto;/*To not establish a new local stacking context*/
  position: static;
  opacity: 1;
  }
/* Global CSS style rule for all the FWD elements */
.fullWindow {
  z-index: 2147483647;
  position: absolute; /*z-index only for positioned elements*/
  margin: 0px !important;/* to eliminate margins */
  flex: 1 1 0% !important;
  object-fit: contain;/*To make the element updates its dimension
      while respecting the aspect ratio*/
  background-color: black;
  }
/* An example of a dynamically created FWD CSS rule for an
   element with 'blockID' id*/
#blockID {
  width: 50%;
  height: 50%;
  left: 0%;
  top: 0%;
  }
```

Listing 5.2 presents three CSS style rules.

The first rule is global and applied to all antecedents of the DOM elements in the Full-Window Design. It ensures that the parent elements do not create a new stacking context. This aims at making the concerned block(e.g., the blockID element) belongs to the stacking context of the page root, i.e., the window element.

The second rule is also global and applied to only the DOM elements concerned in the Full-Window Design. The main thing to note here is the presence of 'z-index' property that determines the stack order of a DOM element. An element with greater stack order is always in front of an element with a lower stack order. With a value set to infinity, e.g., 2147483647, we let the full-window elements overlap the other non-relevant elements. 'z-index' only works for positioned elements, for this reason the element positions are set to absolute. Another main property is the 'flex' [3] that specifies the ability of an item to alter its dimensions to fill available spaces. Flex items

[3] See `https://developer.mozilla.org/en/docs/Web/CSS/flex`

can be stretched to use the available space proportionally to their flex grow factor or their flex shrink factor to prevent overflow as defined in the W3C standards[4]. The values attributed to flex means that the concerned element will have the same width as its siblings. The 0% value refer to the flex basis and it means that initially the concerned element occupies 0% of the available space.

Object-fit [5] defines how an element responds to the height and width of its content box. The assigned value, i.e., contain, means that the concerned element will adjust its height and width while respecting its aspect ratio. As a result, if the element new dimensions do not fill the container width, the remaining space is filled with a black background as defined in line 14.

The third rule is personalized for all the DOM element concerned in the Full-Window Design algorithm. It consists in assigning values for the width, height, top and left properties. These values are percentages and calculated relative to the ultimate container, i.e., the window. In addition, They correspond to the calculated dimensions and position in Section 5.3.3. The selector we have chosen is the 'id' represented by the dash sign.

Once we finish creating the CSS rules, we inject them dynamically into the master application in the form of inline style sheets inside a 'style' tag.

5.3.5 Full-Window Design Evaluation

To evaluate the Full-Window Design algorithm, we test the presence of blank space after applying the algorithm on the master application.

We start identifying some metrics to quantify the amount of blank spaces on an application, and then we apply these metrics on the set of application before and after applying the FWD algorithm.

5.3.5.1 Quantifying the blank space problem and the resulting metrics

Tarasewich [52] defines the percentage of blank space as the percentage of a page not taken up by graphics and text. This definition is slightly different from our definition. We recall that the blank spaces correspond to the blocks of the slave device that are made hidden and to the margins and borders caused by CSS. We

[4] See `https://developer.mozilla.org/en/docs/Web/CSS/flex`

[5] Object-fit, `https://css-tricks.com/almanac/properties/o/object-fit/`

express this definition by the difference between the window area and the master area, as Equation 5.1 shows.

$$Percentage_{BlankSpaces} = \frac{Area_{Window} - Area_{MasterBlocks}}{Area(Window)}; \quad (5.1)$$

Tarasewich [52] states that small percentages of blank space can lead to less scrolling and a more compact visualization, but a higher percentage may lead to more readable pages. In the context of our work, and specifically with the Full-Window Design algorithm, we are looking to minimize the percentage of blank space.

This will not affect the readability of the master component since the amount of content we are dealing with is relatively small, i.e., between 1 and 3 objects as shown in Table 5.1.

5.3.5.2 Applying these metrics on our dataset

We tested the FWD algorithm on our dataset as shown in Table 5.1. As a large device, we considered a PC with a window of 1920*1080, equivalent to the screen of TV HD.

We identified for each master application the number of geometric elements it contains. In 72% of the cases, the master has only one geometric block. In 17% of the cases, the master has two geometric blocks. In 11% of the cases, the master has three geometric blocks. This shows that the master component has a simple user interface with a reduced number of blocks.

Before applying the FWD algorithm, we calculated the percentage of blank spaces on the master applications relative to the window dimensions following Equation 5.1.

Table 5.1 shows that on average 74% of the space on the master application produced by our system is a blank space. This high value of blank spaces on the master application is a quantitative explanation for the necessity of eliminating these non-exploited blank spaces on the browser window.

After applying the FWD algorithm, we calculated the percentage of the device window that is occupied with the content of the master application. In 95% of the tested applications, the updated master component was covering more than 98% of the device window.

In Table 5.1, we reported also the percentages of the horizontal and vertical

Table 5.1: Full-Window Design results on 1920*1080 window

Applications	Geometric objects	Blank-space before FWD%	Blank-space after FWD%	Vertical Stretch%	Horizontal Stretch%
Viewster	1	71	0	100	85
Vimeopro	1	86	0	170	163
Vimeo	1	75	0	104	100
Youtube	1	56	0	52	50
Dailymotion	3	44	1	1719	646
Yahooscreen	1	88	0	2125	2030
Twitch	1	40	0	213	213
Liveleak	3	74	1	655	369
Ustream	1	89	2	203	203
Break	2	80	0	222	203
Metacafe	1	87	0	222	203
Jplayer demo1	1	89	0	203	203
Jplayer demo2	2	88	0	203	203
JWPlayer demo1	1	86	0	163	170
JWPlayer demo2	1	92	10	244	245
JWPlayer demo3	1	91	0	244	233
MediaElements	2	10	0	51	203
Video Players (average)	1.3	76	0	185	210
Video-semantic	1	93	1	270	257
Average	1.4	74	2	343	398

stretching that took place on the blocks. We define respectively the horizontal and vertical stretching as follows in Equation 5.2 where Dim refers to the width or to the height of a geometric block.

$$Percentage_{DimStretching} = \frac{oldDim - newDim}{oldDim}; \tag{5.2}$$

It is notable that for 70% of the tested applications, the difference between the vertical stretching and horizontal stretching is less than 15%. Thus the aspect ratio of the master content is maintained specifically for the video element. For dailymotion and liveleak, the vertical stretching was respectively double and triple the horizontal stretching of the geometric blocks. The FWD is open for refinement to consider the

blocks aspect ratios and the blocks relative dimensions for stretching.

In addition, we conducted unit tests to further validate our FWD algorithm on applications with up to 10 geometric blocks especially because with our dataset, we had at maximum three geometric blocks. Figure 5-7(a) illustrates an example of our unit test pages. It consists of 9 images with red borders positioned within div elements. These div elements are randomly positioned on the page. The result of the FWD algorithm is shown in Figure 5-7(b). Results show that all empty spaces are eliminated and the content occupies all the window space.

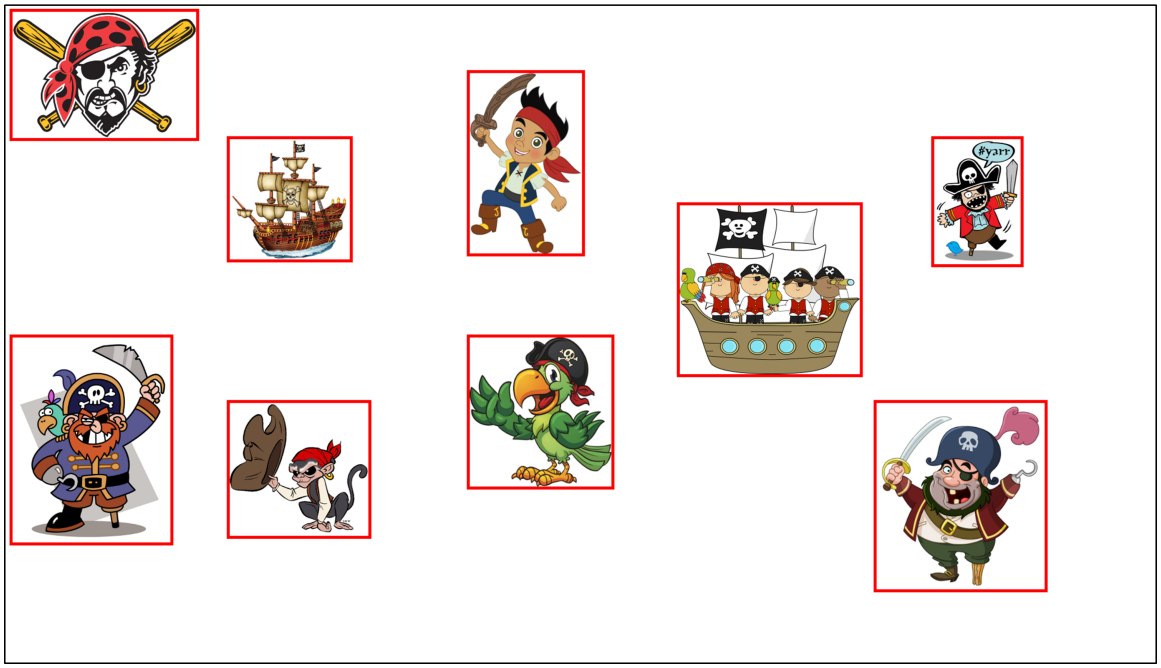
5.4 Slave Adaptation: Responsive Web Layout Re-Design

5.4.1 The responsive web design as the solution to our problem

The objective of this section is to adapt the slave layout to the screen width of the devices on which the slave will run. By re-designing the layout, we do not target a specific device but instead we plan at designing a layout that dynamically identifies the device screen width, and dynamically selects the corresponding layout. This is useful in the multi-screen environment where applications can move seamlessly between various devices, even after the split.

The applications we consider here are dedicated for the desktop. The challenge is to produce a layout that:

1. is capable of reproducing a layout close to the original layout on large devices because the slave can also run on large devices as explained in Section 3.3,
2. avoids horizontal scrolling on small devices,
3. respects the reading order,
4. respects the relative sizing between all blocks on large devices,
5. does not modify the DOM structure, and finally
6. detects dynamically the changes in the window size and adapts to them.



(a)

(b)

Figure 5-7: (a) Example of a unit test page for testing the FWD algorithm with 9 blocks (b) Results of FWD on the unit test page

One obvious solution to avoid the horizontal scrolling on small devices is to resize all the blocks individually to fit into the device window, or to zoom out the complete application. By doing so, we satisfy most of the above requirements but this results in illegible text, image, video, etc. In addition, this makes the user interaction more complex especially on a touch screen device.

The responsive web design (RWD) introduced in Chapter 2 is the response to our objectives and to our requirements (1), (2), (5) and (6) since it uses media queries to attribute dynamically a different layout to each range of device display width and it designs the layout in a grid-like system that is flexible. For each device, a specific grid is designed in advance without adding additional CSS rules and without changing the DOM structure.

Among the most famous RWD frameworks that are based on a grid-like system is Twitter Bootstrap [53]. Using Bootstrap, a designer determines a grid where he places the application content and sets for each cell its width and position. Afterwards, the width and position are attributed to the corresponding DOM element in the form of a set of HTML classes.

In Bootstrap, the width and the position of a DOM element is computed relative to a container element while respecting the relative sizing between the parent and the children(4). Indeed, the width of a DOM element is computed relative to the width of its parent element. Thus, once the parent element changes its dimensions, all its children will follow equally this change.

To re-design the application layout, our approach imitates the RWD designer work and starts by identifying a bootstrap-like grid, then it configures the grid with new position and dimension values. Finally, these new values are attributed to the DOM nodes following the Bootstrap model.

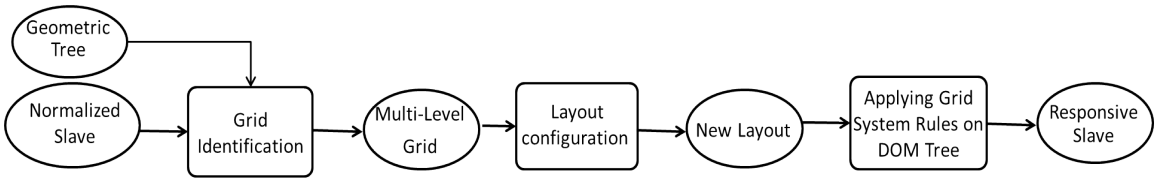


Figure 5-8: Building blocks for making the responsive re-design on the Slave

These building blocks are represented in Figure 5-8.

As indicated earlier, it takes the logical and geometrical trees of the slave component as an input. It delivers a responsive design configured with two layouts using Bootstrap. A one-column-grid layout is produced for small devices with a screen size

smaller than or equal to 768 px. For large devices, the number of rows and columns of the grid is not fixed in advance and the grid is deduced from the fixed-layout of the initial slave. The aim is to reproduce this layout on large devices that have a screen width greater than 768 px.

5.4.2 Identification of the spatial distribution while respecting the DOM structure

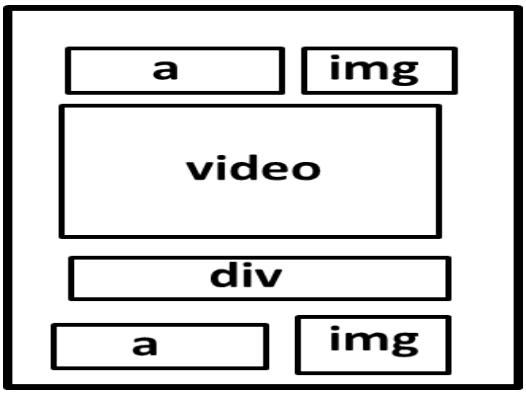
The grid identification problem consists in finding a multi-level grid that respects the Bootstrap grid-model. The grid levels should correspond to the geometric tree levels that itself corresponds to the DOM tree. In another word, for each level in the geometrical tree a grid is identified. Figure 5-9(a) represents the layout corresponding to the DOM tree of Listing 4.1, and Figure 5-9(b) shows its corresponding geometric tree. This geometric tree contains two levels: the first consists of three geometrical objects (i.e., G21, G22 and G23) and the second consists of 6 leaves. Similarly, the associated grid should have two levels.

At any level, a grid cell can contain at max one geometrical object from the corresponding level. But indeed, it can contain other geometrical objects from a deeper level of hierarchy.

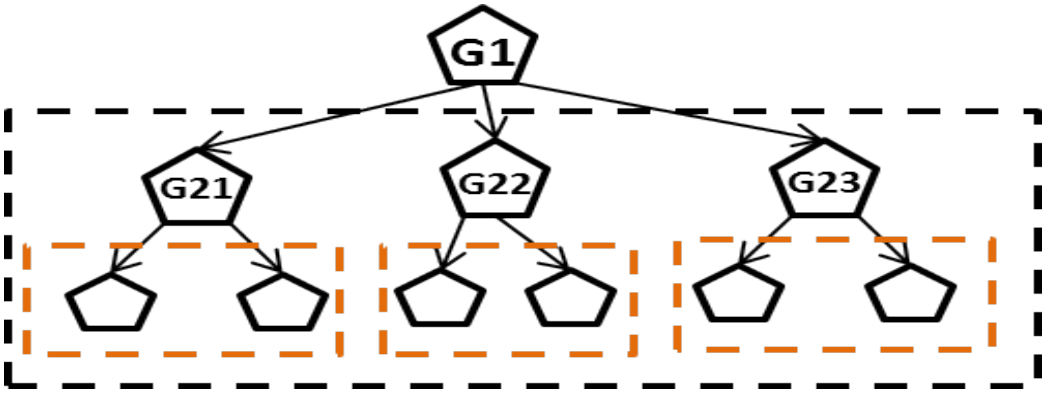
The algorithm starts with the root of the geometric tree. If the root has children, then it is considered as the container of a first-level grid and its direct children are the elements of this grid. The grid container corresponds to the geometrical object that groups all the blocks of a grid. The geometry of these children is then processed to characterize the first grid. We first identify the horizontal and vertical separators by checking the children alignment, similar to Section 5.2.2.

Then, we identify the cell to which each child belongs. Note that not all the cells are occupied by a geometrical object and there might be some empty cells. In our example, the first-level consists of only one grid (Grid1) that contains three cells (G21, G22 and G23) separated horizontally. G1 is considered as the container of the top level grid.

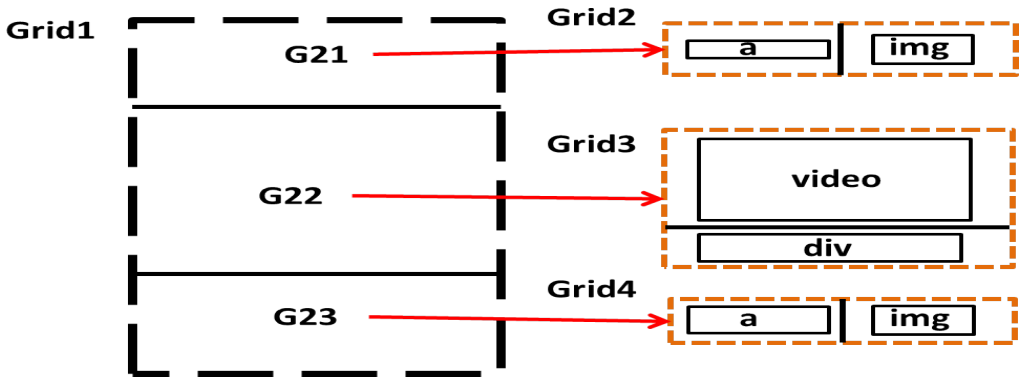
The algorithm iterates over the children sub-trees in a depth-first manner until it reaches the leaves of the geometric tree to form a N-level grid where N is the number of levels of the geometric tree. If a child has descendants, we identify its second-level sub-grid. For instance, G21 (resp. G22, G23) has two descendants in the geometric tree, thus it forms a second-level grid (resp. Grid2, Grid3, Grid4) of two cells. G21



(a)



(b)



(c)

Figure 5-9: (a) Initial application Layout (b) Associated Geometric Tree (b) Grid and separator identifications

(resp. G22, G23) is considered the container of this second-level grid.

We note that the first index in G21 (resp. G22, G23) refers to the level of the grid, i.e., level 2. The second index is specific for each grid in the same level.

5.4.3 RWD layout configuration

5.4.3.1 General Layout configuration

As previously mentioned, we want a layout with two configurations: a one-column grid for small devices and a multi-column grid that reproduces the initial slave layout for large devices. The main requirements here are to respect the aspect ratio of each block and to respect its relative dimension and position compared to other blocks. In addition, we should maintain the blocks reading order in both configurations.

Similar to Bootstrap, during the design phase we abstract from the real value of the screen width since we do not target a specific device but mostly a range of device screen sizes.

We assume that each grid at any level of hierarchy is formed with one row and its total width is decomposed into M effective columns of equal width.

This abstraction phase to which we refer as normalization, is at the base of the grid flexibility in RWD. Its main interest for us is to express the block width and block position relative to the dimensions of the grid container no matter what is its width or its position. The problem of configuring the layout is thus simplified to mapping the grids identified in the previous Section 5.4.2 to the normalized grid that is contains M effective columns.

The algorithm takes as an input the hierarchical grid. It identifies for each grid cell the effective number of columns that span its width and its position as explained in the following paragraph. Note that we do not take into consideration the object heights since our objective is to avoid only the horizontal scrolling.

5.4.3.2 Layout configuration for large devices

The algorithm starts processing the first-level grid and then iterates over the sub-grids in a depth-first manner. Note that a sub-grid refers to an L-level grid where L is an integer referring to the level of hierarchy.

A crucial step here is to determine for each grid or sub-grid the total width that its blocks can span, we call it the reference width. The reference width for the first-level grid is the window width, while the reference width for an L-level grid is the width of its container.

For each L-level grid, the algorithm iterates over the grid cells that contain blocks and calculates for each of them, in terms of number of effective columns: 1) its width relative to the reference width, 2) its left-offset relative to the distance that separates it from the block to its left.

The calculated width and left-offset are immediately applied using CSS styles, on the DOM elements that correspond to each block. This is crucial before moving to the (L+1) level since the width and the position of the associated grid (i.e., (L+1) grid) might have been changed by the layout configuration of the L-level grid. By re-evaluating their values (in pixels), we ensure that if the width and the position were not exact, the error is not propagated to the descendants. Thus the descendants will always have the same width relative to their parent element.

The algorithm passes to the (L+1) level grid and continues until it reaches the final N-level grid.

5.4.3.3 Layout configuration for small devices

The same logic is applied to configure the single-column layout for small devices. The only difference is that we consider that all the blocks of a grid have a width equal to their container, i.e., M effective columns.

In addition, since the screen size of small devices is very small, we eliminate the left offsets that exist between the blocks to eliminate blank spaces.

5.4.4 Applying grid system rules on the DOM tree

For the implementation, we reuse some of the CSS classes defined in Bootstrap [53], among which the '.container-fluid' applied to the root grid of a document for proper alignment and padding for the whole document, and the '.col-xx-yy' that determines the content width and position. The 'yy' is a value between 1 and 12 and it denotes the number of columns that the element spans. The 'xx' is a string that refers to one of the four possible breakpoints provided by Bootstrap:

1. The 'extra-small' breakpoint, represented by 'xs', corresponds to devices with a maximal width of 480px, i.e., phones.
