A Custom Browser Architecture to Execute Web Navigation Sequences

José Losada, Juan Raposo, Alberto Pan, Paula Montoto and Manuel Álvarez

Information and Communications Technology Department, University of A Coruña
Facultad de Informática, Campus de Elviña, s/n, 15071, A Coruña (Spain)

{jlosada, jrs, apan, pmontoto, mad}@udc.es

Abstract. Web automation applications are widely used for different purposes such as B2B integration and automated testing of web applications. Most current systems build the automatic web navigation component by using the APIs of conventional browsers. This approach suffers performance problems for intensive web automation tasks which require real time responses and/or a high degree of parallelism. Other systems use the approach of creating custom browsers to avoid some of the tasks of conventional browsers, but they work like them, when building the internal representation of the web pages. In this paper, we present a complete architecture for a custom browser able to efficiently execute web navigation sequences. The proposed architecture supports some novel automatic optimization techniques that can be applied when loading and building the internal representation of the pages. The tests performed using real web sources show that the reference implementation of the proposed architecture runs significantly faster than other navigation components.

Keywords: Web Automation, Optimization, Browser Architecture.

1 Introduction

Most today's web sources do not provide suitable interfaces for software programs. That is why a growing interest has arisen in so-called web automation applications that are able to automatically navigate through websites simulating the behavior of a human user. Web automation applications are widely used for different purposes such as B2B integration, web mashups, automated testing of web applications, Internet meta-search or business watch. For example, a technology watch application can use web automation to automatically search in the different websites and daily retrieve new patents and articles of a predefined area of knowledge.

A crucial part of web automation technologies is the ability to execute automatic web navigation sequences. An automatic web navigation sequence consists in a sequence of steps representing the actions to be performed by a human user over a web browser to reach a target web page. Figure 1 illustrates an example of a web navigation sequence that retrieves the list of patents matching the search term "*World Wide Web*" in the European Patent Office website (*www.epo.org*).

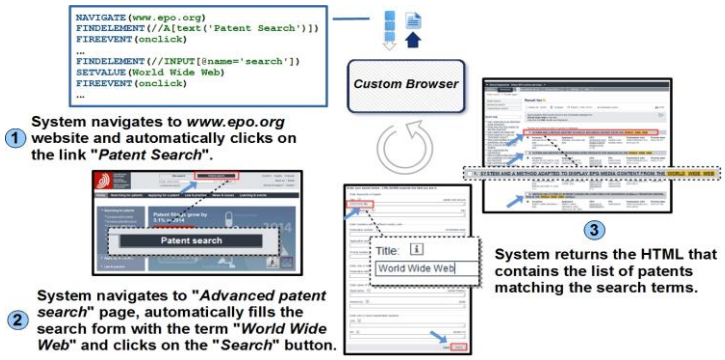

Fig. 1. Navigation Sequence Example.

The approach followed by most of the current web automation systems [2] [9] [11] [15] [16] [17] consists in using the APIs of conventional web browsers to automate the execution of navigation sequences. This approach does not require developing a custom navigation component, and guarantees that the accessed pages will behave the same as when they are accessed by a human user. While this approach is adequate to some web automation applications, it presents performance problems for intensive web automation tasks which require real time responses (because web browsers are client-side applications and they consume a significant amount of resources).

There exist other systems which use the approach of creating custom browsers to execute web navigation sequences [5] [8] [10]. Since they are not oriented to be used by humans, they can avoid some of the tasks of conventional browsers (e.g. page rendering). Nevertheless, they work like conventional browsers when building the internal representation of the web pages. Since this is the most important part in terms of the use of computational resources, their performance enhancements are not very significant.

In this work, we present a custom browser architecture oriented to the efficient execution of web navigation sequences. This architecture is influenced by a set of optimizations that we have designed to be automatically applied during the process of loading and building the internal representation of the web pages. Some of these optimizations are based on the fact that, in the web automation systems, navigation sequences are defined 'a priori' and executed multiple times. Using this peculiarity, the navigation component can extract some useful information during the first execution of the sequence (at definition time) and use that information in the next executions of the same sequence, to minimize the use of resources (CPU, memory, bandwidth and execution time). To support these optimizations, the proposed architecture includes some novel components not present in any other web navigation systems.

The rest of the paper is organized as follows. Section 2 briefly describes the models our approach relies on. Section 3 presents an overview of the architecture and functioning of the conventional and custom browsers, and introduces a set of automatic optimizations that can be applied in custom browsers. Section 4 explains in detail the proposed architecture. Section 5 describes the experimental evaluation of the approach. Section 6 discusses related work. Finally, section 7 summarizes our conclusions.

2 Background

2.1 Document Object Model

The main model we rely on is the Document Object Model (DOM) [4]. This model describes how browsers internally represent the HTML web page currently loaded and how they respond to user performed actions on it. An HTML page is modelled as a tree, where each HTML element is represented by an appropriate type of node. An important type of nodes are the script nodes, used to execute a script code typically written in a scripting language such as JavaScript.

In addition, every node in the tree can receive events produced (directly or indirectly) by the user actions. Event types exist for actions such as clicking on an element (*click*), or moving the mouse cursor over it (*mouseover*), to name but a few. Each node can register a set of listeners for different types of events. An event listener executes arbitrary script code that has the entire page DOM tree accessible and can perform actions such as modifying existing nodes, creating new ones or even launching new events.

