Improving performance of web applications

Grega Jakus, Jaka Sodnik
University of Ljubljana, Faculty of Electrical Engineering, Ljubljana, Slovenia
grega.jakus@fe.uni-lj.si, jaka.sodnik@fe.uni-lj.si

***Abstract*—Although web browsers were initially designed for displaying simple documents, they have gradually evolved into complex pieces of software. Modern browsers are expected to provide support for complex web applications, which are often competing with native applications. By understanding how the browser displays content, developers can implement efficient architectural solutions and more easily overcome obstacles inherited from the browsers' early design. They can efficiently utilize hardware resources and provide user experience comparable to that of native applications. The purpose of this paper is, therefore, to present concisely the architecture of modern browsers, and to suggest some of the most effective methods for optimizing web applications with the focus on networking, rendering and JavaScript.**

I. INTRODUCTION

When introduced in the early nineties, the web browser was designed for displaying simple web documents. However, along with the evolution of the Web that followed, the browser also evolved into a complex piece of software. Instead of merely displaying web content, modern browsers are capable to provide an environment for running complex applications with rich user interfaces with many transitions and animations, multimedia content and interactions with the user.

Despite using relatively complex content generation and delivery models, already being platform-independent, easily accessible and customizable, web applications are also very often expected to provide similar performance and user experience as the native applications. This is, however, hard to achieve, especially because web applications are often subjected to inefficient utilization of client-side hardware resources, which is partly also due to the browser's design inherited from the early design of the web. However, many of these inefficiencies can be avoided by exploiting the knowledge on how the browser handles documents.

Since many of the browser vendors are unwilling to disclose the detailed information on the components of their browsers and the implemented mechanisms are subject to frequent changes, it is difficult to find topical literature. The purpose of this paper is, therefore,

- to present concisely the architecture of modern browsers, and
- to suggest optimizations of the common aspects of web applications that enable relatively straightforward performance improvements. We especially focus on the improvements that affect user experience, which has become one of the key competitive advantages in the software market.

As they represent the sources where most of the optimizations can be carried out, we focus on improving the application parts that rely on browser's network and rendering and JavaScript components.

II. THE BROWSER

A typical modern web browser consists of the following components [1]:

- user interface, which includes all the visible elements of the browser, such as address bar and navigation buttons, except the main window showing the web page content;
- browser engine, which coordinates the user interface and rendering;
- rendering engine, which renders the content. If it is written in HTML (HyperText Markup Language), the engine parses, interprets and displays the content, otherwise it uses one of the third-party plug-ins (e.g. in case of Macromedia Flash);
- user interface backend, responsible for rendering the content using visual components provided by the underlying operating system (buttons, form elements, etc.). As a consequence, web pages can look differently on different operating systems even if they are displayed using the same browser;
- networking component, which is responsible for the transfer of content from and to the web server;
- JavaScript engine, which runs the client-side scripts written in JavaScript language;
- local data storage, which stores the client-side application data using web cookies or various other technologies, such as localStorage and IndexedDb.

The interaction between the components is illustrated in Figure 1.

III. THE OPTIMIZATIONS

A. Network

Timely transmission of content from the server is one of the major factors that affect user experience. If transfer times are too long, the application may act unresponsive. The two most problematic limitations that may affect the application responsiveness are limited network bandwidth and limited number of connections the browser can maintain with a same source. To alleviate these problems, we can optimize the:

- size of the resources (using compression and minimization),
- number of the HTTP (HyperText Transfer Protocol) requests, and

- the resource loading strategy (using separate application domains and preloading).

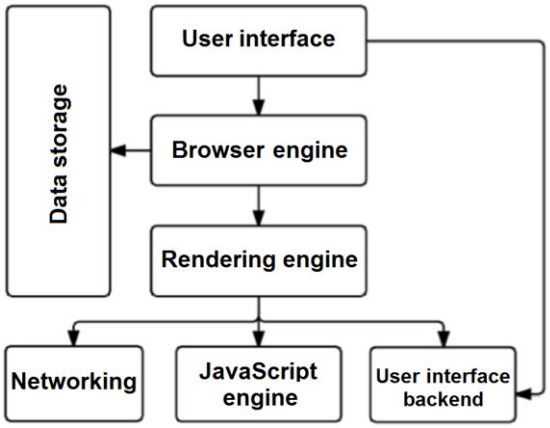

Figure 1: Browser components and their interaction

1) Reducing the size of resources

HTTP compression can be used in order to make the connection between a server and a browser optimally utilized. The compression works by compressing the data the server sends to the browser with a lossless compression algorithm that both, the browser and the server understand. In this way, the amount of the data transferred is reduced, and consequently the data throughput is increased.