2. The 'small' breakpoint, represented by 'sm', corresponds to devices with a maximal width of 768px, i.e., tablets.
3. The 'medium' breakpoint, represented by 'md', corresponds to devices with a maximal width of 992px, i.e., desktops.
4. The 'large' breakpoint, represented by 'lg', corresponds to devices with a maximal width of 1200px, i.e., wide screens of TVs.

The effective columns, described in Section 5.4.3.2, correspond to the Bootstrap columns. To calculate the number of Bootstrap columns for each geometrical object, we use simply the following formula:

$$yy = \left\lfloor \frac{Width(geom.object)}{Width(container)} * 12 \right\rfloor; \tag{5.3}$$

A similar formula is applied to calculate the number of columns representing the offsets between blocks:

$$Offset_{yy} = \left\lfloor \frac{Offset_{Left}(geom.object)}{Width(container)} * 12 \right\rfloor; \tag{5.4}$$

Now that we calculated the Bootstrap width and offset values, we can proceed to attribute them to the '.col-xx' and '.col-xx-offset-yy' classes. For extra small devices, we use the single columns class, i.e., '.col-xs-12', and for devices with a screen size greater than 768 px, the '.col-sm-' and '.col-sm-offset-' to which we concatenate respectively the calculated width and offset.

5.4.5 Evaluation of the RWD algorithm

5.4.5.1 Quantifying the horizontal scrolling problem and the resulting metrics

The problems we try to solve with our RWD layout re-factoring approach essentially relate to the horizontal scrolling.

A set of metrics are found in the literature that quantifies this aspect. Nebeling et al. [39] identifies some metrics related to the scrolling in the context of evaluating a layout on large devices. He distinguishes between:

1. the amount of scrolling needed to see the whole page, defined in Equation 5.5

$$PageWindow\ Ratio = \frac{Area_{Page}}{Area_{Window}}; \tag{5.5}$$

2. the amount of scrolling needed to see only the relevant content of a page, defined in Equation 5.6; where the content area is the sum of the foreground content areas; and the window area refers the room available to display the content without scrolling.

$$ContentWindow\ Ratio = \frac{Area_{Content}}{Area_{Window}}; \tag{5.6}$$

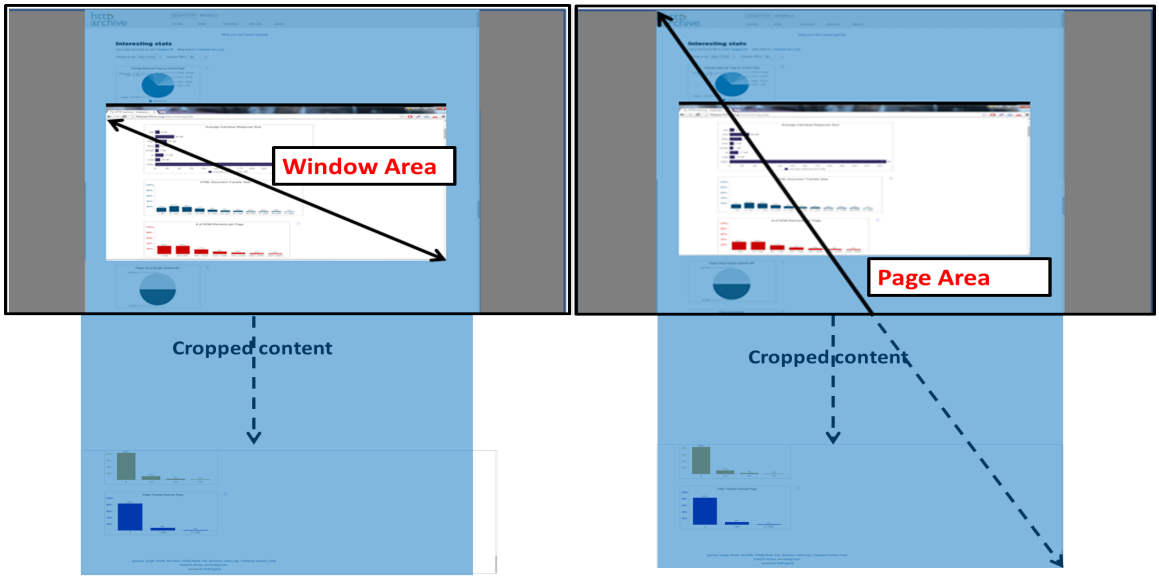


Figure 5-10: Window Area versus Page Area

The difference between the window area and the page area is illustrated in Figure 5-10. Note that only part of the page area was illustrated since the page is very big with a height equal to 12 761 px.

Using the above two metrics, we quantify the required scrolling without making a difference between the horizontal and the vertical scrolling. In our work, we are concerned about the horizontal scrolling especially because the vertical scrolling cannot be avoided on a small device.

In our work, we consider that a page causes horizontal scrolling if:

- at least one of its blocks is partially positioned outside the window box, or

- at least one of its blocks is completely outside the window box.

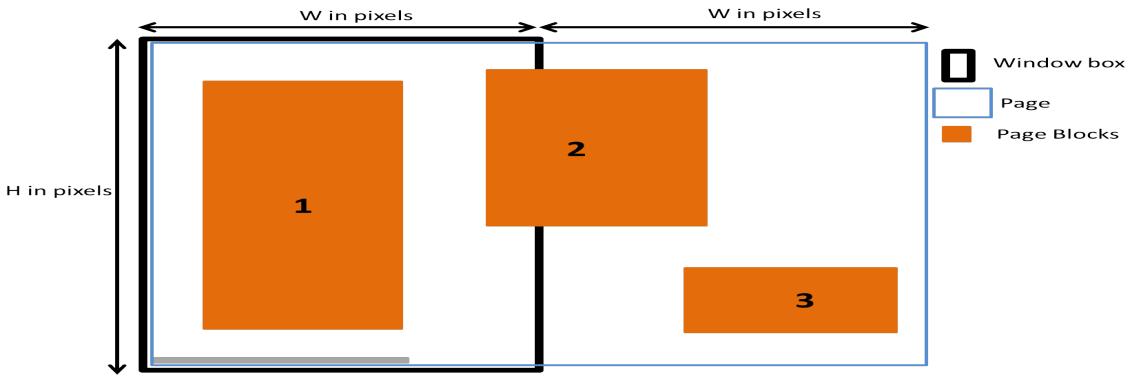


Figure 5-11: Example of a page with horizontal scrolling

Figure 5-11 is an example of a page causing horizontal scrolling. The black border illustrates the window borders, and the blue border represents the page borders. The page consists of three blocks (i.e., block1, block2 and block3), represented with orange rectangles. Block 2 is partially outside the window box and block 3 is completely outside the window box. The page has a height equal to the window height, but it has double the width W of the window. In consequence, in order to see the part of the page that is outside the window box, it is enough to scroll a distance equal to W.

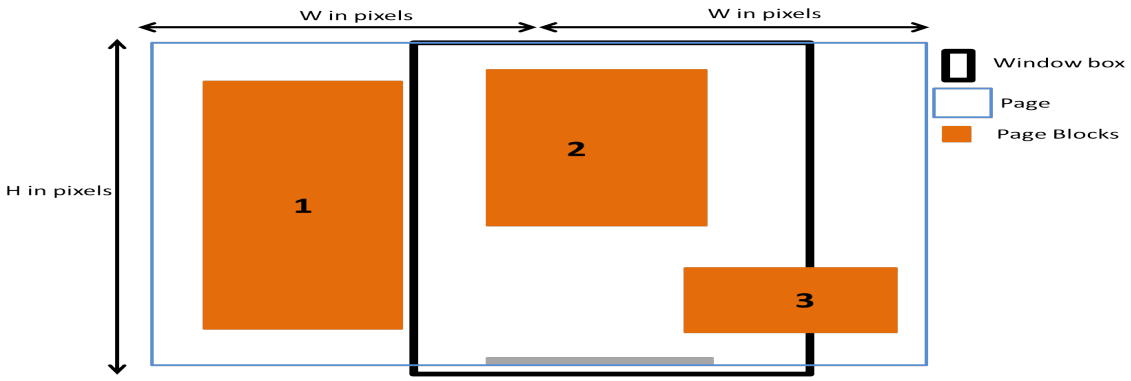


Figure 5-12: Doing horizontal scrolling to see block 2

In contrast, to see the totality of block 2 we only need to scroll a distance smaller than W, as shown in Figure 5-12. But, to see completely block 3 we only need to scroll a distance equal to W, as shows Figure 5-13.

Based on these observations, measuring the horizontal scrolling at the level of the content blocks is more relevant than measuring it at the level of the page. Using CSS, a page (or a block) can have a width that exceeds the maximal abscissa of its

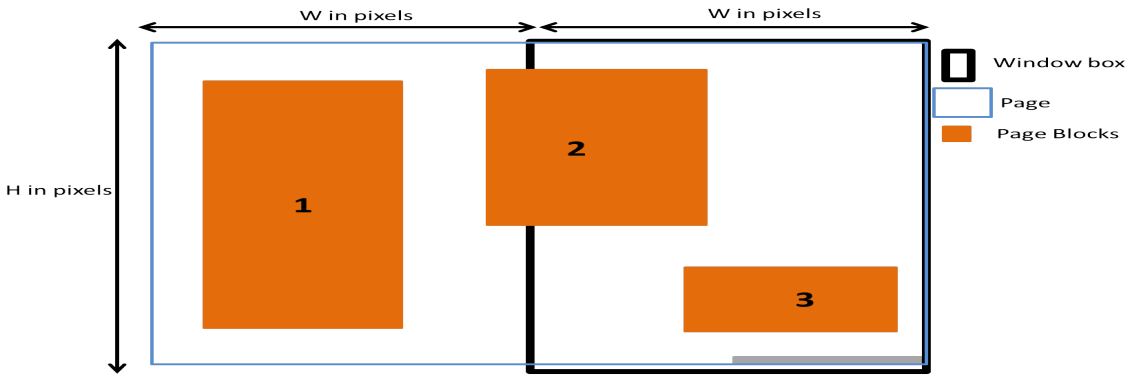


Figure 5-13: Doing horizontal scrolling to see block 3

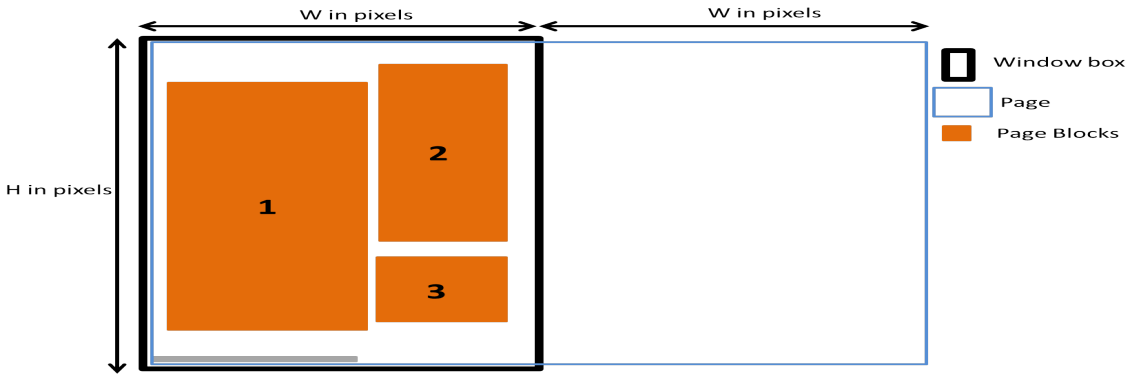


Figure 5-14: False positive for horizontal scrolling

descendants and as a result it causes a horizontal scrolling. In Figure 5-14, we have three boxes that all belong to the window box, but the page still causes horizontal scrolling since the parent element has a width greater than the ideal width to wrap horizontally its descendants. The horizontal scrolling in this case can be considered as false positive since there is no content outside the window box. The false positive detection can be avoided by testing the presence of horizontal scrolling at the level of content blocks rather than the page.

We define the amount of horizontal scrolling for a specific block (x, y, w, h) as the distance between the right border of the window box (W, H), i.e., W, and the right border of the corresponding block, i.e., x + w. Equation 5.7 illustrates this definition.

$$dist(block, window) = x \ + \ w \ - \ W \tag{5.7}$$

Based on the distance equation 5.7, we define the amount of horizontal scrolling for a page in Equation 5.8 as the average amount of horizontal scrolling for all the

blocks causing the horizontal scrolling.

$$Amount\ HS_{page} = \frac{\sum dist(block_{HS}, window)}{Nb.\ blocks_{HS}}; \tag{5.8}$$

5.4.5.2 Setup and Results

For the RWD algorithm, the evaluation of the responsive layout consists in testing the presence and the amount of horizontal scrolling on small devices for slave components.

Tests consist in traversing the geometric tree, checking for each node in the geometric tree its position relative to the window box. If the block is outside the box, then we calculate the distance separating them following Equation 5.7. After checking all the nodes, we calculate the average amount of horizontal following Equation 5.8.

We conducted these tests on 6 non-responsive applications from our dataset. We exclude the demos from the video libraries because they have limited content.

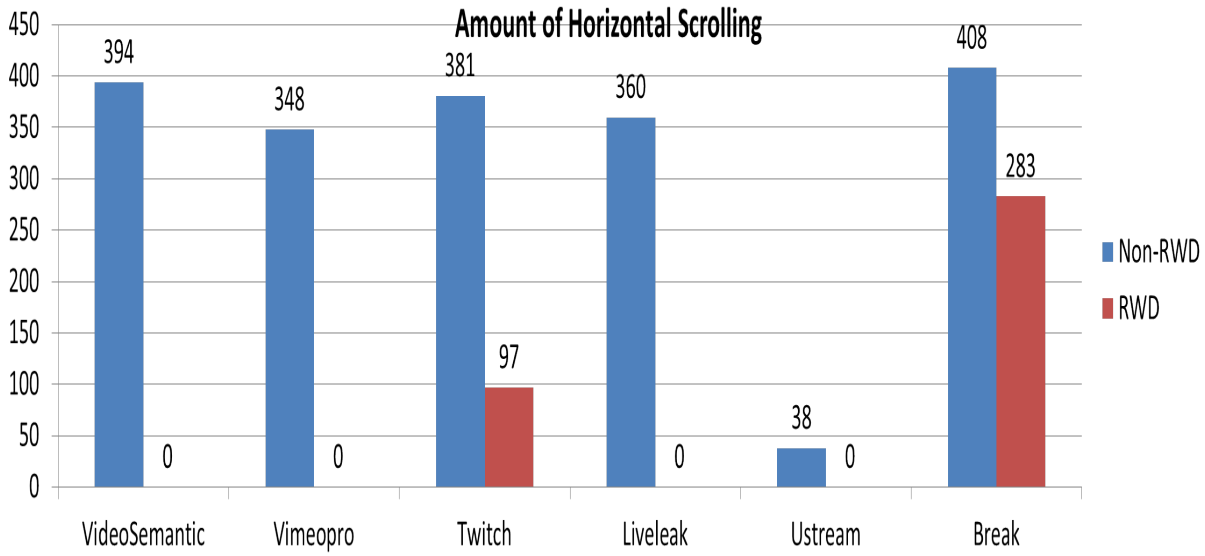


Figure 5-15: Amount of horizontal scrolling for each tested application

Figure 5-15 shows the amount of horizontal scrolling calculated for each application before and after applying our algorithm. This Figure shows that, before applying our algorithm, all slave applications cause horizontal scrolling on the small device. The minimal amount of horizontal scrolling before applying our algorithm is 38 px and it corresponds to the non-RWD Ustream that has 14 blocks responsible for the scrolling. The amount of horizontal scrolling for the remaining non-RWD applications varies between 348 px and 408 px. Relatively to the width of our small device (i.e.,

412px), these values are big.

After applying our algorithm, the horizontal scrolling was completely eliminated for VideoSemantic, VimeoPro, Liveleak and Ustream. In contrast, the amount of horizontal scrolling was reduced from 381 px to 97 px (i.e., 75% of reduction) for Twitch and from 408 px to 283 px for Break (i.e., 30% of reduction). This enhancement is

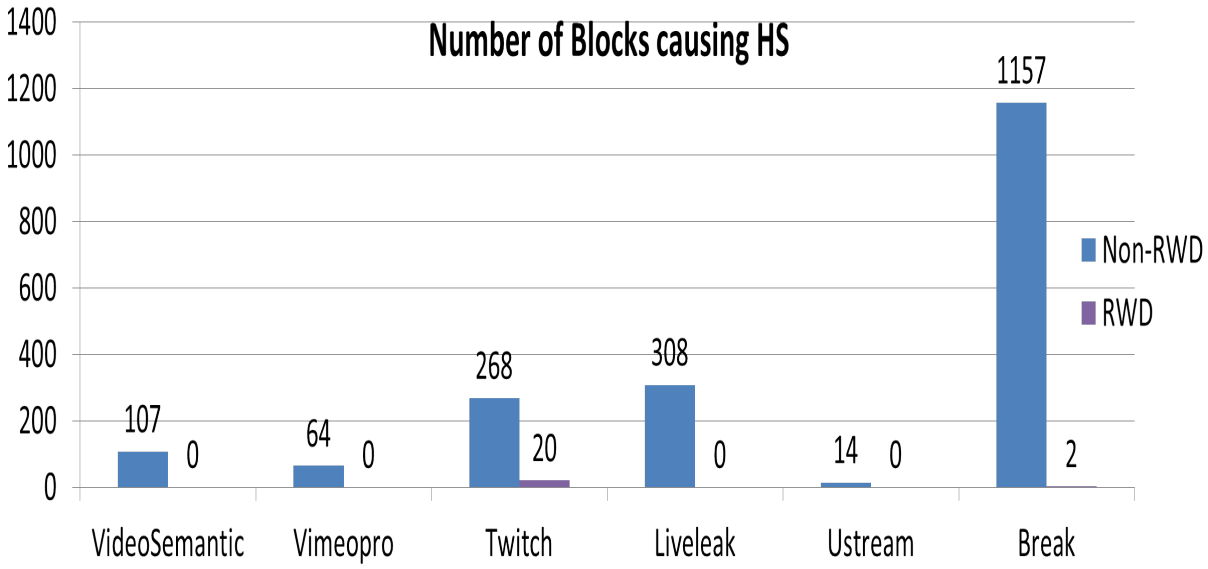
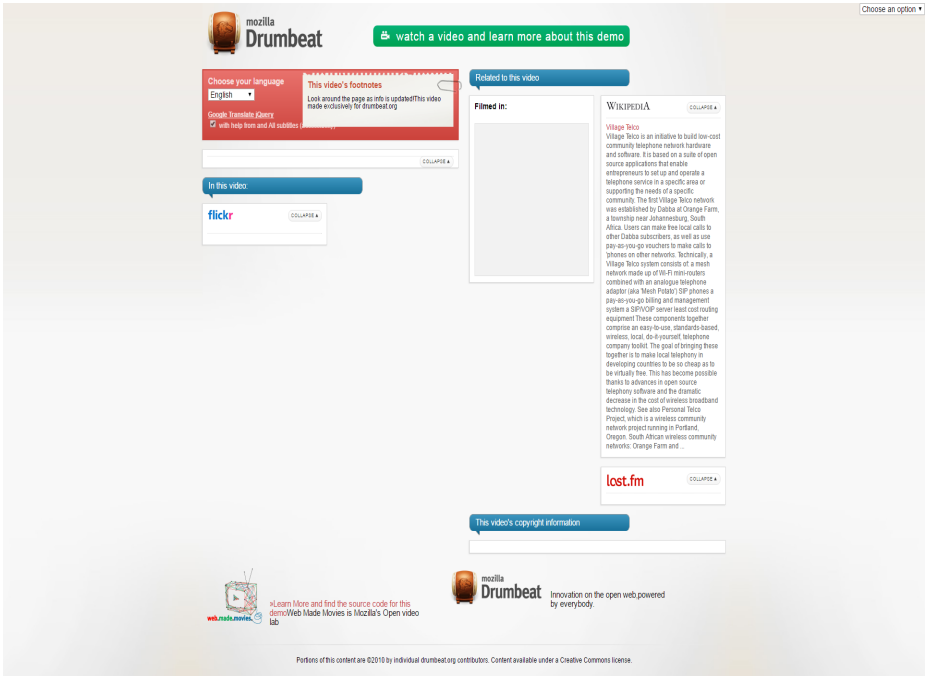


Figure 5-16: Number of blocks causing horizontal scrolling for each tested application

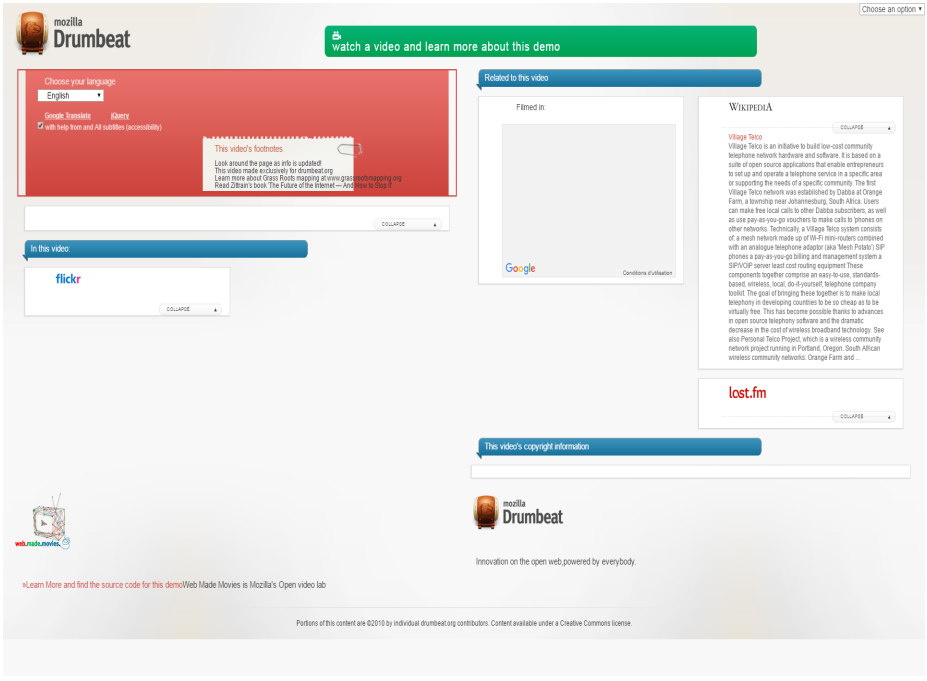
detailed in Figure 5-16 that shows for every application the number of blocks causing the horizontal scrolling before and after applying our algorithm. In this Figure, we can see that our algorithm eliminated the horizontal scrolling from 92% (i.e., (268-20)*100/268) of the Twitch blocks, and it eliminated the horizontal scrolling from 99% of the Break blocks.