2.2 Dependencies between nodes

In our previous work [12], we introduced the concept of dependency between nodes of the DOM. This is a key concept in the custom browser architecture proposed in this work. We can summarize the idea with the following definitions:

Definition 1. We say the node *n1* depends on node *n2* when *n2* is necessary for the correct execution of *n1*. We say that *n2* is a dependency of *n1* and denote it as $n1 \rightarrow n2$. The following rules define this type of dependencies:

1. If the script code of a node *s1* uses an element (e.g. a function or a variable) declared or modified in a previous script node *s2*, then $s1 \rightarrow s2$. Rationale: to be able to execute the script code of *s1*, the node *s2* must be executed previously.
2. If the script code of a node *s* uses a node *n*, then $s \rightarrow n$. Rationale: to be able to execute the script code of *s*, the node *n* must be loaded previously, e.g., if *s* obtains a reference to an anchor node (e.g. using the function *getElementById*) and navigates to the URL specified by its *href* attribute, then it will not be possible to execute *s* unless the anchor node is loaded.
3. If the script code of a node *s* makes a modification in a node *n*, then $n \rightarrow s$. Rationale: the action performed by *s* may be needed to allow *n* to be used later, e.g., if *s* modifies the *action* attribute of a form node to set the target URL, then it will not be possible to submit the form unless *s* is executed previously.

Definition 2. We say that there exists a dependency conditioned to the event *e* being fired over the node *n*, between two nodes *n1* and *n2*, when the node *n2* is necessary for the correct execution of the node *n1*, when the event *e* is fired over the node *n*. We denote this as $n1 \rightarrow^{e|n} n2$. For example, suppose *n* is a node with a listener for the *onMouseOver* event. The listener uses a function defined in *s*. Then $n \rightarrow^{onMouseOver|n} s$. Analogous rules to the ones explained before define this type of dependencies, which, in this case, involve nodes containing event listeners.

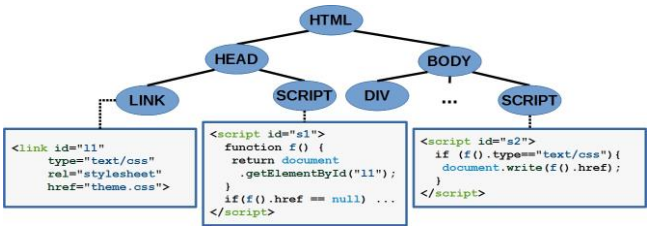

Fig. 2. Example of dependencies between nodes.

Figure 2 illustrates an example of dependencies between nodes. In the example, the script *s1* defines the function *f*. This function access the link node *l1* (using the function *getElementById*), so *s1*→ *l1*. In addition, the script *s2* uses the function *f*, so *s2*→ *s1*.

3 Overview

This section presents an overview about the architecture and functioning of the conventional browsers (section 3.1), custom browsers (section 3.2), and introduces a set of automatic optimizations that can be applied in custom browsers (section 3.3).

3.1 Conventional Web Browsers

A web browser is a software application used for retrieving and presenting resources downloaded from the WWW. The architecture of the modern web browsers (Figure 3.a) [6] includes the high level components: Graphical User Interface, Browser Engine and Rendering Engine; and the auxiliary subsystems: JavaScript Interpreter, Networking, Display Backend, HTML Parser and Data Persistence.

1. The Graphical User Interface includes the browser display area except the main window where the response page is rendered (address bar, toolbars, main menu, etc.).
2. The Browser Engine is a high level interface for querying and manipulating the rendering engine. It provides methods for high level browser actions, e.g., initiate the loading of a URL, go back to the previous page, etc.
3. The Rendering Engine represents the core of the browser. It is the responsible for processing and painting the HTML contents. The page loading process fires a set of events in cascade and most of them are processed sequentially by this component.

Due to the semantics of JavaScript, web browsers execute scripts in a sequential form. Nevertheless, there are some special cases where they can execute JavaScript in parallel. First, the scripts containing the attribute *async* can be executed asynchronously with the rest of the page loading. This feature has been introduced in HTML5 [18]. The other scenario where JavaScript can be executed in parallel is using Web Workers (also introduced in HTML5). A Web Worker can execute JavaScript in background but have the major limitation that the code cannot access the DOM tree objects.

Figure 3.b shows the processing steps of the rendering engine in the web browsers:

1. Download and Decode: the HTML contents are downloaded and decompressed.
2. Processing: the DOM tree is built. For efficiency purposes, this is an incremental process in most of the browsers. When new resources are discovered, they are downloaded and processed (style sheets, scripts, etc.). Style sheets contain presentation information, used to build the page layout. Script nodes contain scripting code.
3. Layout and Rendering: the layout tree contains rectangles with visual attributes like dimensions and colors (this structure is different from the DOM tree). The rendering process paints the layout on the browser window using the display backend layer.

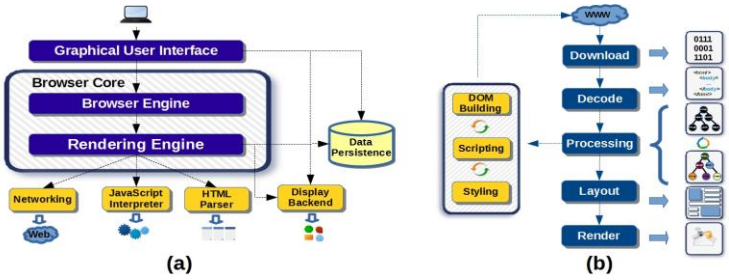

Fig. 3. Web Browsers Reference Architecture and Rendering Engine.