When the browser requests a resource from the server, it advertises the compression methods it supports (for example: "*Accept-Encoding: gzip, deflate*") in the *Accept-Encoding* field of the HTTP request. If the server supports one of the advertised methods, it compresses the contents with the appropriate algorithm and informs the browser about the compressed content using the *Content-Encoding* field in the HTTP response (for example: "*Content-Encoding: gzip*") [2].

The data throughput can be further optimized by removing redundant content in the documents, which does not affect the functionality of the web application. One example is removing the redundancy in JavaScript documents by removing whitespaces, comments and unused code, and using short names for variables and functions. This process, known as *minimization*, is similar to lossless compression with a fundamental difference: it does not require the reverse process. Several tools for minimizing JavaScript code exist including UglifyJS, YUI compressor, Google Closure compiler and JSMin. The disadvantage of minimization is a poor code readability which can be, however, improved by the browser tools known as *prettifies*.

Web applications often include a large amount of images. If their size is not optimized, the transfer includes also unnecessary data, which consequently prolongs the page loading time. It is therefore recommended to use lossless or even lossy image compression methods. The lossy image compression can be more effective than the lossless type, however it has a greater impact on the image quality.

2) Optimizing the HTTP requests

As the images are stored separately from HTML documents within which they are displayed, they are also transferred separately from the web server. If many HTTP requests are needed to display a web page, some of them will have to wait as the browser limits the number of simultaneous connections.

One way to reduce the number of image transfer requests is using the so-called *CSS sprites*. This method includes combining multiple images into a single one and transferring it as such to the browser. Then, by using the CSS (Cascading Style Sheets) language, the individual images of the composite image are displayed within the web page. In this way, we achieve the same effect as if transferring the images separately but save a lot of time because a single HTTP request to the server is required. An example of CSS sprites can be found in [3].

Many web applications use cookies for a variety of purposes, such as for identifying and analyzing users etc. Once created, a cookie is sent to the server within each HTTP request to a particular domain. Although it does not contain a lot of data, it is ineffective to send a cookie when not needed, for example when requesting static content, such as images and scripts. To avoid sending cookies, such content can be served from another domain which does not use cookies. On the other hand, loading resources that can block the displaying of a web page from another domain (e.g. CSS and some JavaScript documents) can have quite the opposite effect to the one intended. This is due to the fact that transferring resources from another domain requires a new DNS (Domain Name System) query, TCP (Transmission Control Protocol) connection and TLS (Transport Layer Security) handshake (when using HTTPS (HTTP Secure) protocol), which increases web page display times.

3) Resource loading strategies

If there is a high probability a user will need or access particular resources of the application, such resources can be loaded in advance in a process known as *preloading*. Preloading a commonly used content enables a more fluent interaction with the web application. When needed, the preloaded content is, namely, already available to the browser and can therefore be displayed almost instantly, which significantly improves user experience.

The types of preloading include link prefetching, DNS prefetching and prerendering. The selected preloading type can be indicated by the corresponding value of the *rel* attribute (“prefetch”, “dns-prefetch” or “prerender”) of the *link* element referring to the HTML-external resource [4].

B. Rendering

When the browser receives the requested content, it passes it to the rendering engine. In popular open source rendering engines, such as Gecko and Webkit, the rendering is performed as follows [1]:

1. HTML and CSS documents are transformed into a document object model (DOM);

2. The dimensions and the layout of the visual elements is computed;

3. The render tree containing a visual representation of the document is constructed;

4. The elements of the render tree are painted – displayed on the screen.

Figure 2 presents the process of displaying a document in the example of the WebKit rendering engine.

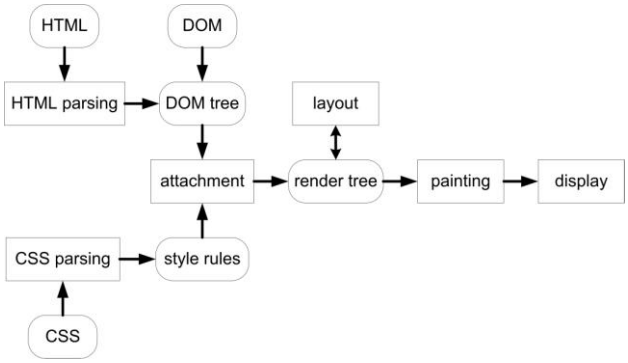

Figure 2. Displaying a web document in WebKit [1]

1) Render tree

The optimizations within the rendering process can be mostly carried out through a careful manipulation of the elements of the render tree. The render tree consists of render objects, which visually represent the web document in the order in which they will be displayed [1]. As the render tree is constructed simultaneously with the DOM tree, any change in the first results in immediate update of the second.