Figures 5-17 and 5-18 illustrate the results obtained by applying the RWD algorithm on the slave component of the video semantic application on the small and large devices. On large devices and by comparing Figures 5-17(a) and 5-17(b), we can see that the RWD reproduced a layout similar to the initial.

On the small device, we represent the window views before and after applying the RWD algorithm in Figures 5-18(a) and 5-18(b). To better illustrate the totality of the slave RWD layout, we apply twice the vertical scrolling and show the corresponding window views in Figures 5-18(d) and 5-18(c). We can see that all the parts of the reference slave applications are aligned in one column and the horizontal scrolling is not required to see the complete application. In addition, we can clearly remark that after applying our RWD algorithm the content is still readable.



(a)



(b)

Figure 5-17: Window view of the video semantic slave component: (a) Non-RWD on the large device (b) RWD on the large device

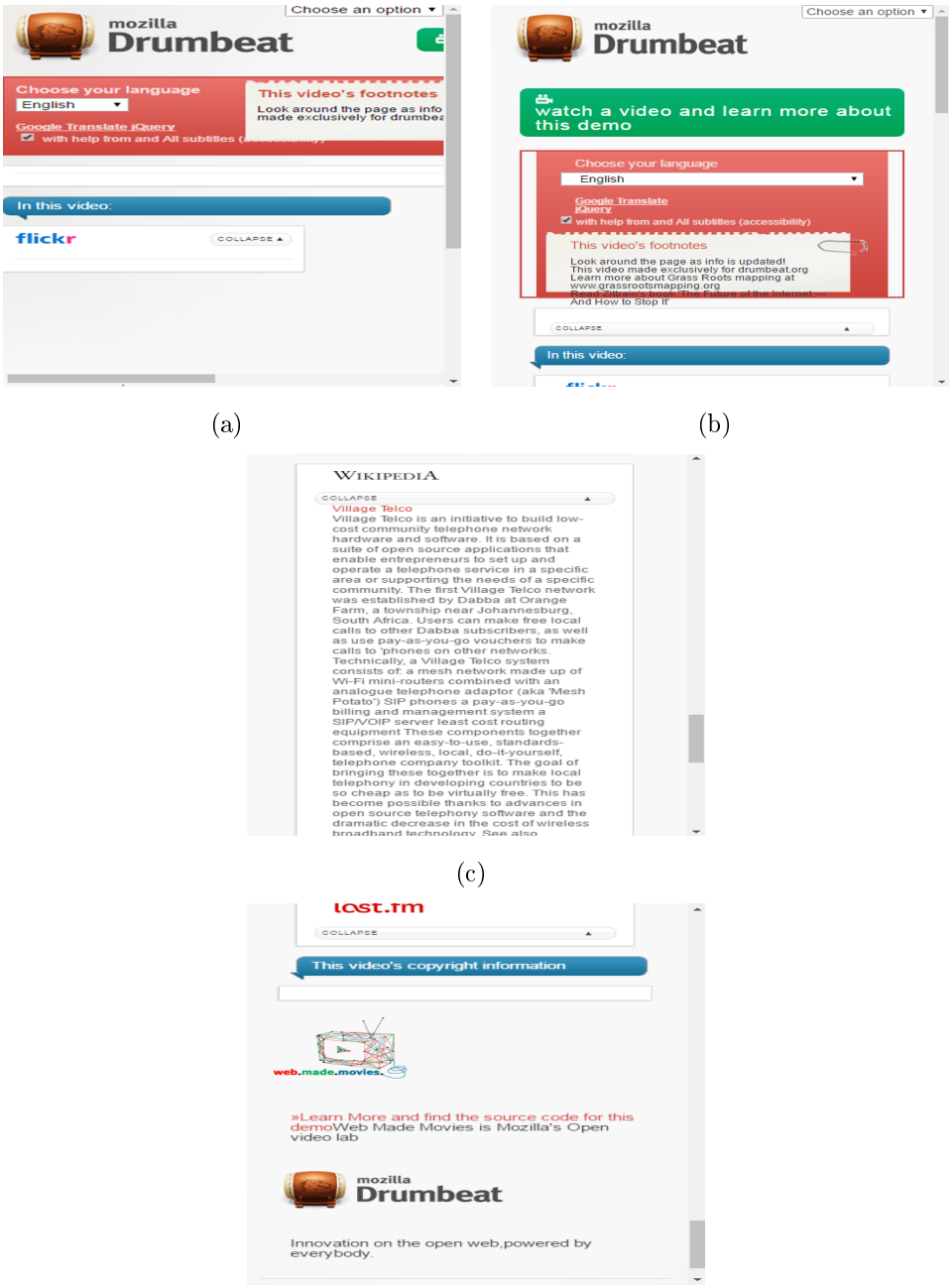


Figure 5-18: Window view of the video semantic slave component: (a) Non-RWD on the small device (b) RWD on the small device (c) RWD on the small device, after one vertical scrolling (d) RWD on the small device, after two vertical scrolling

5.5 Conclusion

Table 5.2: Differences between the two Layout refactoring implementations

Features	Master FWD	Slave RWD
Objective	Adapt the content to the window	Adapt layout to multiple devices
Layout Anomaly	Blank spaces	Horizontal scrolling
Basis	Make content occupy all the window space	Create RWD layouts
Grid	One-level Grid, following blocks geometry	Multi-level Grid, following blocks geometry and the DOM structure
Dimensions and Positions (percentage)	Calculated relative to window	Calculated relative to every grid head (variable)

In this section, we review the master FWD and the slave RWD algorithms. Their differences are summarized in Table 5.2.

While FWD aims at adapting the master content to large devices, RWD aims at adapting the slave layout to multiple devices.

On the one side, FWD resolves the blank space anomaly that is induced by our model for the content distribution. It consists in making the content occupy all the available space by updating their positions and their dimensions. These latter are calculated relative to the window.

On the other side, RWD resolves the horizontal scrolling anomaly that is induced by displaying wide contents on small-screen devices. It consists in re-designing the application layout to produce a responsive web design that identifies and adapts dynamically to the device on which it is running.

Both FWD and RWD are based on the identification of a grid out of their content. But since the new positions and dimensions for FWD are calculated relative to the window size, we abstracted from DOM structure and considered that all the visible contents have one parent, i.e., the window size. Based on the content geometry, we created a one-level grid.

In contrast, the RWD requires not only the respect of the content geometry but as well the respect of the DOM structure for CSS to apply. This lead to the identification of a multi-level grid.

The evaluation of both algorithms, after the quantification of each of the layout anomalies, revealed: (1) the absence of blank spaces on the master components, (2) the absence or the reduction of the horizontal scrolling on the slave components.

As a perspective, FWD can be improved to consider the block aspect ratios and their dimensions relative to each other. The stretching decisions can be taken for instance after studying the importance of a block in the application. As a result, blocks with higher importance should occupy more space than a block with lower importance.

The RWD algorithm can also be improved for instance to not only generate the initial layout on large devices but also to optimize the initial layout to better occupy the available space, and to design automatically an optimized layout for medium devices.

Finally, a user study is mandatory to verify qualitatively the two layouts and to measure the level of end-user satisfaction for the application look and feel.

Chapter 6

Run-time Support and State Distribution

6.1 Introduction

This chapter aims at presenting the work related to transforming the master and the slave components into a multi-screen application.

At the end of Chapter 4, the resulting components are static pages that neither have attached a JavaScript code, nor are interactive, nor can communicate with each other. In contrast, a multi-screen application consists of having two components with a functional user interface, capable of communicating with each other and finally providing a complementary usage experience during runtime.

The required transformations are mainly related to working with JavaScript. They are provided in our system as a part of the UI Distribution phase (number 3) in Figure 8-1. Within this phase, we refer to these transformations as 'logic distribution'. It consists in preparing the two components for the dynamicity of the runtime environment and for the state distribution.

While 'logic distribution' happens before running the multi-screen application, state distribution is a runtime phase. The state distribution is the ultimate objective of this thesis since it makes the application interactive and usable for end-users. Its aim is to preserve the state coherence between the two components without affecting the application overall performance and without implying additional delays. The state distribution is a bidirectional mechanism, i.e., it happens at the level of the two components as we are going to see later in the text.

In Section 3.4.4, we identified two relevant runtime operations that characterize the state distribution: synchronization and redirection. For each of these operations, we describe the challenges they imply on the logic distribution,) our corresponding solutions in Section 6.2 and the techniques used for our solutions in Section 6.3.

In Section 6.4, we evaluate the state distribution phase on one highly dynamic application. In addition, we evaluate the overall performance of the system by calculating the communication delays caused by the network and the processing delays caused by the system. Finally, we conclude about the system usability.

6.2 Logic Distribution

As mentioned in Section 3.4.4, the application logic is kept on the side of the master component that has a slightly modified version of the main DOM tree.

In consequence to our application model described in Section 3.4, Logic Distribution consists in abstracting from the application logic and providing an interface for each of the two components 1) to manage transparently the main logic as introduced in Section 3.4, 2) to track the changes made to the user interfaces of the two components and 3) to ensure the communication between them.

The first two requirements are used to ensure the state synchronization during runtime. Further details are found in Sections 6.2.1 and 6.2.2.

The third requirement is related to the redirection operation and consists in adapting our components to the multi-screen platform, i.e., the COLTRAM platform, and to make them discoverable on the network. Further details are found in Section 6.2.3.

6.2.1 Synchronization

The interaction is the result of user events, mostly connected to scripts, which modify the DOM. If the script is kept on the master component, events occurring on the slave component need to be redirected to that master and any DOM modifications need to be sent to the slave.

Changes to the user interface may happen on the master component when the logic is triggered due to user interactions, or to internal events, i.e., the time of a video. For instance, an image appears at a time t1 of the video, and a text disappears at time t2, etc.

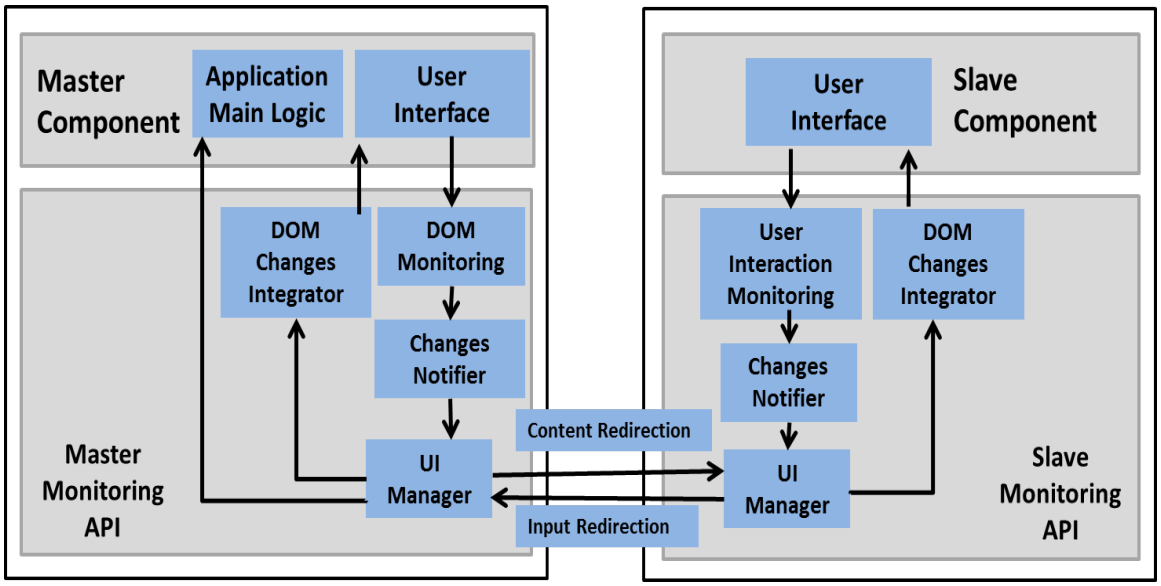


Figure 6-1: Architecture of the master and the slave components including the added monitoring logic

The synchronization challenges illustrated here are 1) to continuously capture these dynamic changes, 2) to characterize (e.g., delete, add or edit) and analyze these changes to check to which component they belong, 3) to redirect them if necessary to the corresponding component, 4) to integrate the changes into this component and finally 5) all this should be done without affecting the application overall performance.

Each of the master and the slave components are extended with additional features to confront with the above challenges.

The four main features as depicted in Figure 6-1 are: the DOM monitoring for the master and its equivalent the User Interface monitoring for the slave, the changes notifier, the UI manager and the DOM changes integrator. These four features make the runtime functioning of the master and the slave components independent from the system especially because they are injected inside each component in the form of a JavaScript API, i.e., Monitoring APIs.

During runtime, the DOM monitoring listens to every change happening on the master DOM tree. Once a change is captured, it is propagated to the changes notifier. The changes notifier analyzes this change and creates a notification that contains information about the type of the change (i.e., addition, deletion or edition of elements, attributes or text), the concerned element and some related information concerning the element position in the DOM tree (e.g., parent element, next and previous siblings, etc.).

The notification is then sent to the UI manager acting as a gateway for the communication between the two components. On the master component, the UI manager checks whether the change concerns the slave. If it is the case, the change notification is redirected to the slave component. We refer to this redirection as 'content redirection' because mainly updates related to the slave content are redirected to the slave. Otherwise if the changes concern the master only, the UI manager skips this change.

Once received by the UI manager of the slave device, the notification is sent directly to the DOM changes integrator. The DOM changes integrator is responsible for integrating all types of changes inside the slave DOM tree.

On the slave component, the User Interaction Monitoring is responsible for listening to the changes related to the user interactions only. Any user interaction is signaled to the changes notifier that creates a notification and then sent to the slave UI manager. The slave UI manager redirects the changes towards the master UI manager. We call it 'input redirection' because mainly user inputs in the form of textual data or requests (i.e., events) are redirected to the master.

Once the master UI manager receives this notification, it determines if the change is at the level of the DOM tree or if the slave is asking to execute a certain function (i.e., event handler) in the application main logic. In the first case, the notification is sent to the DOM changes integrator of the master. In the second case, the master UI manager calls the required logic.

6.2.2 Handling the addition of dynamic elements

Among the changes that can happen on a DOM tree, new elements can be dynamically created and added during runtime in consequence of the execution of the main application logic. In our work, handling the addition of dynamic elements is only related to the master component where the application main logic resides as shown in Figure 6-1. The issue here is that not all the added elements belong to the master user interface especially if the newly added elements are interactive for example. If our objective is to always provide the 'well-fit' content on the best device, thus we have to make the master capable of dividing and distributing on-the-fly the newly created content.

The challenges we face for the content division are the same described in Chapter 4. The only difference is that the on-the-fly division is done by the master and during the application runtime. Thus, we need a simple and non-expensive solution to not

affect the application performance.

Similar to the system UI Division phase, we start checking statically the content tag name, attributes and registered event listeners. If it satisfies one of our 'interactive' or 'multimedia' categories, then the division is straightforward.

Otherwise, we try to divide (or map) it based on its structural relationships and geometrical relationships as described in Section 4.4.2. For instance, if the added DOM element is neither 'interactive' nor 'multimedia' and if it is a descendant of an 'interactive' element, then it should follow its parent.

6.2.3 Integration to the runtime environment

The requirements for this phase are to adapt the master and the slave components to the runtime environment and to the COLTRAM platform in order to prepare them for the redirection operation including the content redirection and the input redirection described in Section 6.2.1 and illustrated in Figure 6-1.

As detailed in Section 2.1, applications running on top of COLTRAM are web services that expose and discover other services. Thus this phase goes back to transforming the slave and master components into web services. On the one side, the master component exposes a 'Content-Mirroring' service that will be discovered by the slave. On the other side, the slave component exposes a 'Input-Mirroring' service that will be discovered by the master.

At the end, the two components can ensure the 'content redirection' and the 'input redirection' of Figure 6-1.

The COLTRAM platform simplifies the tasks for the web service developer. For each component, the developer responsibilities are first to define the name of the service to expose and second to determine the discovery protocol and service name it should discover. Once the master and the slave discover each other, the communication is straightforward in COLTRAM. Then, we need to define the service interfaces and link them to the state distribution components of Figure 6-1.

For each web service, a set of JavaScript documents are added as detailed below.

Listing 6.1: The Master Web Service

```
1 <html>
2     <head>
3         <script src="masterMirroring.js"></script>
```

```
4         <script src="coltramLib.js"></script>
5         <script src="COLTRAM_masterDevice.js"></script>
6         <!-- Application main logic -->
7         <script src="main.js"></script>
8      </head>
9     <body data-coltram = 'master expose'>
10        ---Content-----
11    </body>
12  </html>
```

Listing 6.2: The Slave Web Service

```
1  <html>
2     <head>
3        <script src="slaveMirroring.js"></script>
4        <script src="coltramLib.js"></script>
5        <script src="COLTRAM_slaveDevice.js"></script>
6     </head>
7     <body data-coltram = 'slave expose'>
8          ---Content-----
9     </body>
10 </html>
```

Listings 6.1 and 6.2 are an extract of the HTML documents of the two components. An HTML attribute is added to the body element and its value contains the atom name (e.g., 'master' and 'slave' resp.) and the 'expose' keyword that lets the COLTRAM API (i.e., coltramLib.js) understand that the master and the slave are exposing (or offering) a service.

Listing 6.3: Extract of the COLTRAM_MasterDevice.js document

```
1  var masterProxy = null;
2
3  //the master discovers the slave service
4  Coltram.discoverAny('Input-Mirroring', callback);
5
6  //the master exposes the "Content-Mirroring" service
7  Coltram.serviceImplementation = new Coltram.ServiceImplementation
      ("zeroconf", "Content-Mirroring");
8
9  //.UIManager is the interface of the ``Content-Mirroring" service
       (called when a message arrives)
10 Coltram.serviceImplementation.UIManager = function (msg) {
11   master_domChangesIntegrator(msg); //To process inputs or to
        trigger events
```

```
12 };
13
14 //once the master discovers the slave service, do the binding to
       this service
15 function callback(service) {
16   masterProxy = Coltram.bindService(service.id, Coltram.hostName)
         ;
17 };
```

Listing 6.4: Extract of the COLTRAM_SlaveDevice.js document

```
1  var slaveProxy = null;
2
3  //the slave discovers the master service
4  Coltram.discoverAny('Content_Mirroring', callback);
5
6  //the slave exposes the "Input_Mirroring" service
7  Coltram.serviceImplementation = new Coltram.ServiceImplementation
       ("zeroconf", "Input_Mirroring");
8
9  //.UIManager is the interface to the ``Input-Mirroring" service (
       called when a message arrives)
10 Coltram.serviceImplementation.UIManager = function (msg) {
11   slave_domChangesIntegrator(msg); //To integrate changes to the
         slave DOM tree
12 };
13
14 //once the slave discovers the master service, do the binding to
       this service
15 function callback(service) {
16   slaveProxy = Coltram.bindService(service.id, Coltram.hostName);
17 };
```

The web service interfaces, where the exposed operations of the slave and the master are defined, are added inside a JavaScript document; COLTRAM_masterDevice.js for the master illustrated partially in Listing 6.3 and COLTRAM_slaveDevice.js for the slave illustrated partially in Listing 6.4. In both documents,

- we set the service to discover on line 4
- we determine the service to expose with its name and the service protocol (e.g., zeroconf) on line 7
- once the master and the slave discover each other, a callback on line 15 is

called to establish the communication channel between them and to include the operation exposed by the slave and the master components resp.

- the exposed operations are provided on line 10 of Listings 6.3 and 6.4, i.e., UIManager method for the master and UIManager method for the slave that can be called respectively by the slave and the master once they need to redirect inputs or content. Then, the UIManager of the master (resp. the UIManager of the slave) propagates these information to the master_domChangesIntegrator (resp. slave_domChangesIntegrator).