3.2 Custom Browsers

Custom browsers are navigation components, used in web automation systems, specialized in the execution of navigation sequences. Custom browsers usually simulate the behavior of a real browser and they are designed with two main goals: the perfect emulation of a conventional browser (if the custom browser does not behave just the same as a real browser, the sequence execution could lead to wrong web pages) and efficiency in the execution of the navigation sequences.

Custom browsers are not human-oriented and the visualization of the pages is not necessary. This will increase the efficiency because there is no need for building and render the page layout. In the custom browsers architecture, the Browser Engine is also the entry point for accessing the Rendering Engine, but it does not receive commands from the user interface. Instead, the Browser Engine receives the list of commands of the navigation sequence to be executed. These commands will represent events produced by a human user in a real web browser, e.g., navigations to URLs or user events over the DOM elements of the loaded page.

3.3 Automated Optimizations in Custom Browsers

As we have commented, in custom browsers, the rendering of the page layout is not required, because the visualization of the web page is not necessary for the correct execution of the navigation sequence. A first optimization we have considered consists in avoiding the CSS styling of the DOM elements when it is not necessary. Note that CSS styling is necessary only when the style attributes of a node are used during the JavaScript evaluation. Therefore, the styling information can be calculated on-demand only

for the required DOM nodes. In our approach, each node will contain an internal structure with the visualization attributes, initially set to null. During the JavaScript evaluation, when the style attributes of a DOM node are accessed, the visualization information is generated on-demand only for that node.

A second issue to be considered is that, in web automation environments, navigation sequences are known 'a priori' and executed multiple times. This peculiarity can be used to extract some useful information during a first execution of each navigation sequence, at definition time, with the goal to use that information in the following executions and improve its efficiency. We will focus in two points:

1. Load minimized DOM trees. As described in our previous work [12], there are a lot of fragments of the web pages that are not necessary for the correct execution of the navigation sequences. For example, if the navigation sequence fills a form and fires a *click* event on the submit button, in most of the cases, many fragments of the page will not be involved in this sequence execution (for example, ads, banners, *iframes*, menus, etc.). These irrelevant fragments can be ignored (not added to the DOM tree), without affecting to the correct execution of the navigation sequence.
2. Parallelize the execution of script nodes. Web browsers execute the scripts contained in the web pages sequentially (except some particular exceptions explained in section 3.1), even when scripts have no dependencies between them. Script elements that are not dependent could be executed in parallel without affecting to the correct execution of the navigation sequence.

To achieve these two objectives, in our approach, the custom browser will work in two phases: optimization and execution. The optimization phase requires one execution of the navigation sequence. In this execution, the navigation component automatically calculates some optimization information and saves it. More in detail, it calculates:

1. Which nodes of the DOM tree are necessary for the correct execution of the sequence, and which ones can be discarded (irrelevant nodes). To do so, the script evaluation is monitored to collect all dependencies between the nodes in the DOM tree (following the rules cited in the section 2.2). Using this dependencies, the irrelevant nodes are identified and represented using XPath-like [19] expressions. This process is deeply described in [12].
2. A script dependency graph, which contains, for each script *S* in the page, the list of other scripts that must be executed before, because they contain dependencies necessary for the correct execution of the script *S*. The script dependency graph is also calculated using the dependencies between the nodes obtained during the JavaScript evaluation. The scripts contained in this graph are also represented using XPath-like expressions. This process is deeply described in [13].

The execution phase involves the next executions of the same navigation sequence. In this phase, the rendering engine uses the information previously generated to execute the sequence more efficiently. When each page is loaded, a reduced DOM tree is built, discarding the irrelevant nodes, and the scripts of the page are evaluated in parallel according to the script dependencies graph.

4 Proposed Architecture

In this section, we describe the proposed architecture for a custom browser able to support the previously described optimization techniques.

1. To support the on-demand CSS simulation technique, the custom browser architecture should take into account that:
   (a) The visualization information will be stored in the DOM nodes directly.
   (b) The CSS Subsystem will be accessed only from the JavaScript Engine.
2. To support the minimized DOM optimization technique, the designed architecture should include the following elements:
   (a) A module able to interact with the JavaScript Engine during the optimization phase to collect the dependencies between the nodes.
   (b) A module able to interact with the HTML Subsystem during the execution phase, to detect and discard the irrelevant nodes previously identified.
   (c) The Data Persistent Layer should be extended to provide a mechanism for saving and retrieving the irrelevant nodes of each web page.
3. To support the parallel JavaScript execution technique, the custom browser architecture should include the following elements:
   (a) A module able to calculate the graph with the dependencies between scripts.
   (b) The Data Persistent Layer should be extended to provide a mechanism for saving and retrieving the script dependency graph.
   (c) A pool of reusable threads to execute scripts in parallel.
   (d) A component able to detect available scripts and execute them in parallel using the pool of reusable threads.

4.1 Architecture Core Components.

Figure 4 shows the components of the custom browser architecture: Browser Engine, Rendering Engine, Data Persistence Layer and Browser Core Objects; the Rendering Engine subsystems: Main Thread, Event Queue, Dispatcher Thread, Thread Pool (for the parallel script execution); and the auxiliary subsystems: HTML Engine, JavaScript Engine, CSS Subsystem, Networking Layer, and Optimizer.