Every render object is represented by a rectangular surface, the size of which is defined by the CSS rules. The HTML elements that cannot be represented by a single rectangular surface (e.g. drop down menu represented by the HTML element *select* and multi-line text) are represented with more objects in the render tree.

When the render object is attached to the render tree, its position and size is computed in a process known as layout or reflow. Since laying out an object does not usually affect the layout of other objects, it can be mostly carried out in a single pass of the render tree.

The layout computation is a recursive process. Each render object owns a layout (or a reflow) method, which invokes the layout methods of object's children. During an initial rendering of a document or when an update affecting all the elements in a document takes place (e.g. when changing font size), the recursion starts at the root render object. This is referred to as the *global layout*. Alternatively, when only a few objects need to be laid out (e.g. when a new object is inserted into render tree as a consequence of an insertion into DOM tree) the process is referred to as the *incremental layout*.

Since the layout process is very resource-demanding, browsers tend to recalculate only the properties of the objects where the changes had actually occurred. For this purpose, browsers use dedicated flags indicating that the layout of the flagged objects or its children needs to be recalculated ("dirty bits", "children are dirty").

In the painting phase, the browser traverses the render tree and invokes the paint method in each of the render objects to display their content. Similarly to layout, the painting can also be global or incremental, depending on the number of the updated objects.

2) Optimizing layout and repainting

As we have mentioned, a modification of the DOM tree triggers the recalculations of layout as well as the repainting of the render objects. The former is a particularly resource-demanding process, especially when many elements need to be repositioned. Even though it can be difficult to avoid layout recalculations and repainting when developing modern web applications, we can greatly reduce their frequency by writing efficient code.

It is generally desirable to keep the number of DOM tree updates as minimal as possible. One way to achieve this is to use a fragment - an element to which other elements can be appended. When all the intended elements are appended to it, the fragment itself can be appended to the DOM tree, thus triggering the layout recalculations and repainting only once. The example bellow demonstrates the use of a fragment to append several paragraphs to the DOM tree.

```
var fragment = document.createDocumentFragment();
var element, contents;

//appending paragraphs to the fragment:
for(var i = 0; i < textlist.length; i++) {
    element = document.createElement('p');
    contents = document.createTextNode(textlist[i]);
    element.appendChild(contents);
    fragment.appendChild(element);
}
//appending the fragment to the DOM tree:
document.body.appendChild(fragment);
```

The element positioning also impacts the calculations of their layout. Laying out elements with absolute and fixed positions is less demanding than positioning elements statically or relatively. It is, therefore, highly recommended to use absolute or fixed positioning when the elements often trigger layout recalculations and repainting (as in case of animations).

3) Optimizing the number of render layers

The render objects visually representing the web page can be placed in a number of overlapping layers. Rendering in multiple layers can be faster due to hardware acceleration, however if we use too many rendering layers we can achieve quite the opposite effect.

Developers can directly affect the number of layers using the CSS property *will-change* or indirectly through the choice of the appropriate HTML elements and CSS properties. If, for example, a developer creates animation using CSS, the browser will automatically create a new layer for animation elements.

C. JavaScript

Like any programming language, JavaScript also offers a number of ways to optimize the code execution. Among others we expose:

- using separate threads for running computationally intensive operations, and
- recycling disposed objects.

In recent years, the number of tasks performed with JavaScript increased substantially. Traditionally, many of the tasks, such as retrieving data from databases and compiling HTML content, were performed on the server. The main task of the browser was rendering the documents it obtained from the server. This has, however, drastically changed with the arrival of Web 2.0 and the so-called Single Page Applications. In such applications, the JavaScript is responsible for compiling HTML, navigating to different parts of the application, communicating with the server, processing the transferred data and many other tasks. As many of these tasks are resource-demanding,

executing them in the main thread usually results in an unresponsive user interface.

One solution is to use the so-called *web workers*. A *shared worker* can be used by any script within the domain while a *dedicated worker* can only be used by the script that created the worker [5].