Listing 6.5: Extract of the masterMirroring.js document

```
1  function masterDOMMonitoring(){
2    var queryDevice = [{
3             element: '*[data-device="device2"]',
4             elementAttributes: att //a list of all attributes set
                 on the slave DOM elements
5           },{
6             characterData: true //to watch changes made on text
                 nodes
7       }];
8    //configure mutation-sumary to watch the slave DOM elements
9    var mutation = new Mutation-Summary ({query: queries, callback:
         redirectContent});
10 }
11
12 function redirectContent(content){
13   //send new DOM content to the slave
14   slaveProxy.UIManager(content);
15 }
16 function master_domChangesIntegrator (msg){
17    var change = {};
18    for(change in msg){
19     //if the user adds her data inputs
20     if(change is 'data_inputs')
21        applyToDOMTree(change);
22     //if the user interaction triggers DOM events
23     if(change is 'DOM_event')
24        callEventHandler(change);
25   }
26 }
```

Listing 6.6: Extract of the slaveMirroring.js document

```
1  function slaveUIMonitoring(){
```

```
2    var change = null;
3    //watch changes and wait for user interactions
4    [extracted code]
5
6    if(change)
7      redirectInputs(change);
8  }
9
10 function redirectInputs(inputs){
11   //send new user inputs to the master
12   masterProxy.UIManager(inputs);
13 }
14
15 function slave_domChangesIntegrator (msg){
16   var change = {};
17   for (change in msg){
18     //access DOM tree and integrate the changes received in msg
19     applyToDOMTree();
20   }
21 }
```

DOM changes Integrator for the slave and for the master are defined respectively in the 'slaveMirroring.js' and the 'masterMirroring.js' documents. An extract of these files are illustrated respectively in Listings 6.5 and 6.6. For each received change, the slave component integrates it in the DOM tree (line 17) and the master component does the same only if the slave sent the input data that the user provided. In the case where the user interaction implied a DOM event (e.g., onclick, onmousover, etc.), the master calls the event handler that corresponds to that event (line 24) and that is defined in the application main logic.

On the master component, a query is set on line 2 of Listing 6.6 to watch all the changes made to elements belonging to the slave components, to their attributes and to their text nodes. Using this query, an instance of the mutation-summary library is created and a callback function, i.e., redirectContent(), is called upon the availability of any change. Once the master DOM monitoring captures a change, it uses the web service interface to call the slave UIManager as shown on line 14 of Listing 6.6.

Once the slave UI monitoring captures a change (line 5), it uses the web service interface to call the master UIManager as shown on line 10 of Listing 6.6.

6.3 Synchronization Implementation

As discussed in Section 6.2, we need to detect relevant changes in the DOM tree on the master and the slave applications and to exchange the updates and user interactions between them.

To this end and for the master application, we use the Mutation-Summary library [56] based on the Mutation Observer API [1]. The key advantage of mutation observers is that they observe nodes for changes. Mutation-Observers are not called for every single change in the DOM tree, but they receive periodic call for a group of changes in the DOM tree.

This behavior is opposite to the Mutation-Events spec [2] that was dropped because it is verbose (i.e., fires too often, for every change), slow (because of event propagation) and they prevent UA run-time optimization and finally it is the source of many crashes for browsers [3].

We configure a Mutation-Summary object to watch changes made to DOM elements with 'device2' annotation, i.e., associated to the slave component, including their attributes, their text content, their descendants. Mutation-Summary creates an internal node map for the watched DOM nodes to which its assigns an internal id to simplify the referencing for these elements. If any change happens to these elements, their descendants or attributes, a callback (i.e., redirectContent() in Listing 6.5) is triggered to construct and to send a change message to the slave application.

The change message contains a list of changes represented as change objects. A change object contains different information depending on the change type, as follows:

- for a newly added node, the change object contains the node tag name, its attributes, its parent node, its previous siblings and its descendants if any.
- for a moved node (i.e., reparented), the change object contains the mutation-summary internal id for the node concerned in this change, its new parent node and its new previous siblings if any.
- for a modified attribute or a modified text node, the change object contains the mutation-summary internal id for the node concerned in this change and the new value for this attribute.

[1] Mutation-observer `http://www.w3.org/TR/domcore`

[2] Mutation-events, `https://developer.mozilla.org/en-US/docs/Web/Guide/Events/Mutation_events`

[3] `http://lists.w3.org/Archives/Public/public-webapps/2011JulSep/0779.html`

However, the Mutation Summary library suffers from some limitations. For instance, it cannot detect changes made to standard JavaScript properties that are associated to HTML attributes unless the HTML attributes are synchronized with the JS properties. For instance, the '.id' property and the HTML id attribute are continuously synchronized. This synchronization does not always guarantee the same value as it is the case with the href attribute and '.href' property. The '.href' property returns the absolute URL while the href attribute returns the value within the HTML document. As long as there is synchronization, mutation-summary can monitor the changes. But the problem is when the changes made on JS properties are not synchronized to the HTML attributes. In this case, the mutation-summary is not aware of the change. For instance, the 'checked' attribute that is used with HTML input checkboxes, the 'value' attribute that is used for example to write text in HTML input textboxes, etc. In addition, mutation-summary cannot detect if an event listener is added dynamically to a DOM node using the '.addEventListener()' JavaScript native method especially that these event listeners are not synchronized with their corresponding HTML attributes.

Listing 6.7: Monkey Patching appendChild method

```
1 var origAppendChild = Element.prototype.appendChild;
2 Element.prototype.appendChild = function(child){
3   // dynamic distribution for newly created element
4   var annotation = analyze(child);
5   annotate(child, annotation);
6   origAppendChild.call(this, child);
7 }
```

To overcome these limitations, as well as to support the dynamic division and distribution described in Section 6.2.2 and to update the configuration of the mutation-summary we use the Monkey Patching technique [31] to extend native functions with custom code. The following extensions are done:

- the '.appendChild' method is extended as shown in Listing 6.7 to detect the addition of new elements and to trigger dynamically the annotation and the distribution of these elements;
- the '.setAttribute' method is extended to update the Mutation Summary configuration whenever a new attribute is set on the slave elements. This has the intention of monitoring and mirroring the newly created attributes.
- the '.addEventListener' method is extended 1) to catch the event listeners that

are declared dynamically using JavaScript and 2) to replace an event handler on the slave application with a call to the master application.

These extended functions should be loaded before the application logic loads to ensure that the relevant application events will be captured.

On the slave application, user inputs are sent to the master application upon user request (e.g., the user pushed a button to submit his input data). In the absence of user request (e.g., the user entered his data but did not push the submission button), user inputs are sent upon their availability based on a polling mechanism and using a timer. In the case where only a user request is available (e.g., the user clicked on a button to stop a running video), the request is immediately redirected to the master. In all cases, a function is triggered on the slave device to notify the master about the user queries. First, it identifies 1) the type of the user interaction, e.g., a click, user inputs, etc., 2) the concerned DOM element(s) and 3) the provided input data if any or/and the type of the fired event. Second, it prepares a change message and sends it across the communication channel to the master application.

6.4 State Distribution Evaluation

Testing the runtime performance of our system consists in considering it as a black box and checking if the overall performance fulfills the main objective: to have a functional multi-screen application that ensures the state distribution during run-time without unacceptable delays.

6.4.1 Characterizing the dynamicity of the Video Semantic application

We consider the Video Semantic application developed by Mozilla characterized in Table 3.1. This application is highly dynamic. It is developed using the PopcornJS library [38] that aims at creating time-based interactive media on the web. Figure 6-2 is a screen-shot of the application during run-time. It shows a video with subtitles and four blocks of information representing a Google map, a Wikipedia text, Flickr images and Lastfm music, that are dynamically updated depending on the video time.

In the application, 11 parts are subject to at least one change during the runtime and for the duration of the video, i.e., 4 minutes and 46 seconds. Table 6.1 represents

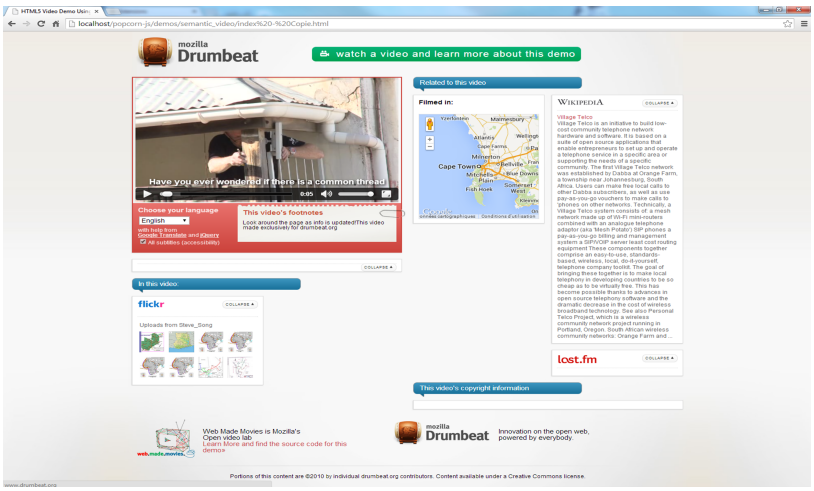


Figure 6-2: Main Application

Table 6.1: Number of changes related to the dynamic parts of Video Semantic

Count / Parts	Total Changes
Subtitle	222 changes
Flicker	8
Wikipedia	26
Googlemap	24
Attribution	26
Footnote	8
Tagthisperson	40
Lastfm	2
Webpage	4
Complete Application	360 changes

the list of these parts, e.g., flicker, wikipedia, googlemap, subtitle, etc.

For each part, we calculated the number of changes. For instance, the Flicker part changes 8 times, the Wikipedia part 26 times, the Googlemap part 24 times, the subtitles part 222 times, etc. The total number of changes is 360 for the application as a whole.

These 360 changes were fired at 267 different instants for the duration of the video, as Table 6.1 shows. This means that on average every 1.07s (i.e., 286s/267 = 1.07s) one or more changes are happening.

The subtitle changes happen at 202 distinct instants. The changes for the remaining parts happen at 83 distinct instants. We note that there are instants at which there are simultaneous changes between the subtitles and the remaining parts (202+83 = 285>268).

Table 6.2: Number of changes related to the dynamic parts of Video Semantic

Count	Distinct Change Instants	Number of Changes
One change only	205 instants	205 changes
Two or more changes	62 instants	155 changes
Total	267 instants	360 changes

At 205 different instants (205*100/267 = 77% of total instants), only one change was triggered, in total 205 changes (205*100/360 = 57% of total changes). At 62 different instants (62*100/267 = 23%), two or multiple changes are triggered simultaneously. In total, 155 changes (155*100/360 = 43% of total changes) are triggered during these 62 different instants, as shown in Table 6.2.

Table 6.3: Simultaneous changes between the dynamic parts of Video Semantic

Simultaneous changes between:	Distinct Change Instants	Max. number of simultaneous changes	Nb of changes
Subtitles - Subtitles	20 instants	2	40 changes
Others - Others	33 instants	5	88 changes
Subtitles - Others	18 instants	6	42 changes

Table 6.3 reports about the application parts that were simultaneously changed. The objective is to study the relations that exist between the different application parts. This is in the intention of building an idea about the amount of the required exchanges that will be redirected from the master component to the slave component.

As it is going to be described in the following section, the subtitles part are attached to the video element on the master component. The remaining parts, listed in Table 6.1, belong to the slave component. We refer to these remaining parts as 'Others' in Table 6.3.

At 20 distinct instants, two simultaneous changes happened between the subtitles. In this case, a simultaneous change refers to deleting one subtitle and adding a new subtitle at the same instant.

At 33 distinct instants, two to five simultaneous changes happened between the 'Others' parts. In this case, a simultaneous change refers to showing a wikipedia text related for example to Telco Village simultaneously with a google map for Telco Village for example. On average, there are 2.7 changes (88/33 = 2.7) per instant.

Finally, at 18 distinct instants, two to six simultaneous changes happened between the subtitle part and the 'Others' parts. In this case, a simultaneous change refers for example to showing a wikipedia text related to Telco Village simultaneously with the appearance of a subtitle containing the word 'Telco Village'. On average, there are 2.4 changes (42/18 = 2.4) per instant.

We can see that at least 18 change messages should be sent from the master to the slave component. Every message contains on average 2.4 changes to be applied on the slave.

6.4.2 Runtime analysis of the state distribution of the Multi-screen video semantic application

To test the performance during run-time, we implemented the complete system as a Google Chrome extension. We load the extension on the Google Chrome browser of a desktop that we consider as a master device. We used another PC as the slave device. Both devices are connected through an ethernet connection in LAN and communicate through the COLTRAM platform.

We run the Video Semantic application on the master device and let our system distribute the application into master and slave web services. Figure 6-3 and Figure 6-4 represent respectively the master and the slave components. The master component contains only the video element with the subtitles, that occupy the full-window. The slave component contains the remaining elements notably the header, the footer, the Wikipedia, the Googlemap, etc.

Figure 6-3: Master Application after splitting the Semantic Video application between large devices

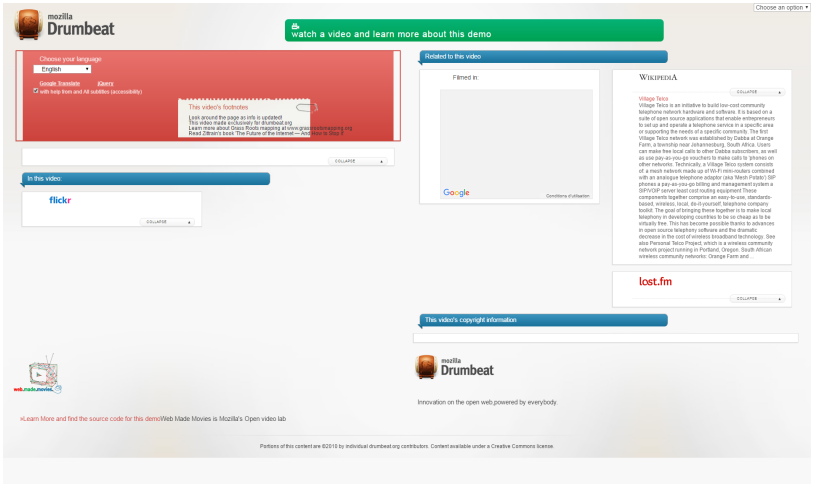


Figure 6-4: Slave Application after splitting the Semantic Video application between large devices

Then, we ran the video on the master and we inspected the number and types of DOM changes that were redirected to the slave component.

Finally, we measured the delays that were added by 1) our system and by 2) the multi-screen platform COLTRAM to check whether our system is realistic and whether it affects the overall functionality of the original application.

6.4.2.1 DOM updates

Table 6.4: Number of Slave Events

Event types	Removed	AddedOrMoved	Attributes
Slave VideoSemantic	27	1001	555

Table 6.4 summarizes the inspection results: 27 updates were received to remove DOM elements from the slave DOM tree. 1001 updates were received to add or move DOM nodes, including text nodes. 555 updates were received to edit, add or remove an HTML attribute. The communication of these updates from the master to the slave required 295 messages in total.

Listing 6.8: Example of a change message as a JSON object

```
{"f": "slave_domChangesIntegrator",
 "args":[
                {"removed" : null},
                {"addedOrMoved": [{"nodeType: 1,
                                   "id": 474,
                                   "tagName": "DIV",
                                   "parent": {"id": 106},
                                   "attributes": {"style": "
                                      position: absolute;
                                      left: 0px; z-index: 1;
                                      visibility: inherit"}
                                   }]},
                {"attributes":[{"id": 192,
                                   "attributes": {"class": "
                                      hover"}
                        }]},
                {"text":null}
          ]
```

```
15 }
```

Each message contained on average 5 DOM updates. Listing 6.8 contains a JSON object corresponding to a message sent from the master to the slave. The JSON object has two property-value pairs, the first one (i.e., f) corresponds to the function name that should be called locally on the slave side. The second property corresponds to an array of changes. We can see that one DIV element was added to the DOM tree, it is a child of a node with id equals to 106, it has 474 as an id and only one attribute (i.e., style with its value). In addition, the node that has an id equals to 192 has an updated class attribute with “hover" as a value.

By comparing the number of messages sent to the slave (i.e., 295 messages) to the 360 application changes of Table 6.1, we can see that on average 1.2 changes on the level of the application user interface triggered the sending of 1 message from the master to the slave.

6.4.2.2 Communication and System Delays

We proceed to measure the communication delay induced by the COLTRAM platform and by our system between two machines (i.e., one desktop and one portable computer) connected via Ethernet. The delay value gives an idea about the presence of visual incoherence for the videoSemantic application. We note that visually no incoherence was detected during runtime between the master component and the slave component.

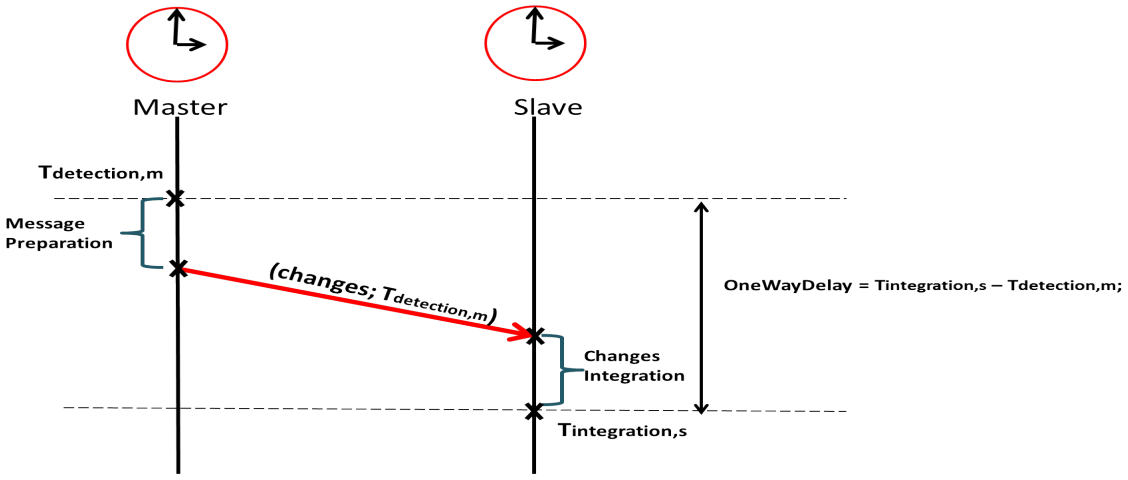


Figure 6-5: Model of ideal one-way delay between master and slave

More specifically, we want to measure the one-way delay between detecting a

change on the master component (i.e., Tdetection,m) and integrating this change on the slave component (i.e., Tintegration,s) as shown in Figure 6-5. The one-way delay is the sum of the system delay (related to the synchronization mechanism) and the communication delay (related to the network and the platform) as expressed in Equation 6.1.

$$OneWayDelay = Delay\_communication + Delay\_system \tag{6.1}$$

Practically, the one-way delay can be calculated following Equation 6.2:

$$OneWayDelay = Tintegration, s - Tdetection, m; \tag{6.2}$$

Equation 6.2 assumes that the clocks of both machines are synchronized. The challenge in real scenarios is that there is always a clock drift that needs to be considered. A first step here consists in calculating this clock drift to exclude it later from the one-way communication delay. Figure 6-6 illustrates the mechanism we set in this

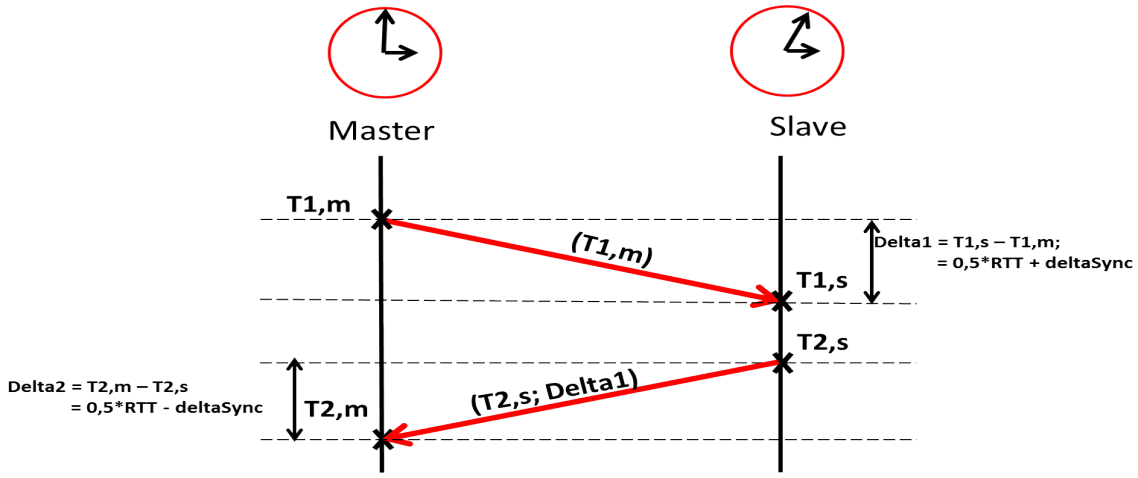


Figure 6-6: Estimation of the clock shift between master and slave

intention. The master, running on the desktop, starts the mechanism and sends a message containing only his sending time (i.e., T1,m) to the slave that is running on a portable computer. Once the slave receives this message, it gets its reception time (i.e., T1,s) and calculates Delta1 that is the difference between T1,s and T1,m. Then, the slave sends Delta1 and his new sending time (i.e., T2,s) to the master. On his side, the master calculates Delta2 that is the difference between his reception time (i.e., T2,m) and T2,s. Both Delta1 and Delta2 include the clock drift (to which we refer as deltaSync) and half the round trip time (RTT) as follows:

- Delta1 = 0.5*RTT + deltaSync;
- Delta2 = 0.5*RTT - deltaSync;

By solving these two equations, we get the following equation:

$$deltaSync = \left| \frac{Delta1 - Delta2}{2} \right|. \tag{6.3}$$

We note that these equations are only valid if connections from the master to the slave and vice versa are symmetric. The master uses the values of Delta1 and Delta2 to calculate the deltaSync value and to compute RTT, i.e., the sum of Delta1 and Delta2. The RTT value corresponds to double the value of Delay_communication.