The **Browser Engine** receives the list of commands from the navigation sequence and translates them into events that are placed in the **Event Queue**. The Event Queue contains the sorted list of events pending for its execution. Each event execution can produce new events (child events) that are also placed in this queue.

The **Dispatcher Thread** is the responsible for assigning events to execution threads. When the custom browser is executed as a regular browser (without using the optimization information), all events are executed in the **Main Thread** one by one, until the queue is empty. When the custom browser is executed using the optimization information previously collected during the optimization phase, the Dispatcher Thread analyzes the event queue, looking for scripts ready for its execution that could be evaluated in parallel. A script is parallelizable when all other scripts that depends on, have already

finished its execution. The **Thread Pool** (containing reusable threads) is used to evaluate these scripts in parallel. Other events are executed in the Main Thread. If all threads of the pool are busy evaluating scripts, the Main Thread can also execute scripts.

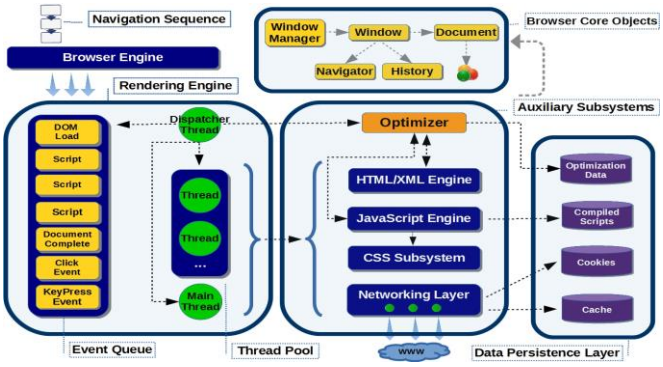

Fig. 4. Browser Architecture Core Components.

During the optimization phase, the **Optimizer** is the responsible for the calculation of the dependencies between the nodes, the set of irrelevant nodes (not required for the correct execution of the sequence) and the script dependency graph. It is also the responsible for saving the optimization information using the **Data Persistence Layer**. During this phase, when the scripts are evaluated, the **JavaScript Engine** invokes the Optimizer to collect the dependencies between the nodes. Then, after the page loading, when all scripts finished its execution, the Optimizer analyzes these dependencies and generates the optimization information using XPath-like expressions to represent the nodes. In the execution phase, the Dispatcher also uses the Optimizer to detect the scripts that could be executed in parallel. When a script finishes its execution, the Optimizer updates the script dependency graph and the Dispatcher is notified. If there are new scripts ready for execution that could be evaluated in parallel, the Dispatcher places them in the available threads of the pool.

The **CSS Subsystem** parses and stores the CSS snippets. The JavaScript Engine uses the CSS Subsystem to dynamically calculate the CSS style attributes on-demand (during the script evaluation). If the JavaScript code does not reference the style attributes, these calculations can be omitted, saving the corresponding processing time. Figure 5 illustrates an example of on-demand CSS styling and also outlines the pseudo-code of the algorithm that calculates the structure with the CSS properties. In the example, the CSS attributes are calculated only for two nodes (*html* and *body*).

The **HTML Engine** is responsible for parsing the HTML and XML streams. It uses the Optimizer (during the DOM building stage) to identify the irrelevant fragments, and build a minimized version of the DOM tree containing only the relevant nodes.

The **Networking Layer** is responsible for the execution of HTTP requests. Multiple downloads can be executed in parallel. A cache of downloaded files is provided to increase the performance and prevent unnecessary downloads.

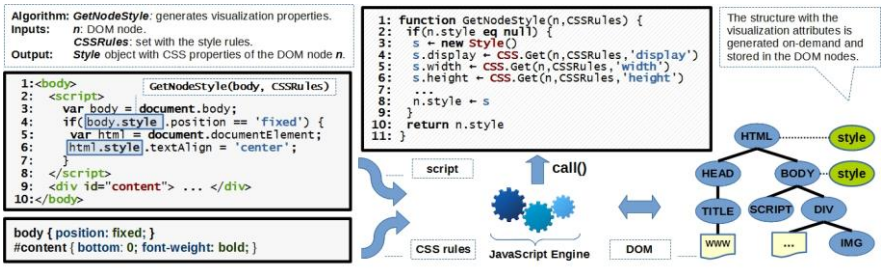

Fig. 5. On-Demand CSS Styling.

All the subsystems can access to the **Browser Core Objects**, including the windows and the documents with the DOM tree of each loaded page. Windows and frames can be accessed thought the window manager object. Each window contains the currently loaded document object (and also the history with previously loaded documents). Each node can contain an additional structure with the visualization information. This structure is generated only if the style attributes are required during the script evaluation.

The **Data Persistence Layer**: provides a mechanism for accessing the persistent information, including the cache of downloaded JavaScript and CSS files, the cache of compiled scripts, cookies, optimization information (irrelevant nodes and the script dependency graph) and browser configuration parameters.

4.2 Event Execution Model

The event execution model considers different types of events (e.g. DOM load, script execution, user actions, etc.) and each event stores information about its execution state. Figure 6 shows the supported event states and the transitions between them:

Created: initial state, before adding the event to the queue. Most events immediately switch to Ready state when they are inserted in the queue. If the event depends on other actions (e.g. download a file), it will be placed in the queue as *Not Ready (Locking)*. If the event requires a delay (e.g. using *setTimeout* function), it will be placed in the queue as *Not Ready (Unlocking)*.