The code intended to be executed in a dedicated worker thread must be in a separate file and can be invoked from the main script using:

```
var worker = new Worker('task.js');
```

To send some data from the main thread to a web worker, the *postMessage* method is invoked:

```
worker.postMessage('My value');
```

The code that listens for the messages sent from the worker back to the main thread is the following:

```
worker.addEventListener('message', function(e) {
  console.log('Worker says: ', e.data);
}, false);
```

There are two ways to terminate the web worker: the *terminate* method can be called in the main script or the method *close* can be called within the worker itself.

The applications written in JavaScript can be very complex and can use many thousands of objects. Creating an object requires a certain space in memory, which is released when the object is no longer in use and referenced from other objects. Releasing the memory can mainly be done in two ways: manully or automatically. In the first one, a programmer manually removes objects from the memory. However, as this requires a lot of effort and discipline, most high-level languages rely on the so-called *garbage collectors* which automatically remove the waste objects from the memory. In this case, the execution environment determines which objects are suitable for removal, which may be a processor-demanding and, consequently, time-consuming task.

The applications with a large quantity of waste objects have a typical saw tooth-shaped memory usage over time, as shown in Figure 3. The sudden drops in memory usage correspond to the operation of the garbage collector. This can result in an increased load of the main thread and consequently in an unresponsive user interface.

When using plenty of objects of the same type, the problem can be solved with an object pool. Instead of constantly creating new objects, the existing ones that are no longer in use can be reused. As the object pool contains a constant number of objects, the memory usage is relatively constant over time (Figure 3). The implementation of the object pool is described in [6].

It should be noted that using object pool does not actually decrease memory consumption but it only reduces the time and the resources spent on unnecessary removal and creation of objects. Using object pools can also have some side effects, such as slower script startup due to the initial pool initialization, and inefficient memory usage when the application does not require all the objects in the pool.

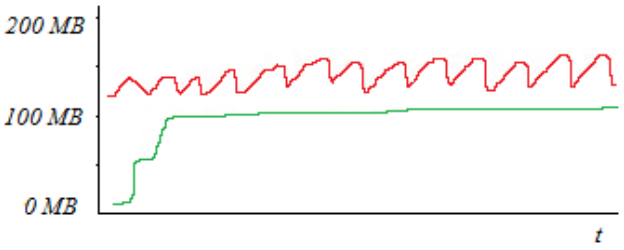

Figure 3. Memory usage over time. Top line: new objects are created when needed and are removed from the memory after use by the garbage collector. Bottom line: object pool with a constant number of reusable objects is used.

IV. Discussion and Conclusion

The environment in which web applications are executed is constantly evolving. Some of the techniques mentioned in this paper will, perhaps, be obsolete in the near future. On the other hand, new challenges will emerge and new solutions will have to be developed. It is essential that developers understand the environment in which their applications are executed and utilize the full potential it offers.

The effect of the optimizations on the application performance depends on many factors including content of the application, quality of its design and code before the optimization, as well as the testing conditions. By implementing the presented optimizations in our sample Single Page Application, we were able to improve our application so that the performance metrics were within the margins established by the RAIL model [7]: all responses to user actions were bellow 100 ms, rendering rate was above 60 Hz and all the content was loaded in less than one second.

Our further work in this field will investigate how the improvements of application performance reflect in the improvements in the actual user experience.

References

[1] T. Garsiel and P. Irsh, “How Browsers Work: Behind the scenes of modern web browsers, http://www.html5rocks.com/en/tutorials/internals/howbrowserswork/, accessed 7.4.2017

[2] Fielding, et al., “Hypertext Transfer Protocol -- HTTP/1.1, 25. 06. 1999”, https://tools.ietf.org/html/rfc2616, accessed 7.4.2017

[3] w3schools.com, “CSS Image Sprites”, https://www.w3schools.com/css/css_image_sprites.asp, accessed 7.4.2017

[4] Wikipedia, “Link prefetching”, https://en.wikipedia.org/wiki/Link_prefetching, accessed 7.4.2017

[5] Mozilla, “Using Web Workers”, https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers, accessed 7.4.2017

[6] SourceMaking.com, “Object Pool”, https://sourcemaking.com/design_patterns/object_pool, accessed 7.4.2017

[7] M. Kearney, “Measure Performance with the RAIL Model”, https://developers.google.com/web/fundamentals/performance/rail, accessed 7.4.2017