This mechanism is repeated enough times (i.e., 600 times) before we compute the average of deltaSync that we take as the value for the clock drift. Similarly, we take the average of RTT values as the value for RTT. In our experiments, RTT varies between 6 ms and 41 ms, on average 8 ms and in consequence Delay_communication varies between 3 ms and 21 ms, on average 4 ms on Ethernet.

Note that similar experiments were conducted on WiFi to measure the communication delay induced by the COLTRAM platform. Results show that on average RTT_wifi is 90 ms.

Considering the clock drift on the slave side, the real OneWayDelay is calculated as follows in Equation 6.4:

$$OneWayDelay = (Tintegration, s - Tdetection, m) - deltaSync; \tag{6.4}$$

It is the difference between the time the slave integrated the change (i.e., Tintegration,s), the time the master detected the change (i.e., Tdetection,m) and the deltasync. This OneWayDelay includes on the one side the communication delay induced by COLTRAM and by the network, and our system processing time on the other side.

After calculating the deltaSync, we run the video on the master component. For every message received on the slave, the slave computes the OneWayDelay following Equation 6.4. In the text below, we present and analyze the obtained results.

In Figure 6-7, we show the variation of the delay values with every received message on the slave for the lifetime of the video. The variations we present are the average values obtained after running 10 times the videoSemantic application. Figure 6-7 shows that a burst of 5-15 consecutive messages on average implies a slightly

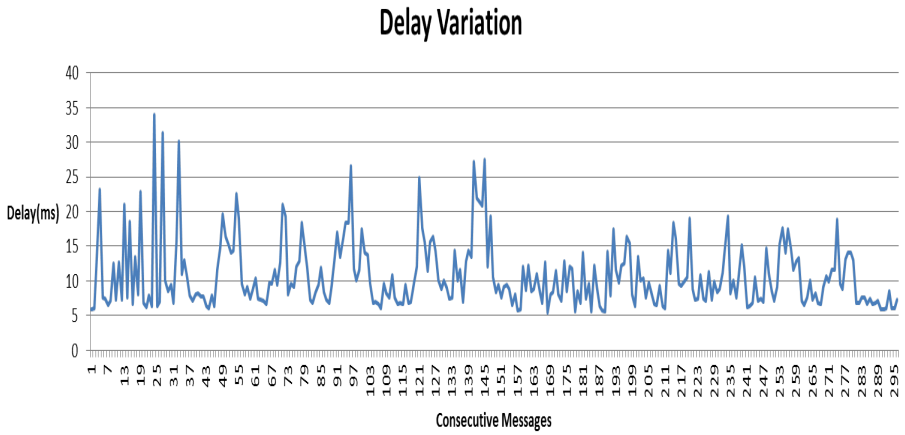


Figure 6-7: Delay Variation in function of time

modified delay value and then the delay changes abruptly. This behavior can be seen for all the 295 messages reported in the figure. The one-way delay (i.e., including Delay_communication and Delay_system) varies between 5 and 34 ms, on average 11 ms [4] on ethernet. Having both the one-way delay and Delay_communication and following Equation 6.1, Delay_system varies between 2 ms and 13 ms on Ethernet.

Using WiFi, Delay_system would increase by 45 ms (i.e., RTT_wifi/2) and the total one-way delay would be between 47 ms and 57 ms.

In Figure 6-8, we present for each delay value the number of messages to which it corresponds. On average more than 65% of messages produce a delay smaller than 11 ms and 95% of messages produce a delay smaller than 20 ms.

The values of these delays, whether on ethernet and on Wifi, are not perceivable by the human eye based on Miller [37]. Miller [37] states that a response time of 100 ms is perceived as instantaneous.

Comparing the one-way delay (i.e., 34 ms at max for ethernet or 57 at max for Wifi) and Delay_system to the threshold of 100 ms, our system mechanism provides an instantaneous synchronization.

[4] We note that these values are the same for UPnP services and Bonjour services in this experiment setup.

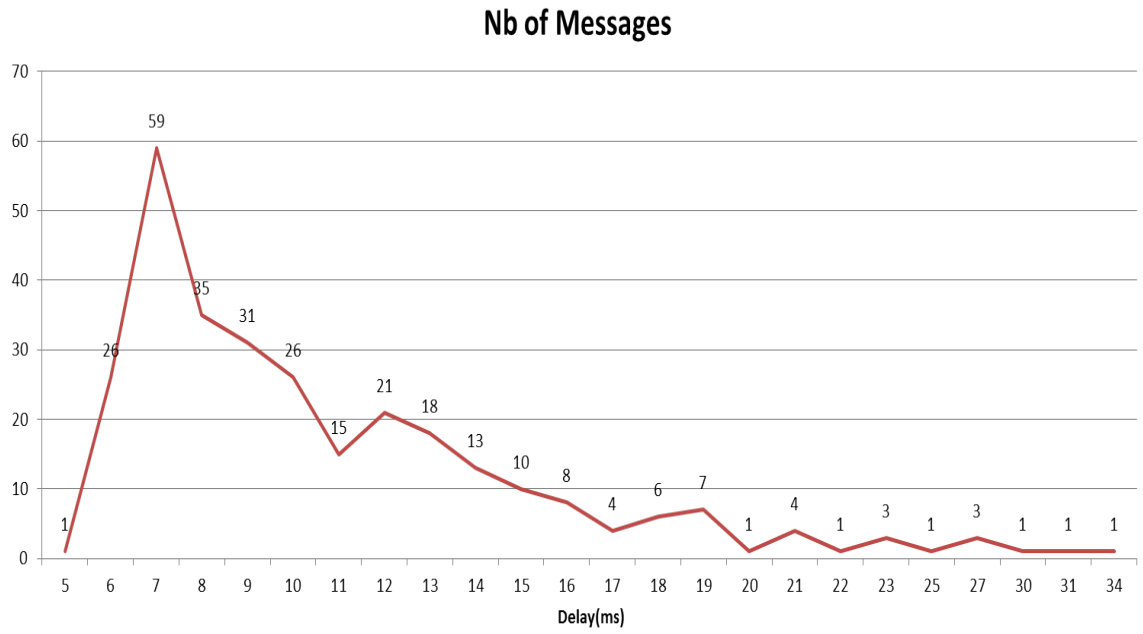


Figure 6-8: Number of messages corresponding to the delay values

6.5 Proof-of-concept: Extending the Refactoring System

All along the thesis, we have been addressing the authoring of multi-screen applications for two devices, i.e., one master and one slave. We have identified some use-cases in the literature where the number of devices involved in the multi-screen experience exceeded two devices. For instance, multiple devices are aggregated to widen the visualization of an application [26], to create a panoramic street views across three devices [57], to schedule a large conference on a wall-size display [32], etc. Another example is an interactive presentation where the presenter and the public share the same version or a customized version of the presentation, or a multi-user game where every user has his own private user interface but they all share one screen where results and interactions take effect.

In our work, we did not focus on these use cases (i.e., considering user profile to deliver customized content, aggregating content from multiple sources (e.g., cameras) to create a panoramic view, or online games). But as a first step towards supporting them, we decided to extend our refactoring system to support the segmentation, the distribution and the synchronization among three devices as it will be described in the next Sections. The relayouting phase is not affected by the increase of the number

of devices since it is applied to each component individually.

To validate the extended system, we tested it on the videoSemantic application in Section 6.5.5.

6.5.1 Extending the application model to three components

We decided to keep the main application model that consists of one master component having a modified version of the main application and one (or more) slave components having each a part of the main DOM tree. It is always the master component that sends the DOM updates to the slave components (i.e., first slave, second slave). Each slave on his turn sends the user data and requests upon their availability to the master component.

Concerning the slaves, we considered two cases:

1. Both slaves are identical. We call them cloned slaves.
2. Slaves are different. Each slave has distinct content.

The application model with cloned slaves does not impose challenges to the extension of the refactoring system. This is because the content is still divided between two components. Then the slave content is duplicated and serialized to the first and the second slave. For the UI synchronization step, the change messages that are detected on the master are the same for both slaves. Thus, it is enough that the master sends simultaneously the same message to both slaves to ensure the state synchronization. If one user interacts with one slave, the corresponding slave sends the change to the master and the master on his turn update the view of both slaves, resulting in coherent slave views.

In the second case where slave components have different content, additional questions and challenges need to be addressed. We summarize them as follows: What are the additional functions that can be extracted from the device features and that map to the application content? Should we allow peer-to-peer communications between the slaves or is it better to always centralize them on the master component?

In both cases, there are additional challenges related to the discovery and to the communication between the master on the one side and the slaves on the other side. Our system has to discover three devices in the network and to characterize them. Out of the device features, the system usually should derive functions that may correspond

to the application content. But by lack of time to refine the characterization of video-centric applications and the mapping to device features, we decided to consider a third customized function, i.e., 'maps'. More details are found in Section 6.5.2.

In the following sections, we briefly describe our solution to solve the challenges while focusing only on the changes brought to the refactoring system, specifically for the case of distinct devices.

6.5.2 Extended UI Division

As mentioned earlier, in addition to our 'multimedia' and 'interactive' functions, we adopted the 'maps' function. Using this function, the map content should be separated from the remaining content, i.e., interactive and multimedia content.

Here, we consider that the input to the UI Division step is an application where only the DOM element associated to the map element is annotated as belonging to the second slave. The remaining DOM nodes are not yet annotated, but they will be during the annotation projection in the UI Distribution step in Section 6.5.3.

Similarly to the basic UI Division, the geometric and the logical trees are constructed and characterized with three functions instead of two. We impose an additional rule that resolves multiple functions into one function. We consider that the 'maps' function has a higher priority than the 'interactive' function, the same as 'multimedia' function has a higher priority than the 'interactive' function. Thus if we have a map element that is interactive, its corresponding logical node will have the 'maps' function.

During the processing of logical nodes:

- 'maps' content should be separated from 'interactive' content,
- 'maps' content should be separated from 'multimedia' content, and
- 'multimedia' content should be separated from 'interactive' content.

As a result of the processing, the segmented application represented by the leaf logical nodes is annotated with three different functions.

6.5.3 Extended UI Distribution

We recall that UI Distribution consists of three phases: projection, propagation and MSA creation. The only change for the projection and the propagation steps is solely at the level of the annotations.

In the case of cloned slaves, there are only two functions on the logical tree similarly to the basic implementation of our system. Thus, on the DOM tree only three annotations can be found, i.e., 'device1', 'device2' and 'dev1/dev2' annotations. In contrast to this first case, the second case with two distinct slaves requires having additional annotations. The first is 'device3' that is reserved for logical nodes having 'maps' as a function. The second is 'dev1/dev3' (and 'dev2/dev3' resp.) that is reserved for DOM nodes grouping content from the master component (and the first slave component resp.). The third is 'dev1/dev2/dev3' that is associated to DOM nodes grouping content from the three components.

During the MSA creation, for cloned slaves, DOM elements annotated with 'device2' are serialized simultaneously to the first slave and to the second slave and then made hidden on the master component.

For distinct slaves, DOM elements annotated with 'device2' (and resp. 'device3') are serialized to the first slave (and resp. second slave) and then made hidden on the master component.

6.5.4 Extended State Synchronization

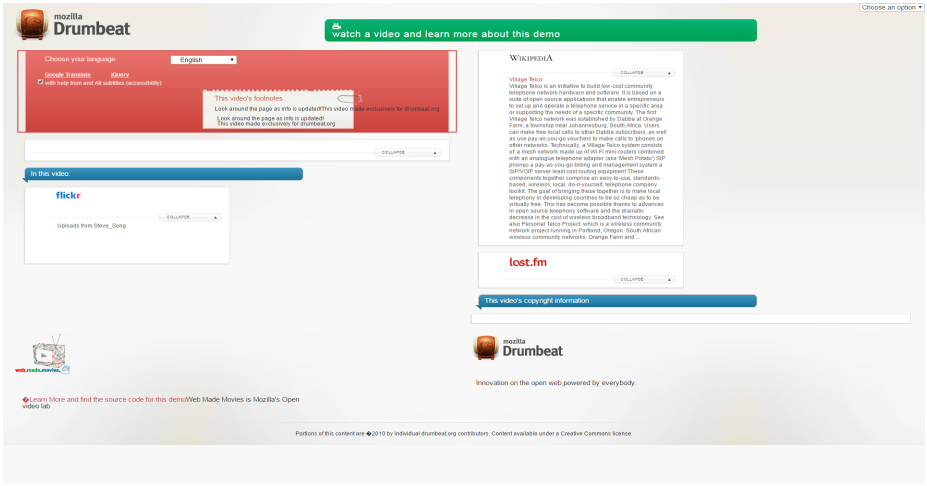
In the second case, for each slave a new instance of the mutation-summary is created and configured to watch the changes made respectively to 'device2' DOM elements and to 'device3' DOM elements. Upon the detection of a change on the first slave (resp. second slave), a callback function is called to send these changes to the corresponding slave using its COLTRAM interface.

6.5.5 Validation of the VideoSemantic application

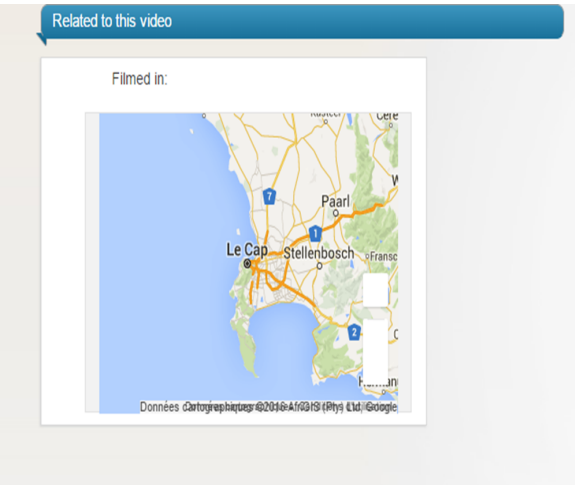
We have validated the system on the semanticVideo application in both cases.

For the case of distinct slaves, we show the segmentation results in Figure 6-9. The master component contains only the video element with the subtitles, in Full-Window Design mode as shown in Figure 6-9(a). The first slave component shown in

(a)



(b)



(c)

Figure 6-9: (a) Master component containing only a video element in Full-window Design mode (b) First RWD Slave component containing interactive content, except the map block (c) Second RWD Slave component containing only the map block

Figure 6-9(b) is a responsive application that contains all the content of Figure 6-2 except the video element, and except the map block with its corresponding header. This latter, i.e., was reserved for the second slave component shown in Figure 6-9(c).

In the case of cloned slaves, both slave components were visually synchronized and no time shift is visually perceivable during runtime. The slave components stayed synchronized during the video playing for both cases, i.e., cloned slaves and distinct slaves.

6.6 Conclusion

This chapter focused on the transformation of the static master and slave components into a multiscreen application. Each of the master and the slave went through two additions (i.e., mirroring and adaptation to the COLTRAM platform) to become capable of providing the synchronization and the redirection operations.

During runtime, these two operations provide the state distribution mechanism that can function independently from our system. Being the ultimate objective, the state distribution was evaluated on the VideoSemantic application that is highly dynamic. Results show that the DOM changes were continuously captured during the lifetime of the video element and no visual incoherence was detected.

This test helped us in validating our application model and in showing that restricting the state distribution to the DOM tree state can be enough for some applications. In contrast, this model imposes some constraints for HTML elements that use JavaScript to fill their content, e.g., canvas, Google maps, etc. In such cases, these HTML elements should be kept on the master where the logic resides.

As a perspective, a study can be conducted to deal with these constraints and to extend the state concept to include also JavaScript objects related to these elements, and the browser native state.

In addition, we evaluated the performance of the overall system. Results show that our system is very realistic during runtime. The total delay, including system and communication delays, is of 34 ms at max on Ethernet, on average 11 ms, and the delay produced by the system processing varies between 2 ms and 13 ms.

Chapter 7

Conclusion and perspectives

7.1 Summary

In this dissertation, we have addressed the reuse of video-centric web applications for the automatic creation of distributed applications in a multi-screen context.

The literature on this subject focuses on the design of an application model, on the creation of a platform adapted to the application requirements and on its functioning after the distribution. But the works in the literature remain inconclusive about the content distribution, since in most of the cases a manual distribution is adopted. Another gap was identified and it is related to the layout adaptation after the distribution. This dissertation sought to cover these two gaps.

The work has been carried out as follows:

- The design of a model for the distributed applications
- The characterization and the exploitation of the multi-screen environment for the user interface distribution
- The adaptation of the distributed application to the multi-screen environment, including to the multitude of devices and to the multi-screen platform
- The development and the evaluation of a refactoring system that integrates the above three points all together.

In Section 7.2, we synthesize the findings of this thesis while citing some of its limitations. Section 7.3 is an outline plan for further research in the topic.

7.2 Synthesis

Among the results, we have shown that single-screen applications can be reused to create functional multi-screen applications. The key points here are: the application division and the design of a model for the multi-screen applications while optimizing the damages on the application documents.

Concerning the first point, we have shown that our hybrid segmentation approach not only identifies coherent blocks of content, but it also separates interactive content from multimedia content. Thus, it allows the distribution of the “best-fit" content to the “best-fit" device following device features.

In this work, the distribution was limited to the DOM tree while keeping the application logic on one side of the distributed application. This resulted in a particular multi-screen application model consisting of a master component and its corresponding slave component. These components are made capable of executing two cross-device operations: redirection and synchronization, using a monitoring technique for the DOM tree only.

Results show that the DOM monitoring was enough to provide the state distribution and the state coherence especially that no visual incoherence was reported during our tests.

After distributing the content, we have also covered the dynamic adaptation of the component layouts to resolve two main layout anomalies: blank spaces and horizontal scrolling. To eliminate blank spaces from the master layout, we neglected the DOM structure and imposed a new design calculated relatively to the window dimensions. For eliminating the horizontal scrolling from the slave layout, the DOM structure was respected.

The evaluation of both algorithms, after the quantification of each of the anomalies, revealed: (1) the absence of blank spaces on the master components, (2) the absence or the reduction of the horizontal scrolling on the slave components. Both algorithms are relatively basic but they resolve the layout problems created by our application model or by the dynamic nature of the multi-screen environment. As a perspective, the two algorithms are open for refinement and improvement, for instance to consider the importance of each content on the page to re-position it and re-dimension it accordingly and to respect the aspect ratio during the content re-dimension.

Results obtained after evaluating the performance of the overall system show that

our system is realistic during runtime with a max delay of 34 ms, on average 11 ms on ethernet. This delay that is small comparing to the 100 ms defined by Miller [37] shows that the response time of our system is instantaneous. In addition, the system is compatible with all the tested environments for developing web applications. This helps in enlarging its applicability to most web applications.

7.3 Perspectives

Several important issues could not be investigated during this thesis owing to the precise thesis objectives in the domain of document engineering, while focusing solely on web applications. This includes:

- Extending the system applicability to other XML-based languages different from HTML, such as SMIL and NCL. This requires the study of the nature and the type of these applications to verify 1) whether there is a way to analyze every element to determine its function and its geometrical parameters, and 2) whether there is a runtime programming interface that allows interfacing the mirroring code.
- Extending the environment exploitation to include not only the device features, but as well the user preferences. This requires conducting a user-study to investigate about user preferences in multiple scenarios.
- A usability study can be conducted to evaluate qualitatively 1) the layout redesign of each of the master and the slave components, and 2) the results of the content distribution.
- Extending the mirroring mechanism to synchronize not only the DOM tree states but as well the states of JavaScript objects. Such a mechanism eliminates our system limitation of keeping special content (e.g., canvas) next to the main logic on the master component.
- Studying the possibility of integrating approaches from the web semantic domain to give a semantic meaning for the blocks produced at the end of the UI division.
- Pushing the design of HTML-based applications one step further to optimize application DOM trees, especially given that only 14% of DOM elements in our dataset are visible and only 3% of DOM elements take part in the application

user interface. We believe that using the logical tree representation jointly with the basis of our FWD algorithm, (i.e., to design the user interface independently from the DOM tree), we can reduce the depth and the number of nodes in the DOM tree and still reproduce the application look.

Another type of perspective is identified and it is related to making our system into a product. In our implementation, we slightly changed the main DOM tree by adjusting the visibility of slave contents. One challenge here is to circumvent the case where the visibility of these slave contents decides on the application functionality. This challenge can be solved for instance by keeping an updated record (using JS objects) of the slave content visibility on the master application.

In addition, the separation of a video element from its control bar is not possible for HTML5 videos and HTML5 audios since the control bar is natively defined within them. To circumvent this limitation, a customized control bar can be added to the application DOM tree and can be linked to the video element using JavaScript and then it can be separated from the video element. We have developed a first prototype that validates the importance of this perspective if we want to make our system a final product.