Not Ready: the event is in the queue but it cannot be executed because there are unfinished pending actions associated to the event. *Not Ready (Locking)* events block the queue and no other events can be executed until this event finishes (except parallelizable scripts). *Not Ready (Unlocking)* events does not block the queue and other events can be executed in the meantime.

Ready: the event is ready to be executed. If it is a script event and it is parallelizable, then it can be executed, in the Main Thread or in a thread of the pool, when all its dependencies (according to the script dependency graph) are in *Completed* state. In other case, it will be executed in the Main Thread following the queue order.

Running: the event is out of the queue and it is being executed. This running event can generate new events that must be executed immediately (even before finishing its own execution). In that case, this running event pauses its execution and returns to the queue. For example, if a style sheet is discovered during the HTML parsing, a CSS

event is created and placed in the queue; the HTML parsing stops its execution, returns to the queue (in a position higher than the CSS event), and it will continue just after finishing the CSS processing.

If the event is a periodic timer event (e.g. using *setInterval* function), it will be placed in the queue again, just after finishing its execution.

Finished: the event finished its execution but has unfinished child events. This state is used to correctly detect when a script has completely finished (necessary to evaluate the scripts in parallel). If a script *S*, produces JavaScript child events, other scripts in the page detected as dependent of *S* cannot be executed until the child events of *S* end its execution (at that moment, *S* switch to *Completed* state and the script dependency graph is updated).

Completed: the event and its child events have finished (this is a recursive process).

Failed: the event (or one associated preload action) has finished with errors.

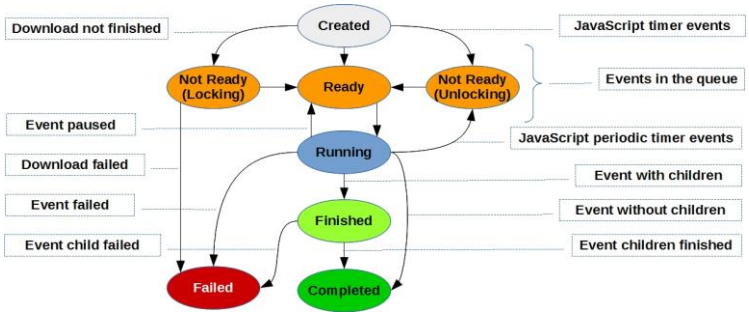

Fig. 6. Event Transition States.

4.3 Rendering Engine Processing Steps and Thread Model

Figure 7.a shows the processing steps of the rendering engine of a custom browser designed according to the proposed architecture (using its automatic optimization capabilities at execution time). We can summarize the differences with other navigation systems as follows: the Layout and Render steps are not necessary; CSS rules are applied on-demand only to the required nodes; when building the page, irrelevant fragments are identified and not added to the tree; and finally, the scripts are evaluated in parallel, following the script dependency graph.

Figure 7.b illustrates the multi-thread model used in the browser architecture.

Browser Engine Thread: using a separated thread for the Browser Engine allows a better control on the rendering engine. In addition, the navigation sequence commands can be placed in the queue asynchronously.

Dispatcher Thread: responsible for selecting events from the queue and assigning these events to the execution threads. Events can be selected in queue order (*pop* method) to be executed in the Main Thread (note that, the *pop* method does not always return the first event in the queue, due to the state *Not Ready (Unlocking)*), or not following the queue order (*get* method) if they are script events that can be executed in a thread of the pool.

Main Thread: executes all kind of events. This thread can access to the event queue to place new events generated during the execution of the running event.

Parallel Scripts Thread Pool: execute JavaScript events in parallel. These threads can also place new events in the queue.

Network Thread Pool: executes HTTP requests in parallel.

The inter-thread communication is managed through the Event Queue and the events inserted in it. The Browser Engine will keep a reference to the events added to the queue (when the navigation sequence commands are translated to browser events). Using this reference, the Browser Engine will be able to know the execution progress because each event stores information about its current state, previous transitions between states, child events generated, etc. The Browser Engine will place in the queue special types of events for actions such as stop the execution after a predefined timeout, etc.

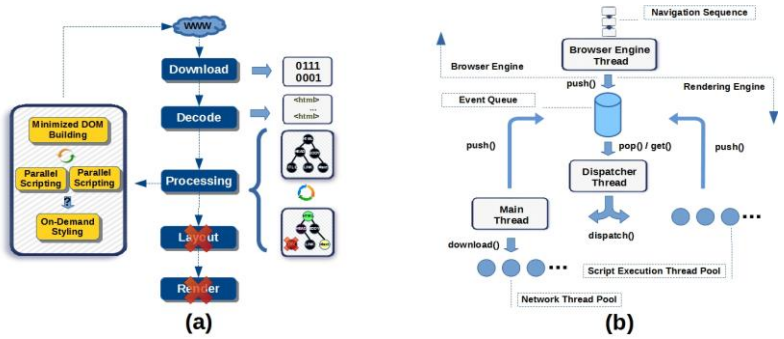

Fig. 7. Rendering Engine Processing Steps and Thread Model.

5 Evaluation

To evaluate the validity of the proposed architecture, we developed a reference implementation that emulates Microsoft Internet Explorer. This navigation component was implemented in Java using open source libraries. In the experiments, we selected websites from different domains and different countries, included in the top 500 sites on the web according to Alexa [1]. The test machine was a quad-core with 16GB of RAM. The thread pool size (for parallel script evaluation) was limited to a maximum of 3.

In the first experiment, we tested the architecture performance comparing the execution time of our custom browser using its automatic optimization capabilities with the custom browser without using the optimization techniques, and also with other representative navigation components. On one hand, we used a navigation component based on HtmlUnit [8] because it is a popular open source project with JavaScript and CSS support. On the other hand, we used a navigation component developed using the APIs of Microsoft Internet Explorer (MSIE from now). The three navigation components (the reference implementation, HtmlUnit and MSIE) were configured to use its caching capabilities. In addition, MSIE was configured to prevent image downloading and plugin execution (e.g. to avoid showing banner videos). In each website we recorded a navigation sequence representative of its main function (e.g. a product search

in an e-commerce website). Every sequence executed events to fill and submit forms, to navigate through hyperlinks, etc.

Table 1 shows the average execution time of 30 consecutive executions of each navigation sequence used in the tests, discarding those executions that do not fit in the range of the standard deviation. The table also shows, between brackets (in the second, third and fourth columns), the percentage of the execution time in comparison with the custom browser using the automatic optimization capabilities (first column).

The execution of the custom browser using its optimization capabilities always got best results (first column). Calculating the average of the percentages, the execution time of the custom browser without using its automatic optimization capabilities is 3.1 times slower (310%). Discarding the results that do not fit in the range of the average ± standard deviation it is 2.4 times slower, and the median value indicates that it is 2.46 times slower. Regarding the other two browsers, HtmlUnit is the one that got better results. It is, in average, 4.7 times slower (470%). Discarding the results that do not fit in the range of the average ± standard deviation it is 3.49 times slower, and the median value indicates that it is 3.78 times slower. The navigation component based on MSIE is, in average, 6.84 times slower (684%). Discarding the results that do not fit in the range of the average ± standard deviation it is 5.44 times slower, and the median value indicates that it is 5.34 times slower.

In the second experiment, we executed a load test benchmark using multiple browsers executing the same navigation sequence in parallel. This experiment was not executed using the real websites because most of them do not allow the level of concurrency required for the parallel load testing. Instead, we simulated the real web site saving the contents of the downloaded pages (including the JavaScript files, CSS files, etc.) in a local web server, and modifying HTML contents and JavaScript files to emulate the form submission and the AJAX requests (this simulation forbade HTTP requests outside the local web server). In this experiment, 30 different instances of the same type of browser (e.g. 30 custom browser instances, 30 MSIE instances and 30 HtmlUnit instances) executed the same navigation sequence in parallel during 5 minutes.

Table 2 shows the number of finished executions of the navigation sequence using the custom browser (with and without using its optimization capabilities), and also using HtmlUnit and MSIE. The custom browser when uses its optimization capabilities always got best results (first column). Compared with the custom browser without using optimization capabilities, it completed, in average, 4.89 times more executions (489%). Discarding the results that do not fit in the range of the average ± standard deviation, it completed 3.5 times more executions, and the median value indicates that it completed 3.6 times more executions. Compared with HtmlUnit, the custom browser using its automatic optimization capabilities completed, in average, 14.25 times more executions (1425%). Discarding the results that do not fit in the range of the average ± standard deviation, it completed 9.29 times more executions and the median value indicates that it completed 9.68 times more executions. Compared with MSIE, it completed, in average, 20.57 times more executions (2057%). Discarding the results that do not fit in the range of the average ± standard deviation, it completed 12.33 times more executions, and the median value indicates that it completed 12.99 times more executions.

Table 1. Execution Times.