Chapter 8

Resume en francais

8.1 Introduction

La prépondérance des technologies du Web et leur évolution continue, surtout avec l'émergence de HTML5, a conduit à des applications Web puissantes. Ceci permet aux utilisateurs d'accéder au même contenu à partir de différents dispositifs. La consommation du contenu par l'utilisateur a changé en raison de cette prépondérance et du nombre d'appareils connectés disponibles pour un seul utilisateur, chacun d'eux possédant des caractéristiques physiques différentes, à savoir, la taille de l'écran ou les types d'entrée-sortie. Une étude Google [23] indique que 90% de nos interactions ne se limitent plus à un seul appareil. Dans cette thèse, nous supposons qu'une application multi-écran est constituée de multiples Composants. Chaque composant correspond à un contenu distinct qui offre une tâche spécifique. Chaque composant fonctionne sur un dispositif distinct. Ces composants communiquent les uns avec les autres pour fournir une utilisation complémentaire.

Les applications multi-écrans imposent de multiples défis pour les développeurs. Tout d'abord, ils doivent concevoir une application qui tire partie de l'environnement multi-écran. Ensuite, ils doivent déterminer comment le contenu de l'application sera répartis sur les différents dispositifs en fonction de leurs capacités. Ils doivent également gérer la synchronisation et la cohérence des contenus distribués. En outre, les développeurs doivent fournir un rendu uniforme du contenu à travers les dispositifs.

L'utilisation des technologies web aide à réduire la complexité de ces tâches et augmente la possibilité de déployer une application sur divers dispositifs. Alors que des travaux scientifiques visent à développer des applications multi-écrans à partir

de rien, notre objectif est de réutiliser des applications existantes déstinées pour un écran unique et de les transformer.

Plusieurs problèmes se posent lorsque nous essayons de réutiliser des applications qui ne sont pas destinées à l'utilisation multi-écran complémentaire. Tout d'abord, les applications ne sont pas nécessairement développées de façon modulaire facilitant l'identification des composants indépendants de contenu afin de les distribuer. Deuxièmement, ces applications ne sont pas destinées à fonctionner simultanément sur plusieurs appareils avec différentes caractéristiques physiques, en particulier sur les appareils mobiles avec un petit écran. Par conséquent, l'expérience utilisateur peut être rompue par la distribution à travers les dispositifs. Troisièmement, une application web consiste en un ensemble de documents, i.e., document HTML qui détermine le contenu et sa structure, un document JavaScript contenant la logique et des document de styles (CSS) pour la mise en page du contenu. Il existe des relations étroites entre ces trois derniers. La séparation du contenu d'une application implique une charge de travail supplémentaire sur la logique et sur la mise en page si nous voulons satisfaire un utilisateur.

L'objectif principal de cette thèse est d'étudier et de relever les défis liés à la création d'applications multi-écrans, notamment à partir d'une application mono-écran. Les objectifs intermédiaires comprennent: 1) la création d'une interface utilisateur distribuée, 2) l'adaptation à une multitude d'appareils et à la quantité de contenu de chaque côté de l'application multi-écran, et 3) la synchronisation et l'orchestration entre les composants de l'application distribuée. De plus, ce travail de thèse comprend la caractérisation des applications vidéo-centrique pour identifier la présence d'une dualité entre le contenu et les caractéristiques des dispositifs. Cette caractérisation a pour but d'associer une partie de l'application au meilleur dispositif sur le réseau. L'objectif ultime de cette thèse est de concevoir un système automatisé dédié pour les utilisateurs finaux et qui élimine la nécessité de l'intervention des développeurs.

Au cours de cette thèse, nous avons conçu et développé un système de refactoring des applications existantes, qui est guidé par les caractéristiques de l'environnement multi-écran pour produire automatiquement des applications multi-écrans.

Le système de refactoring peut être utilisé directement et de manière transparente par un utilisateur final. Il suppose la présence d'au moins deux dispositifs découverts dans le réseau. Cette hypothèse est cruciale puisque notre système exploite les caractéristiques physiques des dispositifs pour guider l'analyse des contenus des applications et par conséquence la conception de l'application multi-écran. Le système est dédié principalement, mais sans être limité, pour les applications multimédia et

les applications vidéo-centriques consistant en au moins un élément vidéo.

Les sections qui suivent s'articulent de la façon suivante. Section 8.2 dresse l'état de l'art des plateformes multi-écran et des travaux ayant comme objectif la création des applications multi-écrans.

Dans la Section 8.3, nous introduisons le système global que nous avons mis au point avec ses principales caractéristiques, ses entrées et le modèle adopté pour les applications multi-écrans. En outre, nous décrivons et caractérisons le jeu d'applications utilisées tout au long de cette thèse pour valider les composants du système. Les différents composants du système, i.e., la division du contenu, la distribution du contenu, la synchronisation et l'adaptation de la mise en forme des applications, sont brièvement décrits. Les méthodes d'évaluation ainsi que les résultats obtenus seront détaillés par la suite.

Dans la Section 8.4, nous résumons les travaux de cette thèse et nous tirons des conclusions concernant la validité de notre système dans des situations réelles. Enfin, nous présentons les perspectives de notre travail.

8.2 Etat de l'art

Dans des travaux antérieurs [25], un 'WebSplitter' a été proposé pour diviser les applications web, sur la base de méta-données contenus dans un fichier. Ce fichier est unique pour chaque application. Il contient les associations entre les contenus des applications et les appareils possibles. L'architecture du 'WebSplitter' exige un proxy qui sépare le contenu de l'application dans des vues partielles et un composant côté client qui reçoit des données envoyées par le serveur. L'architecture est centralisée et requiert une distribution manuelle de chaque élément DOM de l'application.

Dans ses recherches, Cheng [13] a proposé un navigateur virtuel capable de séparer la logique d'une application de son rendu. La logique est maintenue dans une page web virtuelle. Automatiquement, le navigateur virtuel divise l'arbre DOM principale (DOM) en plusieurs sous-arbres DOM et associe ces sous-arbres au dispositifs comme noté dans un fichier d'indices qui est spécifique à chaque application. Ce fichier d'indices est créé manuellement par le développeur. Les opérations multi-appareils sont exécutées de manière centralisée en fonction du navigateur virtuel qui constitue un proxy entre le serveur Web et les navigateurs des appareils des utilisateurs finaux.

Bassbouss et al. [4] décrit comment transformer des applications traditionnelles

en applications multi-écrans. Une application est d'abord développée comme une application mono-écran, mais afin de la distribuer elle nécessite un navigateur spécifique multi-écran. De même, la distribution du contenu se fait sur la base de méta-données fournies manuellement par le développeur. Le modèle de l'application générée est maitre-esclave.

Zorrilla et al. [60] a proposé une architecture de distribution d'une application mono-écran sur plusieurs dispositifs tout en assurant aux utilisateurs une expérience cohérente à travers les dispositifs. L'architecture décide de la meilleure configuration entre le contenu (sous forme de Web Components) et dispositifs et cela d'une façon dynamique en se basant sur des indices fournis par le développeur. Ces indices décrivent le comportement ciblé de l'application dans un environnement dynamique multi-dispositif, mais ils ne portent pas d'informations sur le(s) dispositif(s) cible(s).

Contrairement à [25] et [13], notre système a une architecture décentralisée. Semblable à [4], il fournit des applications suivant un modèle maître-esclave. Le point commun pour les travaux ci-dessus est qu'ils exigent un seul environnement de développement qui facilite la création des applications multi-écrans. Mais cela signifie que chaque application mono-écran doit être soit conçue de façon modulaire [60] soit analysée par le développeur pour identifier le différents modules ou blocs qui la composent. Il n'existe pas de procédé automatique pour effectuer une analyse générique qui pourra être appliquée à un grand ensemble d'applications existantes afin de distribuer ces applications.

8.3 Contribution: Système de refactoring

8.3.1 Introduction globale du système

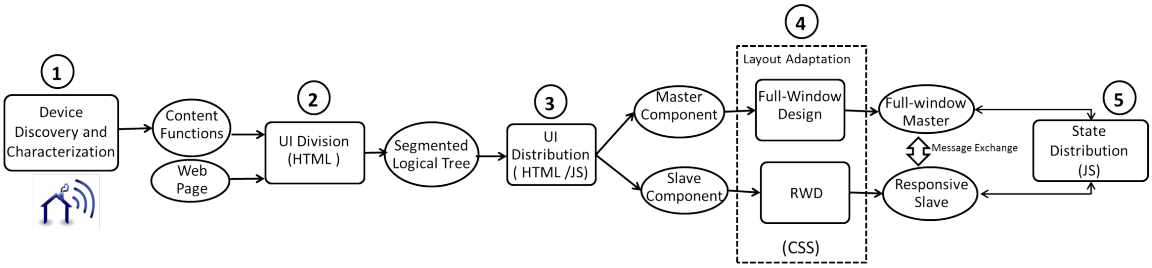


Figure 8-1: L'architecture du système de refactoring

Le système de refactoring détaillé dans ce chapitre et illustré dans la Figure 8-1 se

compose de 5 phases principales: Découverte et Caractérisation de l'environnement, UI Division, UI Distribution, Layout Refactoring et Distribution de l'État.

Functions / Devices	Multimedia	Text	Interactive
TV	++	+	–
PC	++	++	+
Smart Phone	+	+	++
Tablet	+	++	++

Table 8.1: Caractérisation des dispositifs avec des functions

8.3.2 Decouverte et Caractérisation de l'environnement

Le système détecte automatiquement tous les périphériques sur le réseau. Dans le travail de cette thèse, on suppose que uniquement deux dispositifs sont simultanément présents sur le réseau. Les caractéristiques des appareils sur lequel nous nous concentrons sont: (1) nombre d'écrans, (2) taille de l'écran, (3) moyens d'interaction, (4) type des dispositifs, à savoir, TV, PC, tablette ou smartphone. En utilisant un service Web fourni par la plate-forme COLTRAM, nous collectons ces caractéristiques pour les deux appareils. Pour chaque dispositif, nous identifions sa caractéristique dominante que nous considérons comme 'La fonction' du dispositif, en se basant sur Tableau 8.1. Par exemple, dans le cas où deux grands appareils ont été détectés, à savoir, 2 PC, nous comparons en premier leurs tailles d'écran. Nous considérons qu'un PC avec un plus grand écran est plus adapté à l'affichage 'multimédia' que l'autre PC avec un plus petit écran. La caractéristique dominante constitue l'entrée pour la phase de UI Division.

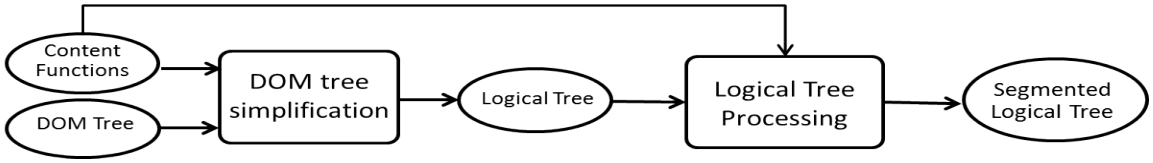


Figure 8-2: Les différentes parties de l'algorithme de segmentation

8.3.3 Division et distribution du contenu

8.3.3.1 Introduction

La phase UI Division consiste à identifier les éléments qui correspondent aux fonctions fournies par la phase précédente et à construire des blocs visuels et sémantiques autour de ces éléments. On note que même dans le cas d'un document HTML mal-formé, un navigateur produit un arbre DOM correct.

En général, la segmentation signifie qu'un gros bloc est décomposé en sous-blocs de tailles plus petites. En revanche, la segmentation d'une application en se basant sur l'analyse de sa structure fonctionne à l'inverse. Notre analyse commence par les particules, à savoir, les éléments DOM, que nous essayons de regrouper pour former des blocs plus grands. Deux principaux défis apparaissent ici: 1) comment déterminer le point auquel nous devrions arrêter l'agrégation des éléments, et, 2) comment détecter les éléments DOM visibles ou pertinents à notre segmentation. En outre, le nombre d'éléments DOM que nous devons faire face varie entre les applications et peut être élevé (plusieurs milliers) comme des statistiques indiquent.

Compte tenu de ces défis, notre algorithme de segmentation se compose de deux étapes: la simplification de l'arbre DOM et le traitement de l'arbre simplifié comme indiqué dans Figure 8-2. Il prend l'arbre DOM et les fonctions de contenu comme entrées et offre une page segmentée et étiquetée en sortie.

8.3.3.2 Simplification de l'arbre DOM et l'étiquetage

L'objectif de simplifier l'arbre DOM est de ne garder que les éléments qui forment l'interface utilisateur (UI). Le processus de simplification analyse et classifie automatiquement tous les éléments DOM afin de les étiqueter avec l'une des fonctions de la section 8.3.2. Puis il décide de la création d'un nœud logique pour former l'arbre logique, en traversant l'arbre DOM avec un parcours en profondeur.

Pour la classification, nous utilisons trois types d'analyse. L'analyse géométrique qui vérifie la visibilité d'un élément DOM. L'analyse statique qui considère les attributs HTML et les noms des balises HTML pour déterminer: les éléments qui ont un comportement correspondant à nos fonctions et les éléments qui devraient être ignorés, par exemple, meta, br. L'analyse dynamique détermine si un élément est interactif en capturant les event-listeners qui sont dynamiquement configurés sur cet élément. Par conséquent, les éléments visibles qui sont classés comme ayant une

fonction (par exemple, une vidéo visible) sont appelés des “éléments pertinents". Un nœud logique est en conséquence créé et il est associé à cet élément, et il est étiqueté avec la fonction de l'élément DOM, à savoir, 'multimédia' ou 'interactive'. Les éléments visibles qui ne disposent pas d'une fonction sont appelés simplement 'éléments visibles', par exemple, les nœuds texte. En fonction de sa position dans l'arbre DOM, un élément visible peut former ou non un noeud logique non-étiqueté. Enfin, des éléments n'ayant pas un aspect visuel sont considérés comme des 'éléments non pertinents' et ne créent pas un nœud logique.

En conséquence, l'arbre logique contient un nombre réduit d'éléments comparé à l'arbre DOM et il est partiellement étiqueté avec des fonctions. Certains des nœuds feuilles sont étiquetés, leur nombre est grand et ils forment géométriquement des petits blocs dans la plupart des cas.

8.3.3.3 Segmentation: Traitement de l'arbre simplifié

La phase de traitement consiste à agréger les nœuds logiques en respectant un ensemble de contraintes, pour produire des blocs étiquetés. Pour regrouper deux nœuds logiques dans un bloc indépendant, ils doivent satisfaire trois conditions: être frères dans l'arbre logique, non étiquetés avec des fonctions différentes, et satisfaire les lois dites 'Gestalt' qui tentent 'de reconnaître des objets comme étant une seule chose' [42] et qui sont basés sur la proximité, la similitude, la fermeture et la simplicité. Une contrainte géométrique supplémentaire est imposée pour déterminer le point pour lequel un nœud logique pourra être considéré comme un bloc final, et cela si ses descendants n'ont pas d'étiquettes différentes. Pour appliquer cette contrainte, nous adoptons la notion de paramètre de granularité (pG) [46] qui détermine le point sous lequel un nœud peut être considéré comme un bloc final. Plus le pG local est grand, moins nombreux sont les blocs produits et mieux les résultats de segmentation sont. Contrairement à BoM [46], on calcule automatiquement et en continu plusieurs valeurs pour PG, à savoir, un PG global et des pG locaux, et cela durant le traitement des nœuds logiques pour adapter la segmentation au contenu de l'application.

Ensuite, l'algorithme de traitement traverse l'arbre logique, en utilisant le pG global. Un nœud étiqueté est fusionné avec son frère si les contraintes ci-dessus sont satisfaites pour les deux parties. Pour un nœud non marqué qui a des descendants avec différents étiquettes, nous traitons le sous-arbre en premier. Pour un nœud non marqué qui a des descendants avec une seule étiquette, si sa superficie est plus grande que pG, alors nous traitons son sous-arbre. Dans le cas contraire, si sa surface relative

est inférieure à pG, nous essayons de le fusionner avec ses frères qui respectent les contraintes ci-dessus. A la fin du traitement, les feuilles de l'arbre logique représentent les blocs finaux et ils sont étiquetés et prêts pour la distribution.

8.3.3.4 UI distribution

La phase UI Distribution représente le moment auquel l'application mono-écran se transforme en une application multi-écran formée d'un composant maître et d'un composant esclave. Cette phase prend en entrée l'arbre logique étiqueté et elle sépare les feuilles de l'arbre qui sont 'interactives' de celles qui sont 'multimédia' pour créer l'interface utilisateur distribuée de l'application multi-écran (à savoir, les documents HTML et JS). Les défis sont premièrement de partitionner l'arbre DOM, en particulier parce que l'arbre logique ne représentent pas tout l'arbre DOM et deuxièmement d'adapter d'une manière transparente le script associé à l'interface de chaque composant.

Pour réellement diviser l'arbre DOM, nous avons besoin de projeter les fonctions des nœuds logiques sur le arbre DOM. La projection est facilitée par la correspondance entre les nœuds logiques d'un part et les nœuds DOM de l'autre part. L'algorithme de projection traverse l'arbre logique à partir des feuilles. Les éléments DOM associés à des nœuds logiques étiquetés avec une fonction 'multimédia' (resp. 'interactive') appartiennent au composant maitre, (resp. au composant esclave). Les éléments DOM dont les enfants directs correspondent à des noeuds logiques de différents de fonctions sont partagés entre le maître et les composants esclaves. En conséquence de cette projection, les éléments annotés sont rares dans l'arbre DOM. Nous décidons alors les annotations des autres nœuds sur la base des éléments annotés. Pour chaque élément annoté, dénommé 'centre', nous résolvons les annotations de ses descendants, ses frères (à savoir sur des bases géométriques et structurelles) et ses parents, si possible. Les descendants du 'centre' héritent leur fonction du parent, car ils appartiennent au même bloc d'interface utilisateur. Pour un élément qui ne possède pas de fonction, l'algorithme itère chacun de ses frères et il vérifie d'abord si le frère chevauche géométriquement un 'centre'. Si c'est le cas, l'élément obtient la même fonction du 'centre'. Si non, il obtient par défaut la fonction du premier 'centre' à sa gauche dans l'arbre. Notez que si un élément n'a pas de frère comme 'centre', l'algorithme se déplace vers le haut dans l'arbre pour régler d'abord la fonction de l'élément parent.

Une fois que tous les enfants du 'centre' obtiennent une fonction, le parent décide de sa propre fonction et adopte la fonction de ses enfants si elle est unique. Si les

enfants ont différents fonctions, le parent est partagé entre le maître et l'esclave.

La deuxième étape est la production des documents HTML du maître et de l'esclave. Pour produire l'interface principale, le système agit sur l'application principale et cache les éléments DOM 'interactives'. Le maitre est alors une version modifiée de l'application principale, où seulement les blocs maîtres sont affichés et où les éléments cachés servent comme raccourcis chaque fois que la logique de l'application principale nécessite la lecture ou la modification des éléments du composant esclave. Pour créer l'interface utilisateur de l'esclave, les éléments ayant une étiquette 'interactive' et ceux qui sont partagés dans l'application principale sont extraits et importés au nouveau composant, l'esclave, assurant ainsi la création de l'application multi-écran mais pas encore fonctionnel.

8.3.3.5 Résultats et Bilan

Applications	Precision	Rappel	Non-correspondance	
			Sur-segmentation	Absence de relation
Social pages	0.6	0.7	0.2	0.08
Video player pages	0.73	0.81	0.05	0.11
Semantic Video Application	0.71	0.83	0.07	0.035

Table 8.2: Evaluation de l'approche de segmentation

Dans cette section, nous évaluons l'approche de segmentation en la comparant à un Vérité Terrain (GT). La procédure d'évaluation est basée sur l'évaluation de deux paramètres: la cohérence visuelle des blocs et la justesse de la fonction attribuée à chaque bloc.

La Vérité Terrain a été créé manuellement, où des blocs cohérents étaient déterminés et affectés une fonction. Ensuite, nous avons comparé les résultats de notre segmentation à ceux du GT et nous fournissons les résultats de la comparaison dans le Tableau 8.2 sous forme de taux de précision et de rappel que nous définissons comme suit:

$$Precision = \frac{Nb\ of\ Matching\ Blocks}{Nb\ of\ Resulting\ Blocks} \tag{8.1}$$

$$Rappel = \frac{Nb\ of\ Matching\ Blocks}{Nb\ of\ GT\ Blocks} \tag{8.2}$$

Le Rappel est égal à un si l'algorithme de segmentation peut identifier correctement tous les blocs de la GT. La précision est égale à un si notre segmentation n'a pas produit un bloc 'non-correspondant'. La colonne 'Non-correspondance' indique le nombre moyen de blocs qui: 1) sont trop segmentés par l'algorithme de segmentation, i.e. quand un bloc dans le GT correspond à plusieurs blocs dans nos résultats, 2) ont aucune correspondance avec un bloc du GT ou s'ils ne sont pas correctement étiquetés.

Dans notre cas, la sur-segmentation ne présente pas un problème tant que tous les blocs résultants ont la même fonction. Seule la fonction du bloc affecte la distribution de l'interface utilisateur graphique, à savoir le pourcentage des blocs "Absence de relation" du tableau 8.2. En outre, cette catégorie couvre l'absence d'une fonction, ce qui ne présente pas un problème dans notre travail, surtout si elle est liée à du contenu qui ne soit ni multimédia, ni interactif.

En regardant le tableau 8.2, notre algorithme est fonctionnel puisque le pourcentage des blocs sans correspondance est entre 0.035 et 0.11, et la valeur de 0.11 pour les pages de vidéo player correspond à des blocs pour lesquels aucune étiquette est affectée.

Le tableau 8.2 montre également que les métriques calculées sont plus ou moins cohérentes pour les trois ensembles d'applications indépendamment de la hauteur de l'application ou du nombre des nœuds de DOM. Le taux de précision est le plus bas pour les applications sociales avec 0.61 par comparaison à celui du lecteur vidéo et à l'application VidéoSémantique avec respectivement 0.73 et 0.71.