	OPTIMIZED (MS)	NOT OPTIMIZED (MS)	HTMLUNIT (MS)	MSIE (MS)
360.CN	2613	5274 (219%)	6116 (234%)	7974 (305%)
ALIBABA.COM	2759	6249 (226%)	13350 (483%)	12025 (435%)
ALLEGRO.PL	1027	5142 (500%)	9372 (912%)	11185 (1089%)
AMAZONWS.COM	2093	3455 (165%)	9802 (468%)	9722 (464%)
BBC.COM	1220	4194 (343%)	7901 (647%)	6898 (565%)
BET365.COM	1092	1787 (163%)	3195 (292%)	10922 (1000%)
BILD.DE	3450	18003 (521%)	21207 (614%)	15161 (439%)
BLOGGER.COM	1004	4033 (401%)	4409 (439%)	7750 (771%)
BLOOMBERG.COM	2880	5738 (199%)	11149 (387%)	11874 (412%)
BOOKING.COM	5105	6731 (131%)	11378 (222%)	12649 (247%)
CNET.COM	1298	2586 (199%)	3360 (258%)	11586 (892%)
ENGADGET.COM	496	1890 (381%)	5843 (1178%)	9198 (1854%)
FORBES.COM	754	3420 (453%)	3846 (510%)	6706 (889%)
GITHUB.COM	1183	3587 (303%)	3001 (253%)	7254 (613%)
GIZMODO.COM	607	1752 (288%)	2124 (349%)	9109 (1500%)
GSMARENA.COM	1388	8172 (588%)	9084 (654%)	10706 (771%)
IGN.COM	2269	4768 (210%)	4834 (213%)	9135 (402%)
IKEA.COM	199	1105 (555%)	1494 (750%)	5470 (2748%)
IMGUR.COM	2048	14556 (710%)	17402 (849%)	13343 (651%)
INDIATIMES.COM	1517	7732 (509%)	6512 (429%)	7930 (522%)
INSTAGRAM.COM	1367	2419 (176%)	1969 (144%)	7867 (575%)
LEMONDE.FR	405	1950 (481%)	7996 (1974%)	8679 (2142%)
LIBERO.IT	852	2605 (305%)	1930 (226%)	4386 (514%)
LIFEHACKER.COM	1162	1862 (160%)	4298 (369%)	8702 (748%)
LINKEDIN.COM	1507	4405 (292%)	6685 (443%)	5754 (381%)
LIVEJOURNAL.COM	1350	10655 (789%)	19849 (1470%)	17942 (1329%)
MARCA.COM	899	8007 (890%)	10026 (1115%)	9741 (1083%)
MASHABLE.COM	666	1879 (282%)	3089 (463%)	6742 (1012%)
MEDIAFIRE.COM	4271	5935 (125%)	6409 (135%)	7832 (165%)
PETFLOW.COM	1283	5433 (423%)	5853 (456%)	6673 (520%)
PINTEREST.COM	5263	6877 (130%)	6463 (122%)	8310 (157%)
REDIFF.COM	1799	5024 (279%)	5865 (326%)	7531 (418%)
REUTERS.COM	7021	18125 (258%)	20031 (285%)	16620 (236%)
RT.COM	2566	7064 (275%)	12033 (468%)	9913 (386%)
SCRIPBD.COM	8005	9673 (120%)	11923 (148%)	14817 (185%)
SOFTONIC.COM	933	3524 (377%)	4821 (516%)	6403 (686%)
SOURCEFORGE.NET	4868	10593 (217%)	12828 (263%)	18260 (375%)
SPEEDTEST.NET	2139	4932 (230%)	6386 (298%)	10823 (505%)
STACKEXCHANGE.COM	2097	5008 (238%)	5541 (264%)	9312 (444%)
TAOBAO.COM	1588	2472 (155%)	8051 (506%)	11610 (731%)
TARINGA.NET	5249	10886 (207%)	8734 (166%)	9695 (184%)
TECHCRUNCH.COM	604	2095 (346%)	5868 (971%)	8551 (1415%)
THEFREEDICTIONARY.COM	915	6307 (689%)	6539 (714%)	7112 (777%)
TIME.COM	4092	6243 (152%)	12103 (295%)	11676 (285%)
TRIPADVISOR.COM	1050	2281 (217%)	6561 (624%)	6997 (666%)
TUMBLR.COM	3469	5054 (145%)	5805 (167%)	7858 (226%)
UPLOADED.NET	1496	3321 (221%)	3682 (246%)	9558 (638%)
UPS.COM	2232	4016 (179%)	3057 (136%)	5846 (261%)
USATODAY.COM	335	1280 (382%)	2161 (645%)	4478 (1336%)
WARRIORFORUM.COM	1540	3091 (200%)	3396 (220%)	7913 (513%)
WEATHER.COM	2045	4457 (217%)	11260 (550%)	10407 (508%)
WIX.COM	1655	3097 (186%)	4024 (241%)	4908 (294%)
WORDPRESS.COM	1975	2770 (140%)	2848 (144%)	10793 (546%)
WORDREFERENCE.COM	832	5086 (611%)	7778 (934%)	6507 (782%)
XDA-DEVELOPERS.COM	3175	5966 (187%)	9793 (308%)	10585 (333%)
YAHOO.COM	3680	5483 (148%)	6590 (179%)	8734 (237%)
YOUTUBE.COM	664	1872 (281%)	2730 (411%)	6334 (953%)
ZIPPYSHARE.COM	1049	2680 (255%)	2228 (212%)	5867 (559%)
AVERAGE		310%	470%	684%
AVERAGE ± STDEV		240%	349%	544%
MEDIAN		246%	378%	534%

Table 2. Load Tests BenchMark.

	OPTIMIZED	NOT OPTIMIZED	HTMLUNIT	MSIE
AMAZON.COM	16520	1728 (956%)	552 (2992%)	345 (4788%)
APPLE.COM	6570	3986 (164%)	654 (1004%)	858 (765%)
EBAY.COM	6171	3504 (176%)	1026 (601%)	492 (1254%)
FLICKR.COM	34792	6244 (557%)	909 (3827%)	588 (5917%)
GOOGLE.COM	19719	1737 (1135%)	2413 (817%)	1823 (1081%)
IMDB.COM	6048	2007 (301%)	519 (1165%)	633 (955%)
LINKEDIN.COM	28364	11483 (247%)	4495 (631%)	2110 (1344%)
WALMART.COM	3342	870 (384%)	240 (1392%)	189 (1768%)
WIKIPEDIA.COM	12722	3779 (336%)	1430 (889%)	669 (1901%)
WSJ.COM	4146	651 (636%)	444 (933%)	516 (803%)
AVERAGE		489%	1425%	2057%
AVERAGE ± STDEV		350%	929%	1233%
MEDIAN		360%	968%	1299%

6 Related Work

Most of the current web automation systems (Smart Bookmarks [9], Wargo [15], Selenium [17], Kapow [11], WebVCR [2],WebMacros [16]) use the APIs of conventional browsers to automate the execution of navigation sequences. This approach has two important advantages: it does not require to develop a new browser (which is costly), and it is guaranteed that the page will behave in the same way as when a human user access it with her browser. Nevertheless, it presents performance problems for intensive web automation tasks which require real time responses. This is because web browsers are designed to be client-side applications and they consume a significant amount of resources.