Les applications de la catégorie du Player vidéo sont simples et la plupart du temps composées d'un élément vidéo, une barre de contrôle personnalisée et dans certains cas, des sous-titres. Les résultats de leurs segmentations montrent des valeurs élevées de Précision et de Rappel (0.73 et 0.81 respectivement). Ceci indique que l'algorithme est capable de séparer la barre de contrôle de la vidéo elle-même.

Pour l'application VideoSémantique, le taux de précision est 0.71 indiquant que la plupart des blocs du GT ont été identifiés par notre algorithme, même pour cette application complexe.

8.3.4 Adaptation de l'agencement des applications au contenu et aux appareils

La phase de Layout Refactoring adapte la mise en page de l'application multi-écran au contenu de l'application et à l'appareil sans changer l'arbre DOM. La mise en page de l'application pourra être assimilée à une grille. La mise en page détermine la répartition spatiale du contenu de l'application et les dimensions du contenu d'une page à l'aide des feuilles de style. Comme indiqué précédemment, travailler avec l'arbre DOM est difficile en particulier étant donné que pas tous ses nœuds sont visibles. Du fait, nous utilisons l'arbre logique qui contient des informations sur l'ordre de la lecture du contenu. Cependant, l'arbre logique est pas assez proche de l'arbre DOM, notamment les relations entre parents et enfants sont cassées et ces relations doivent être préservées pour appliquer correctement les styles CSS.

Comme décrit dans BoM [46], la construction de l'arbre logique produit également une structure intermédiaire appelée l'arbre géométrique, qui a une structure très proche de l'arbre DOM. Afin de préserver l'ordre de lecture et les relations parentales, nous utilisons les deux structures: arbres logiques et géométriques.

Cette phase consiste à déterminer la nouvelle configuration pour chacun des composants esclave et maître sous la forme de nouvelles feuilles de style CSS. Le composant maitre est souvent composé d'un seul bloc. Dans ce cas, l'optimisation du layout consiste simplement à étirer ce bloc, d'une manière similaire à la Full-Window API résultant du travail de W3C. Cependant, il y a des cas où le maître est composé de 2 blocs ou plus. Pour ces cas, nous avons développé un algorithme, appelé Full-Window Design (FWD). Par souci de concision, nous limitons la description de FWD à son objectif algorithme et ses principes comme suit. FWD vise à exploiter les espaces disponibles sur le maitre sur la base de la disposition des blocs et de leurs géométries. Ensuite, FWD repositionne ces blocs et les étire horizontalement et verticalement pour recouvrir les espaces vides.

Le composant esclave passe par le re-design réactif (RWD) décrit comme suit.

8.3.4.1 Responsive Web Re-design

Avec notre algorithme RWD, nous ne ciblons pas un dispositif spécifique mais nous visons la conception d'une mise en page, qui dynamiquement identifie la largeur de l'écran du dispositif sur lequel l'esclave s'execute, et sélectionne la disposition correspondante. Ceci est utile dans l'environnement multi-écran où les applications

peuvent se déplacer de manière transparente entre les dispositifs (par exemple, durant la migration des services). Les défis sont de produire une mise en page qui est (1) proche de la mise en page originale sur les grands et moyens appareils et qui (2) évite le défilement horizontal sur les petits ou extra-petits appareils et cela(3) sans casser l'ordre de lecture du contenu. Ces deux mise en page devraient également (4) respecter le dimensionnement relatif entre tous les blocs (5) ainsi que leurs ratio largeur/hauteur. Enfin, suite à un changement de taille de la fenêtre, (6), la mise en page devrait s'adapter dynamiquement. Le Responsive Web Design (RWD) est la réponse à nos besoins (1), (2) et (6) puisqu'il utilise des 'Media Queries' pour détecter des changements et pour affecter différente mise en page pour chaque gamme de largeurs de la fenêtre du dispositif. Finalement, il conçoit dynamiquement une mise en page à la base d'un système de grille flexible.

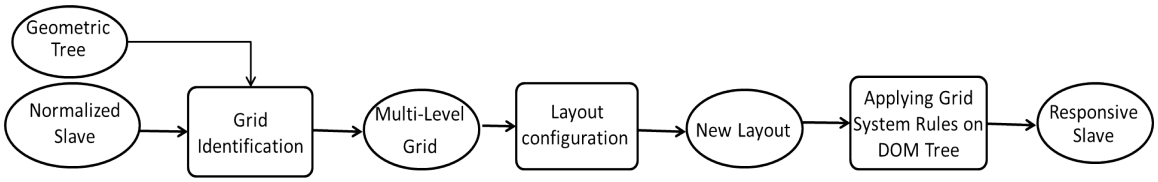
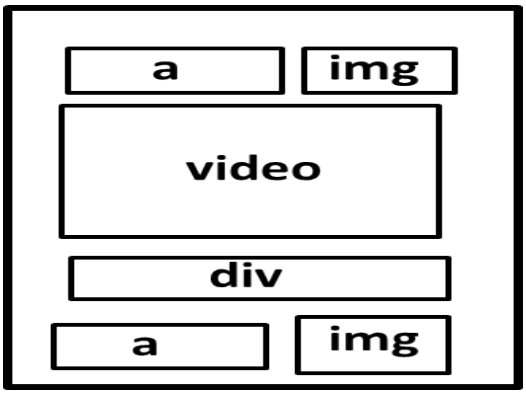


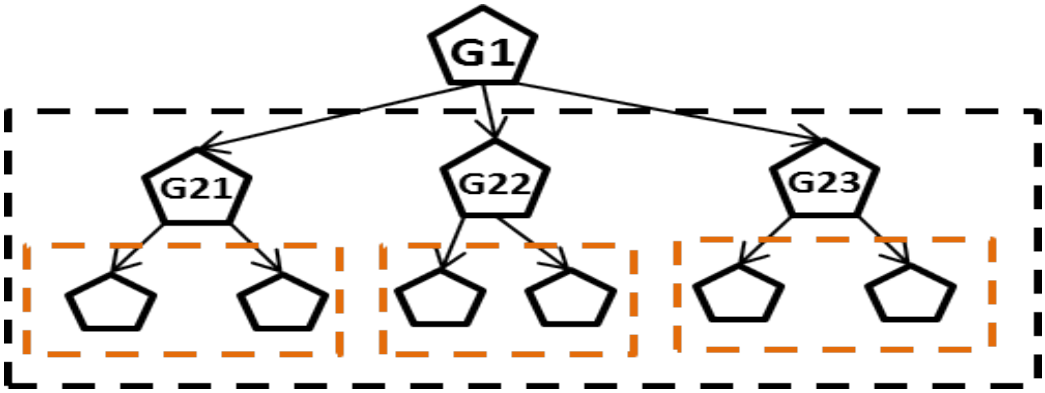
Figure 8-3: Blocs de bases pour le responsive design sur le composant Esclave

Pour rendre l'esclave responsif, nous avons choisi la bibliothèque Twitter Bootstrap [53] pour la conception des mises en pages responsives. En utilisant Bootstrap, la largeur et la position d'un élément sont calculées par rapport à un élément conteneur (en pourcentage) tout en respectant le dimensionnement relatif (4) et les ratio largeur/hauteur de chaque élément (5). En imitant le travail d'un développeur Bootstrap, notre algorithme consiste en trois phases comme indiqué dans la Figure 8-3 et comme suit: Identification de grille, Configuration de la mise en page et Application de la mise en page sur l'arbre DOM. Plus de détails sont donnés ci-dessous.

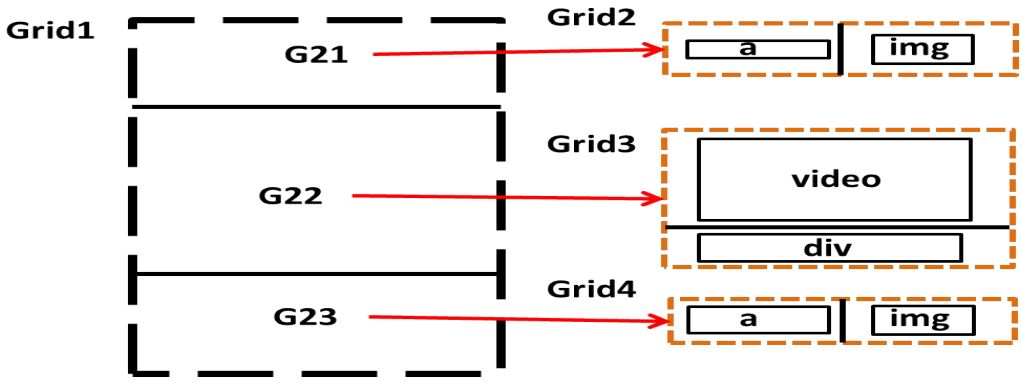
Identification de la grille Le problème d'identification de grille consiste à identifier une grille multi-niveaux qui respecte le modèle de grille de Bootstrap. Figure 8-4(a) représente un arbre DOM et Figure 8-4(b) montre l'arbre géométrique correspondant. Cet arbre géométrique comporte deux niveaux: 3 éléments (à savoir, G21, G22 et G23) au premier niveau et 6 éléments feuille dans le second. Comme nous pouvons le voir, les niveaux de grille correspondent exactement aux niveaux de l'arbre DOM. De même, la grille associée doit avoir deux niveaux. L'algorithme commence à partir de la racine de l'arbre géométrique. Si la racine a des enfants, la racine est alors considérée comme la tête d'une grille de premier niveau et ses enfants sont les



(a)



(b)



(c)

Figure 8-4: (a) Mise en page initiale de l'application (b) Arbre géométrique associé (b) Identification des séparateurs de la grille

éléments de cette grille. La tête de la grille correspond à l'élément géométrique regroupant l'ensemble des blocs d'une grille. Itérativement, la géométrie des enfants est ensuite traitée pour vérifier l'alignement des enfants et cela dans le but d'identifier les séparateurs horizontaux et verticaux et de situer tous les enfants dans la grille. Une cellule contient au plus un élément du même niveau de hiérarchie. Il pourrait y avoir quelques cellules vides. Dans notre exemple, le premier niveau se compose d'une seule grille (Grid1) qui contient trois cellules (G21, G22 et G23) séparées horizontalement et G1 est considéré comme la tête de la grille. L'algorithme itère sur les sous-arbres des nœuds enfants jusqu'à ce qu'il atteigne les feuilles pour former une grille à N niveaux où N est le nombre de niveaux hiérarchique dans l'arbre géométrique. Par exemple, G21 (resp. G22, G23) a deux descendants dans l'arbre géométrique, donc elle forme une grille de second niveau (resp. Grid2, Grid3, Grid4) contenant deux cellules.

Configuration et application de la mise en page Similaire à Bootstrap, au cours de la phase de conception nous faisons abstraction de la valeur réelle de la largeur de l'écran puisque nous ciblons une gamme de tailles d'écran. Nous supposons que chaque grille, à tout niveau hiérarchique, consiste en une seule ligne et sa largeur totale est décomposée en M colonnes effectives ayant des largeurs égaux. Cette phase d'abstraction à laquelle nous référons comme la normalisation, exprime la largeur du bloc et sa position par rapport aux dimensions de la tête de la grille. Ainsi, une modification de la largeur/position de la tête déclenche un changement similaire à ses descendants. Le point clé ici est d'exprimer les dimensions des cellules de la grille en termes de nombre de colonnes effectives comme expliqué ci-dessous.

Pour les dispositifs à grands écrans L'algorithme commence à traiter la première grille et ensuite itère sur les sous-grilles. A noter qu'une sous-grille fait référence à une grille de niveau L où L est un nombre entier se référant au niveau de la hiérarchie. Une étape cruciale ici est de déterminer pour chaque grille ou sous-grille la largeur totale (en px) que les blocs peuvent couvrir, nous l'appelons la largeur de référence. La largeur de référence pour le premier niveau de la grille est la largeur de la fenêtre, tandis que la largeur de référence pour une grille de niveau L est la largeur de sa tête. Pour chaque grille de niveau L, l'algorithme itère sur ses cellules et calcule, en termes de nombre de colonnes effectives: 1) sa largeur par rapport à la largeur référence, 2) son offset gauche (défini par rapport à la distance qui le sépare du bloc se trouvant à sa gauche). Les valeurs calculées sont immédiatement appliquées à l'élément DOM correspondant à chaque élément géométrique. L'algorithme passe

itérativement à la grille de niveau (L + 1) et se poursuit jusqu'à ce qu'il atteint la grille niveau N.

Pour les petits dispositifs La même logique est appliquée pour configurer la mise en page en une seule colonne pour les petits dispositifs. La largeur de référence de la grille au niveau de la première grille est la largeur de la fenêtre du dispositif. Nous considérons que, dans n'importe quel grille de niveau L, toutes les cellules ont une largeur égale à la largeur de la tête, à savoir, M Colonnes effectives. En outre, nous éliminons l'offset-gauche pour éliminer les espaces vides, étant donné que la taille de la fenêtre pour petit appareil est très petite).

8.3.4.2 Evaluation de l'Algorithme RWD

Pour l'algorithme RWD, l'évaluation de la présentation est limitée à tester la présence et la quantité du défilement horizontal (HS) sur les petits appareils. Le principe consiste à comparer les dimensions des éléments géométriques à la dimension de la fenêtre. Notre objectif est d'identifier le nombre d'éléments géométriques qui sont à l'origine du défilement horizontal en suivant la règle suivante.

Table 8.3: Défilement horizontal sur les applications esclave avant et après RWD

Applications	Block causant du défilement		Etendue du défilement	
	Avant RWD	Après RWD	Avant RWD	Après RWD
Video-semantic	107	0	394	0
Vimeopro	64	0	348	0
Twitch	268	20	381	97
Liveleak	308	0	360	0
Ustream	14	0	38	0
Break	1157	2	408	283

Un élément géométrique provoque du défilement horizontal si l'un de ses abscisses, à savoir, la position gauche et la position droite, est en dehors de la fenêtre du dispositif. Nous définissons la quantité de HS comme la distance moyenne entre la fenêtre du navigateur et les blocs provoquant le défilement horizontal.

Tableau 8.3 indique le nombre de blocs provoquant HS ainsi que la quantité de HS calculé avant et après l'application de notre algorithme. Ce tableau montre qu'avant d'appliquer notre algorithme, toutes les applications esclaves provoquaient HS sur

le petit dispositif. La quantité minimale de HS avant d'appliquer notre algorithme est de 38 px et elle correspond à la page Ustream non-RWD qui comporte 14 blocs responsables du défilement. La quantité de HS pour les autres applications non-RWD varie entre 348 px et 408 px. Relativement à la largeur de notre petit appareil (à savoir, 412px), ces valeurs sont grandes. Après l'application de notre algorithme, le HS a été complètement éliminé pour VideoSemantique, VimeoPro, Liveleak et Ustream. En revanche, la quantité de HS était réduite de 381 px à 97 px (à savoir, 75% de réduction) pour Twitch et de 408 px à 283 px pour Break (à savoir, 30% de réduction). Tableau 8.3 montre également que notre algorithme a éliminé le HS de 92% des blocs de Twitch, et il a éliminé le HS de 99% des blocs de Break.

8.3.5 Distribution de l'état des applications et synchronisation

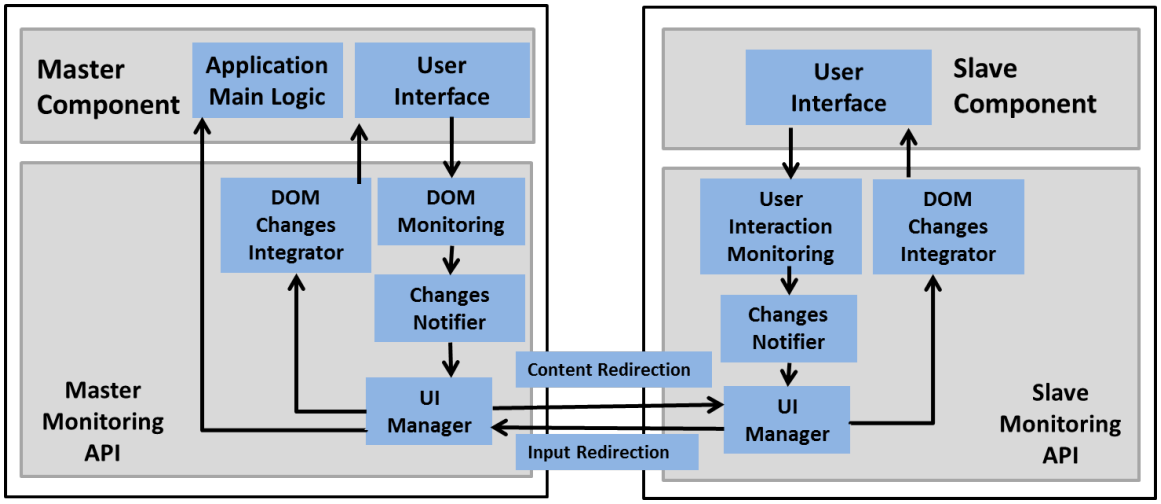


Figure 8-5: Les caractéristiques des composant maitre et esclave

La solution de la distribution que nous avons proposé consiste à adapter chaque composant à l'environnement multi-écrans et à fournir un composant esclave sans logique. Les événements survenus coté esclave doivent être redirigés vers le composant maître et les événements coté maître (événements liés à la vidéo) doivent être renvoyés à l'esclave une fois détectés. Les défis principaux sont à capturer et à caractériser continuellement ces changements dynamiques et de les envoyer, si nécessaire, vers le composant correspondant sans affecter la performance globale de l'application.

8.3.5.1 Adaptation de l'application multi-écran pour la distribution de l'État

Chacun des composants maître et esclave est enrichi par quatre caractéristiques, à savoir, DOM/UI Monitoring, Modification Notifier, UI Manager et DOM Changes Integrator, comme représenté sur la Figure 8-5.

DOM Monitoring écoute et capture les changements qui se produisent sur l'arbre DOM du maître et les transmet ensuite aux DOM Changes Notifier qui est responsable de la caractérisation du changement. UIManager agit comme une passerelle pour assurer la communication entre les deux composants. Coté maître, UIManager vérifie si le changement concerne l'esclave et il l'envoie si c'est le cas. Nous appelons cela 'Redirection de contenu'. Une fois reçue par le UIManager de l'esclave, le changement est envoyé directement aux DOM Changes Integrator qui est responsable de l'intégration de ces changements dans l'arbre DOM.

Coté esclave, UI Monitoring écoute les interactions de l'utilisateur. Sur une interaction, l'interaction est envoyé aux Changes Notifier, puis au UIManager qui redirige le changement vers le UIManager du maître. Nous appelons cela 'Redirection de requête'. Une fois reçu du côté du maître, UIManager détermine si le changement contient des entrées de l'utilisateur et les envoie au DOM Changes Integrator. Dans le cas contraire, si le changement demande l'exécution d'une certaine logique, alors UIManager déclenche cette logique.

8.3.5.2 Répartition de l'Etat pendant l'exécution

La distribution de l'État est une phase d'exécution qui est basée sur une technique de mise en miroir. Durant la phase de la distribution de interface utilisateur, certains contenus sont doublés entre le maître et l'esclave. La technique de mise en miroir assure en permanence que l'esclave a un arbre DOM qui est un miroir exact de l'arbre DOM caché sur le maître et elle assure que les interactions de l'utilisateur sur l'application esclave sont transmises au maître.

Sur le maître, tout changement dynamique affectant les éléments de l'esclave, par exemple, modification, suppression ou création des nœuds, est d'abord capturé et analysé. Ensuite, les modifications sont envoyées au composant esclave par le biais des messages de changement. Sur la réception d'un message, l'esclave met à jour son arbre DOM et intègre ce changement. Coté esclave, nous devons rediriger les entrées de l'utilisateur et ses requêtes vers le maître où la logique d'application réside. Pour

cela, l'esclave écoute les interactions des utilisateurs et envoie les entrées utilisateur au maître.

8.3.5.3 Expérimentation et Résultats

Pour tester les performances lors du runtime, nous avons déployé notre système dans le navigateur Chrome d'un PC que nous considérons comme un dispositif maître. Nous utilisons un autre PC comme dispositif esclave. Les deux appareils sont connectés au réseau LAN e ils communiquent à travers la plate-forme de COLTRAM.

Nous lançons l'application VideoSemantic sur le dispositif maître et laissons notre système distribuer l'application entre les composants maître et esclave. Par conséquence, le maître contient uniquement l'élément vidéo avec les sous-titres, qui occupent la fenêtre complète. L'esclave contient le reste des éléments, notamment l'en-tête, le pied de page, les sections Wikipédia et Googlemap, etc.

Résultats durant le runtime Nous avons lançé la vidéo sur le maître et nous avons inspecté le nombre et le type des mises-à-jour dans l'arbre DOM qui ont été redirigés vers l'esclave. Et puis, nous avons mesuré les délais de retard qui ont été ajoutés par notre système, y compris ceux en relation avec le plate-forme multi-écran pour vérifier si notre système maintient la fonctionnalité globale de l'application.

Les résultats de l'inspection sont les suivantes: 27 mises à jour ont été reçues pour enlever des éléments DOM de l'arbre DOM de l'esclave. 1001 mises à jour ont été reçues pour ajouter ou déplacer un élément DOM. 555 mises à jour ont été reçues pour modifier, ajouter ou supprimer un attribut HTML. La communication de ces mises-à-jour par le maître à l'esclave exige 293 messages. Chaque message contient en moyenne 5 mises à jour.

Nous avons ensuite mesuré le délai global de notre système. Nous avons évalué le RTT en relation avec la plate-forme COLTRAM et avec la couche physique, en envoyant des messages de ping ne contenant que des valeurs de temps. 6000 messages ping ont été échangés entre le maître et l'esclave. RTT varie entre 6 ms et 41 ms, en moyenne 8 ms entre deux machines connectées via Ethernet et en utilisant un service Bonjour. Ainsi, la communication unidirectionnelle induit un délai qui varie entre 3 ms et 21 ms, en moyenne 4 ms.

Ensuite, nous avons lancé l'application videoSemantic 10 fois pour calculer le retard global du système sur les 295 * 10 messages de changement reçus sur l'esclave.

Le retard global du système varie entre 5 ms et 34 ms, en moyenne 11 ms. L'écart-type calculé est de 4.8. 65% des messages produisent un retard inférieur à 11 ms et 95% ne dépassent pas le délai de 20 ms. Ayant le retard global du système et le retard unidirectionnel, le délai de traitement relié seulement à la surveillance des changements de l'arbre DOM sur le maître et à l'intégration des changements sur l'esclave varie entre 2 ms et 13 ms. En se basant sur les travaux de Miller [37], un délai inférieur à 100 ms n'est pas perceptible par l'œil humain. Le retard du système global, y compris la communication, même à sa valeur maximale, ne dépasse pas les 34 ms. Ainsi, notre système assure une synchronisation satisfaisante aux utilisateurs.

Complexité du système et Bilan Les différents parties du système fonctionnent sur trois différentes arbre: arbre DOM, arbre géométrique et l'arbre logique. L'arbre géométrique qui est une structure intermédiaire entre arbre DOM et arbre logique. Dans notre dataset, en moyenne, un arbre géométrique contient 73% du nombre de nœuds DOM et 94% du nombre de niveaux hiérarchiques dans l'arbre DOM. Un arbre logique contient seulement 14% du nombre de nœuds DOM et 43,5% du nombre de niveaux hiérarchiques. Le processus de simplification diminue la complexité du traitement de certains des parties du système.

Notre système nécessite 5 traversées de l'arbre DOM, 2 traversées de l'arbre géométrique et 4 traversées de l'arbre logique pour produire les composants maître et esclave. Toutes ces traversées ont une complexité de calcul de O(n). Au cours de l'exécution, une fois le maître et l'esclave commencent à communiquer les changements (distribution de l'État), Mutation-Observer traverse l'arbre DOM une seule fois pour identifier les changements dans l'arbre DOM. La Mutation-Summary ne parcourt pas l'arbre DOM, mais seulement elle analyse les records du Mutation-Observer. La division et la distribution du contenu durant le runtime nécessitent une analyse locale des nœuds autour d'un nouveau noeud (parent, nœuds voisins). L'algorithme de distribution de l'État, avec une complexité en O(n), ne pénalise pas la fonctionnalité de l'application multi-écran pendant l'exécution. Cette analyse est validée dans la section précédente, où le retard du système varie entre 2 ms et 13 ms et notre système est aperçu comme instantanée.

8.4 Conclusion

Dans cette thèse, nous avons abordé la réutilisation des applications Web vidéo-centrique pour la création automatique d'applications distribuées dans le contexte des applications multi-écran.

La littérature sur ce sujet se concentre sur la conception d'un modèle d'application, sur la création d'une plate-forme adaptée aux exigences de l'application et sur son fonctionnement après la distribution. Mais les œuvres de la littérature ne sont pas concluantes sur la distribution de contenu, étant donné que dans la plupart des cas, une distribution manuelle est adoptée. Une autre lacune a été identifiée et elle est liée à l'adaptation de la mise en page après la distribution. Cette thèse a cherché à couvrir ces deux lacunes.

Le travail a été effectué comme suit:

- La conception d'un modèle pour les applications distribuées
- La caractérisation et l'exploitation de l'environnement multi-écran pour la distribution de l'interface utilisateur
- L'adaptation de l'application distribuée à l'environnement multi-écran, y compris à la multitude de dispositifs et de la plate-forme multi-écran
- Le développement et l'évaluation du système de refactoring qui intègre les trois points ci-dessus.

Parmi les résultats, nous avons montré que les applications mono-écrans peuvent être réutilisées pour créer des applications multi-écrans fonctionnelles. Les points clés sont: la division du contenu de l'application et la conception d'un modèle pour les applications multi-écrans, tout en optimisant les dommages sur les documents de l'application.

En ce qui concerne le premier point ci-dessus, nous avons montré que notre approche de segmentation hybride non seulement identifie des blocs cohérents, mais elle sépare également le contenu interactif du contenu multimédia. Ainsi, elle permet la distribution du meilleur contenu sur le meilleur périphérique.

Dans ce travail, la distribution a été limitée à l'arbre DOM tout en gardant la logique de l'application sur un côté de l'application distribuée. En conséquence, il en résulte un modèle particulier pour les applications multi-écran constitués d'un

composant maître et un composant esclave correspondant. Ces composants sont en mesure d'exécuter deux opérations cross-device: la redirection et la synchronisation, en utilisant une technique de contrôle pour l'arbre DOM seulement. Les résultats montrent que la surveillance de l'arbre DOM était suffisante pour assurer la distribution de l'état et la cohérence de l'état spécialement que pas d'incohérence visuelle a été signalé lors de nos tests.

Après la distribution du contenu, nous avons également couvert l'adaptation dynamique de la mise en page des composants maitre et esclave pour résoudre deux principales anomalies dans la mise en page: les espaces vides et le défilement horizontal. Pour éliminer les espaces vides du composant maitre, nous avons négligé la structure de l'arbre DOM et nous avons imposé un nouveau design calculé par rapport aux dimensions de la fenêtre. Pour éliminer le défilement horizontal du composant esclave, la structure DOM était obligatoirement respectée. L'évaluation de ces deux algorithmes, après la quantification de chacune des anomalies, a révélé: (1) l'absence d'espaces vides sur les composants de base, (2) l'absence ou la réduction du défilement horizontal sur les composants esclaves. Tous les deux algorithmes sont relativement basiques mais résolvent les problèmes de mise en page créés par notre modèle d'application ou par la nature dynamique de l'environnement multi-écran. Les deux algorithmes sont ouverts à des améliorations. Les résultats obtenus après l'évaluation de la performance globale du système, indiquent que notre système est réaliste avec un retard maximum de 34 ms, en moyenne 11 ms sur Ethernet, lors de l'execution. Ce retard qui est petit comparé au 100 ms défini par Miller [37] montre que le temps de réponse de notre système est suffisant pour conserver l'interactivité. En outre, le système est compatible avec tous les environnements testés pour le développement d'applications web. Ceci contribue à élargir son application à la plupart des applications web.

Plusieurs questions importantes n'ont pas été étudiées au cours de cette thèse en raison des objectifs précis de la thèse dans le domaine de Document Engineering. Ces questions comprennent:

- L'extension du système à d'autres langages basés sur XML et différents de HTML, comme SMIL et NCL. Cela nécessite l'étude de la nature et le type de ces applications pour vérifier 1) s'il existe un moyen d'analyser chaque élément afin de déterminer sa fonction et ses paramètres géométriques, et 2) s'il y a une interface de programmation d'exécution qui permet l'interfaçage du code de surveillance du contenu.

- L'extension de l'exploitation de l'environnement afin d'inclure non seulement les caractéristiques de l'appareil, mais aussi les préférences de l'utilisateur. Cela nécessite la réalisation d'une étude utilisateur pour enquêter sur les préférences de l'utilisateur dans des multiples scénarios.
- Une étude de faisabilité pourra être menée pour évaluer qualitativement 1) la refonte de la mise en page de chacun du composant maître et esclave, et 2) les résultats de la distribution de contenu par notre système.

List of Publications

Journal papers

[J.1] Mira Sarkis, Cyril Concolato, and Jean-Claude Dufourd, *A multiscreen Refactoring System for Video-centric Web Applications*, Multimedia Tools and Applications. MTAP, Submitted in April 2016, Revised submission in August.

Conference papers

[C.1] Mira Sarkis, Cyril Concolato, and Jean-Claude Dufourd. 2014. *The virtual splitter: refactoring web applications for themultiscreen environment.* In Proceedings of the 2014 ACM symposium on Document engineering (DocEng '14). ACM, New York, NY, USA, 139-142. DOI=http://dx.doi.org/10.1145/2644866.2644893

[C.2] Mira Sarkis, Cyril Concolato, and Jean-Claude Dufourd. 2015. *MSoS: A Multi-Screen-Oriented Web Page Segmentation Approach.* In Proceedings of the 2015 ACM Symposium on Document Engineering (DocEng '15). ACM, New York, NY, USA, 85-88. DOI=http://dx.doi.org/10.1145/2682571.2797090

Bibliography

[1] W3C recommendation (2010) mobile web application best practices. `http://www.w3.org/TR/css3-mediaqueries/`.

[2] J. Allard, V. Chinta, S. Gundala, and G. G. Richard III. Jini meets upnp: An architecture for jini/upnp interoperability. In *Proceedings of the 2003 Symposium on Applications and the Internet*, SAINT '03, pages 268–, Washington, DC, USA, 2003. IEEE Computer Society.

[3] Sriram Karthik Badam and Niklas Elmqvist. Polychrome: A cross-device framework for collaborative web visualization. In *Proceedings of the Ninth ACM International Conference on Interactive Tabletops and Surfaces*, ITS '14, pages 109–118, New York, NY, USA, 2014. ACM.

[4] 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.

[5] Federico Bellucci, Giuseppe Ghiani, Fabio Paternò, and Carmen Santoro. Engineering javascript state persistence of web applications migrating across multiple devices. In *Proceedings of the 3rd ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '11, pages 105–110, New York, NY, USA, 2011. ACM.

[6] Regina Bernhaupt, Marianna Obrist, Astrid Weiss, Elke Beck, and Manfred Tscheligi. Trends in the living room and beyond: Results from ethnographic studies using creative and playful probing. *Comput. Entertain.*, 6(1):5:1–5:23, May 2008.

[7] Martí Bosch, Pierre Genevès, and Nabil Layaïida. Automated refactoring for size reduction of css style sheets. In *Proceedings of the 2014 ACM Symposium on*

Document Engineering, DocEng '14, pages 13–16, New York, NY, USA, 2014. ACM.

[8] D Cai, S Yu, JR Wen, and WY Ma. Vips: A vision-based page segmentation algorithm. Technical report, Microsoft, MSR-TR-2003-79, 2003.

[9] Deng Cai, Shipeng Yu, Ji-Rong Wen, and Wei-Ying Ma. Extracting content structure for web pages based on visual representation. In *Proceedings of the 5th Asia-Pacific Web Conference on Web Technologies and Applications*, APWeb'03, pages 406–417, Berlin, Heidelberg, 2003. Springer-Verlag.

[10] Pablo Cesar and Konstantinos Chorianopoulos. The evolution of tv systems, content, and users toward interactivity. *Foundations and Trends in Human Computer Interaction*, 2(4):279–373, 2009.

[11] Soumen Chakrabarti. Integrating the document object model with hyperlinks for enhanced topic distillation and information extraction. In *Proceedings of the 10th International Conference on World Wide Web*, WWW '01, pages 211–220, New York, NY, USA, 2001. ACM.

[12] J Chen, B Zhou, J Shi, H Zhang, and Q Fengwu. Function-based object model towards website adaptation. In *Proceedings of the 10th International Conference on World Wide Web*, WWW '01, pages 587–596, New York, NY, USA, 2001. ACM.

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

[14] Zlatko Čović, Miodrag Ivković, and Biljana Radulović. Mobile detection algorithm in mobile device detection and content adaptation. *Acta Polytechnica Hungarica*, 9(2):95–113, 2012.

[15] N Elmqvst. Distributed user interfaces: State of the art. In *ACM Press (2011), DUI*, pages 7–11, 2011.

[16] P Faraday. Visually critiquing web pages. In *Multimedia 99*, pages 155–166. Springer, 2000.

[17] B Frain. *Responsive web design with HTML5 and CSS3*. Packt Publishing Ltd, 2012.

[18] A Garrard. Multi-display desktops and the case for more pixels. In J. Chen, W. Cranton, and M. Fihn, editors, *Handbook of Visual Display Technology*, pages 2571–2579. Springer Berlin Heidelberg, 2012.

[19] J Gentle. Sharejs - live concurrent editing in your app, 2012. `http://sharejs.org/`.

[20] G Ghiani, J Polet, V Antila, and J Mäntyjärvi. Evaluating context-aware user interface migration in multi-device environments. *Journal of Ambient Intelligence and Humanized Computing*, 6(2):259–277, 2013.

[21] Giuseppe Ghiani, Fabio Paternò, and Carmen Santoro. On-demand cross-device interface components migration. In *Proceedings of the 12th International Conference on Human Computer Interaction with Mobile Devices and Services*, MobileHCI '10, pages 299–308, New York, NY, USA, 2010. ACM.

[22] GitHub. Coltram: Collaborative transmedia services for home networks. accessed:06 April 2014. [Online].Available:`http://github.com/COLTRAM`.

[23] Google. The multi-screen world study. `https://www.thinkwithgoogle.com/research-studies/the-new-multi-screen-world-study.html`.

[24] S Gupta, G Kaiser, D Neistadtand, and P Grimm. Dom-based content extraction of html documents. In *Proceedings of the 12th International Conference on World Wide Web*, WWW '03, pages 207–214, New York, NY, 2003. ACM.

[25] 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, USA, December 2000.

[26] B Hartmann, M Beaudouin-Lafon, and WE Mackay. Hydrascope: Creating multi-surface meta-applications through view synchronization and input multiplexing. In *Proceedings of the 2Nd ACM International Symposium on Pervasive Displays*, PerDis '13, pages 43–48, New York, NY, USA, 2013. ACM.

[27] M Heinrich, FJ Grüneberger, T Springer, and M Gaedke. Exploiting annotations for the rapid development of collaborative web applications. In *Proceedings of the 22nd international conference on World Wide Web*, pages 551–560. International World Wide Web Conferences Steering Committee, 2013.

[28] Jer Lang Hong, Eu-Gene Siew, and Simon Egerton. Information extraction for search engines using fast heuristic techniques. *Data and Knowledge Engineering*, 69(2):169–196, 2010.

[29] Simon Holm Jensen, Magnus Madsen, and Anders Møller. Modeling the html dom and browser api in static analysis of javascript web applications. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 59–69, New York, NY, USA, 2011. ACM.

[30] C Kohlschütter and W Nejdl. A densitometric approach to web page segmentation. In *Proceedings of the 17th ACM conference on Information and knowledge management*, pages 1173–1182. ACM, 2008.

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

[32] Can Liu, Olivier Chapuis, Michel Beaudouin-Lafon, Eric Lecolinet, and Wendy E. Mackay. Effects of display size and navigation type on a classification task. In *Proceedings of the 32Nd Annual ACM Conference on Human Factors in Computing Systems*, CHI '14, pages 4147–4156, New York, NY, USA, 2014. ACM.

[33] James Teng Kin Lo, Eric Wohlstadter, and Ali Mesbah. Imagen: Runtime migration of browser sessions for javascript web applications. In *Proceedings of the 22Nd International Conference on World Wide Web*, WWW '13, pages 815–826, New York, NY, USA, 2013. ACM.

[34] Jianli Luo, Jie Shen, and Cuihua Xie. Segmenting the web document with document object model. In *Services Computing, 2004. (SCC 2004). Proceedings. 2004 IEEE International Conference on*, pages 449–452, Sept 2004.

[35] Sergio Maffeis, John C. Mitchell, and Ankur Taly. An operational semantics for javascript. In *Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, APLAS '08, pages 307–325, Berlin, Heidelberg, 2008. Springer-Verlag.

[36] Jérémie Melchior, Donatien Grolaux, Jean Vanderdonckt, and Peter Van Roy. A toolkit for peer-to-peer distributed user interfaces: Concepts, implementation,

and applications. In *Proceedings of the 1st ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '09, pages 69–78, New York, NY, USA, 2009. ACM.

[37] Robert B. Miller. Response time in man-computer conversational transactions. In *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*, AFIPS '68 (Fall, part I), pages 267–277, New York, NY, USA, 1968. ACM.

[38] Mozilla. Semantic video with popcorn.js, 2011. accessed 22 May 2014. [Online] . Available:`http://popcornjs.org/demo/semantic-video`.

[39] M Nebeling, F Matulic, and M Norrie. Metrics for the evaluation of news site content layout in large-screen contexts. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '11, pages 1511–1520. ACM, 2011.

[40] M Nebeling, M Speicher, and M Norrie. Crowdadapt: Enabling crowdsourced web page adaptation for individual viewing conditions and preferences. In *Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '13, pages 23–32, New York, NY, USA, 2013. ACM.

[41] C. K. Nguyen, L. Likforman-Sulem, J. C. Moissinac, C. Faure, and J. Lardon. Web document analysis based on visual segmentation and page rendering. In *Document Analysis Systems (DAS), 2012 10th IAPR International Workshop*, pages 354–358, March 2012.

[42] D Pelli, N Majaj, N Raizman, C Christian, E Kim, and M Palomares. Grouping in object recognition: The role of a gestalt law in letter identification. *Cognitive Neuropsychology*, 26(1):36–49, 2009.

[43] A Pnueli, R Bergman, S Schein, and O Barkol. Web page layout via visual segmentation. *HP Laboratories*, 2009.

[44] Dennis Quan, David Huynh, David R. Karger, and Robert Miller. User interface continuations. In *Proceedings of the 16th Annual ACM Symposium on User Interface Software and Technology*, UIST '03, pages 145–148, New York, NY, USA, 2003. ACM.

[45] M. R. Rahman and S. Akhter. Real time bi-directional traffic management support system with gps and websocket. In *Computer and Information Technology;*

Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing (CIT/IUCC/DASC/PICOM), 2015 IEEE International Conference on, pages 959–964, Oct 2015.

[46] A Sanoja and S Gançarski. Block-o-matic: A web page segmentation framework. In *Multimedia Computing and Systems (ICMCS), 2014 International Conference on*, pages 595–600. IEEE, 2014.

[47] M Sarkis, C Concolato, and JC Dufourd. The virtual splitter: Refactoring web applications for the multiscreen environment. In *Proceedings of the 2014 ACM Symposium on Document Engineering*, DocEng '14, pages 139–142. ACM, 2014.

[48] M Sarkis, C Concolato, and JC Dufourd. Msos: A multi-screen-oriented web page segmentation approach. In *Proceedings of the 2015 ACM Symposium on Document Engineering*, DocEng '15, pages 85–88. ACM, 2015.

[49] W Seok. A framework proposal of ux evaluation of the contents consistency on multi screens. In Constantine Stephanidis, editor, *HCI International 2015 - Posters Extended Abstracts*, volume 528 of *Communications in Computer and Information Science*, pages 69–73. Springer International Publishing, 2015.

[50] Ruihua Song, Haifeng Liu, Ji-Rong Wen, and Wei-Ying Ma. Learning block importance models for web pages. In *Proceedings of the 13th International Conference on World Wide Web*, WWW04, pages 203–211, New York, NY, USA, 2004. ACM.

[51] C Sun, S Xia, D Sun, D Chen, H Shen, and W Cai. Transparent adaptation of single-user applications for multi-user real-time collaboration. *ACM Trans. Comput.-Hum. Interact.*, 13(4):531–582, December 2006.

[52] P Tarasewich. An investigation into web site design complexity and usability metrics. *Quarterly Journal of Electronic Commerce*, 2008. Northeastern University.

[53] Twitter. Bootstrap framework for responsive web design. `twitter.github.com/bootstrap`.

[54] S Vadrevu, F Gelgi, and H Davulcu. Semantic partitioning of web pages. In *Web Information Systems Engineering–WISE 2005*, pages 107–118. Springer, 2005.

[55] C. N. Ververidis and G. C. Polyzos. Service discovery for mobile ad hoc networks: a survey of issues and techniques. *IEEE Communications Surveys Tutorials*, 10(3):30–45, Third 2008.

[56] R Weinstein, A Klein, M Kearney, E Delgado, A Komorosko, E Bidelman, and P Irish. Mutation-summary, 2013. accessed 21 February 2014. [Online]. Available: `http://code.google.com/p/mutation-summary/`.

[57] J Yang and D Wigdor. Panelrama: Enabling easy specification of cross-device web applications. In *Proceedings of the 32Nd Annual ACM Conference on Human Factors in Computing Systems*, CHI '14, pages 2783–2792, New York, NY, USA, 2014. ACM.

[58] Yudong Yang and HongJiang Zhang. Html page analysis based on visual cues. In *Document Analysis and Recognition, 2001. Proceedings. Sixth International Conference on*, pages 859–864, 2001.

[59] Dongsong Zhang. Web content adaptation for mobile handheld devices. *Commun. ACM*, 50(2):75–79, February 2007.

[60] M Zorrilla, N Borch, F Daoust, A Erk, J Flórez, and A Lafuente. A web-based distributed architecture for multi-device adaptation in media applications. *Personal and Ubiquitous Computing*, 19(5-6):803–820, 2015.

[61] M Zorrilla, I Tamayo, A Martin, and A Dominguez. User interface adaptation for multi-device web-based media applications. In *Broadband Multimedia Systems and Broadcasting (BMSB), 2015 IEEE International Symposium*, pages 1–7, June 2015.

[62] Zurb. Foundation framework for responsive web design. `http://foundation.zurb.com/`.