Other systems use the approach of creating simplified custom browsers. For example, Jaunt [10] lacks the ability to execute JavaScript. HtmlUnit [8] and EnvJS [5] use their own custom browser with support for advanced JavaScript features. They are more efficient than conventional web browsers, because they are not oriented to be used by humans and can avoid some tasks (e.g. rendering). Nevertheless, they work like conventional browsers when building the internal representation of the web pages. Since this is the most important part in terms of the use of computational resources, their performance enhancements are smaller than the ones achieved with our approach.

Traditional web browsers (Firefox, Chrome, etc.) implement some optimizations, (e.g., Mozilla Firefox uses the speculative parsing [13] to early discover resources and start preload actions), but they always calculate the CSS visualization information of all the DOM nodes, evaluate scripts in a sequential form, and load the pages completely.

Other browsers exploit different levels of optimization and parallelism. For example, ZOOMM [3] is a parallel browser engine that exploits HTML pre-scanning with resource prefetching, concurrent CSS styling and parallel script compilation, and Adrenaline [7] speeds up page processing by splitting the original page in mini-pages, rendering each of these mini-pages in a separate process.

7 Conclusions

In this paper we presented a complete architecture for a headless custom browser specialized in the execution of web navigation sequences. The architecture supports a set of novel automatic optimization techniques not implemented in any other navigation component and includes some elements not present in any other navigation system.

This architecture design, exploits some peculiarities of web automation environments. First, custom browsers do not require some operations that are unconditionally executed in conventional browsers (e.g. build page layout and rendering), and second, the fact that, in the web automation systems, the same navigation sequences are executed multiple times. This peculiarity is used to extract some useful information during a first execution of each navigation sequence, with the goal to use that information in the following executions and improve the efficiency.

To evaluate the validity of the proposed architecture, we developed a reference implementation following the architecture principles. In the experiments, we analyzed the

performance of the architecture comparing our custom browser with other navigation components. The reference implementation, using optimization techniques, got the best results, followed by the same reference implementation without using those optimization capabilities, and, at a greater distance, by the other navigation components.

We can conclude that a custom browser built according to the proposed architecture is able to execute the navigation sequences faster, consuming fewer resources than other existing navigation components.

8 References

1. Alexa. The Web Information Company. http://www.alexa.com.
2. Anupam, V., Freire, J., Kumar, B., Lieuwen, D.: Automating web navigation with the WebVCR. Comput. Netw. 33(1–6), 503–517 (2000).
3. Calin Cascaval, Seth Fowler, Pablo Montesinos-Ortego, Wayne Piekarski, Mehrdad Reshadi, Behnam Robatmili, Michael Weber, and Vrajesh Bhavsar. 2013. ZOOMM: a parallel web browser engine for multicore mobile devices. In Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP '13). ACM, New York, NY, USA, 271-280.
4. Document Object Model (DOM). http://www.w3.org/DOM/.
5. EnvJS. http://www.envjs.com/.
6. Grosskurth, A., Godfrey, M. W., September 2005. A reference architecture for web browsers. In: ICSM'05: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05). pp. 661–664.
7. H. Mai, S. Tang, S. T. King, C. Cascaval, and P. Montesinos. A case for parallelizing web pages. In Proceedings of the 4th USENIX conference on Hot Topics in Parallelism, HotPar'12, Berkeley, CA, USA, June 2012. USENIX Association.
8. HtmlUnit. http://htmlunit.sourceforge.net/.
9. Hupp D., Miller R.C.: Smart Bookmarks: automatic retroactive macro recording on the web. In: Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology, pp. 81–90. ACM New York, Newport (2007).
10. Jaunt. Java Web Scraping & Automation. http://jaunt-api.com.
11. Kapow: http://kapowsoftware.com/.
12. Losada J., Raposo J., Pan A., Montoto P.: Efficient execution of web navigation sequences. World Wide Web Journal. DOI 10.1007/s11280-013-0259-8. ISSN 1386-145X.
13. Losada J., Raposo J., Pan A., Montoto P., Álvarez M.: Optimization Techniques to Speed Up the Page Loading in Custom Web Browsers. Manuscript accepted for publication in ICEBE 2015. Beijing, China (23-25 October 2015).
14. Mozilla HTML5 Parser. https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/HTML5/HTML5_Parser
15. Pan A., Raposo J., Álvarez M., Hidalgo J., Viña A.: Semiautomatic wrapper generation for commercial web sources. In: IFIP WG8.1 Working Conf. on Engineering Information Systems in the Internet Context, pp. 265–283. Kluwer, B.V. Deventer, Japan (2002).
16. Safonov A., Konstan J., Carlis J.: Beyond hard-to-reach pages: interactive, parametric web macros. In: 7th Conference on Human Factors & the Web. Madison 2001.
17. Selenium: http://seleniumhq.org.
18. HTML5. https://html.spec.whatwg.org.
19. XML Path Language (XPath), http://www.w3.org/TR/xpath