Parallel Network Data Processing in Client Side JavaScript Applications

Matthias Wenzel, Christoph Meinel
Hasso Plattner Institute Potsdam
Prof. Dr. Helmert Str. 2-3, Potsdam, Germany
{matthias.wenzel, christoph.meinel}@hpi.de

***Abstract*—In modern computer systems, multicore processors are prevalent, even on mobile devices. Since JavaScript WebWorkers provide execution parallelism in a web browser, they can help utilize multicore CPUs more effectively. However, WebWorker limitations include a lack of access to web browser's native XML processing capabilities and related Document Object Model (DOM). We present a JavaScript DOM and XML processing implementation that adds missing APIs to WebWorkers. This way, it is possible to use JavaScript code that relies on native APIs within WebWorkers. We show and evaluate the seamless integration of an external XMPP library to enable parallel network data and user input processing in a web based real-time remote collaboration system. Evaluation shows that our XML processing solution has the same linear execution time complexity as its native API counterparts. The proposed JavaScript solution is a general approach to enable parallel XML data processing within web browser-based applications. By implementing standards compliant DOM interfaces, our implementation is useful for existing libraries and applications to leverage the processing power of multicore systems.**

***Keywords*—Web-enabled Collaboration; JavaScript; WebWorker; XML**

I. Introduction

Modern web browsers offer a multitude of new features to application programmers that were usually available only in traditional programming environments. Especially in the course of HTML5, lots of functionality, typically requiring plugin technology (e.g. Adobe Flash), can be realized using native browser APIs (e.g. HTML5 Canvas). This way, opportunities arise to develop applications that run on desktop as well as on mobile devices sharing the same code base. Web browsers therefore become an increasingly important application platform [1], making creation and maintenance of multiple, platform specific programs dispensable in more and more cases.

Great improvements on JavaScript engine performance enable even large scale web applications. Optimizations in modern JavaScript engines result in near native code execution performance [2], [3]. An essential constraint, compared to other programming languages, is JavaScript's lack of parallel programming mechanisms. Today, there are multicore processors even on mobile devices. In contrast, traditional JavaScript has a single-threaded execution model [4], which in fact is an obstacle for leveraging the computing power of those devices. In order to solve this issue, the actor-based [5] HTML5 *WebWorker* API [6] has been integrated into modern web browsers. WebWorkers provide a mechanism for running scripts in a thread separated from the main thread and therefore do not interfere with the web browser's user interface rendering and user interaction. Communication is only possible via message passing since there is no shared memory. WebWorkers' data access is limited to a subset of the JavaScript API. Scripts running within WebWorkers cannot access the corresponding *Document Object Model* (DOM).

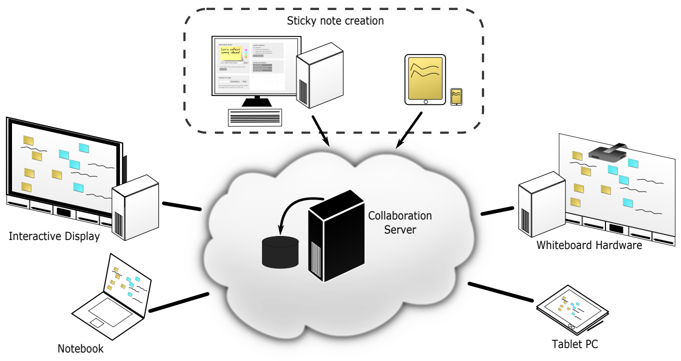


Figure 1. The Tele-Board software system architecture. Web browser based clients can be run on multiple devices. Virtual whiteboard data is synchronized among connected clients by the central collaboration server, relying on XMPP communication protocol.

During a web browser based re-implementation of our digital whiteboard collaboration system *Tele-Board* [7], there was the challenge to combine browser based networking and rendering in a single web application. The application provides a virtual whiteboard surface, where users can draw onto or create and manipulate sticky notes and images. This content data is synchronized automatically by a central server (see Figure 1) among all connected client applications. In order to prevent interruption of user interaction when whiteboard content is synchronized simultaneously, we deployed the network component to a separate WebWorker thread [8]. The current server uses the XML based *Extensible Messaging and Presence Protocol* (XMPP) [9] as communication protocol [10]. Ensuring compatibility with existing infrastructure, our goal was to integrate XMPP message handling in a JavaScript WebWorker script. Due to the mentioned WebWorker access limitations, browser's native functions for XML processing are not available within WebWorkers.

In this paper, we propose and evaluate a JavaScript DOM and XML serialization/parsing implementation. In contrast to existing solutions, our work relies on APIs available in WebWorker scripts. Our standards compliant DOM implementation paves the way for utilizing external JavaScript libraries, relying on native web browser APIs, inside WebWorkers. This way, existing solutions for complex tasks can be transferred to a separate thread.

Utilizing WebWorkers for handling background I/O is a common use case [6]. Our approach solves this task for WebWorker contained processing of XML data combined with XMPP based communication. It is applicable in existing major web browsers such as *Microsoft Internet Explorer*, *Google Chrome* and *Mozilla Firefox*. We developed components for XML string parsing and serialization, an XML document API providing a subset of web browser's native XML document functions and an XMPP message API. We focus on an application-neutral XML processing implementation and show how these components can be used exemplarily with the *Strophe.js*[1] XMPP library for JavaScript.

II. Related Work

JavaScript's status as a scripting language has changed in recent years to a general-purpose programming language [1], [11]. In this context, research has focussed on parallelism in JavaScript as respective applications may profit from that concept in the same way as traditional native applications. Contributions range from compiler to application level implementations.

Mehrara et al. [12] apply parallelism for accelerating JavaScript programs. However, the authors' approach relates to a lower execution level, utilizing multithreading within JavaScript engine optimization techniques such as trace-based just-in-time compilation. Their execution framework facilitates an increased sequential program performance, which differs from our goal of dealing with WebWorker parallelism on application level.

Higher level approaches originate from distinct aspects of WebWorker limitations. The missing support for distributed computation is addressed in [13]. Authors describe a programming model called *generic workers* for unifying local and remote parallelism based on WebWorkers. It is an abstraction layer, that encompasses local and server-side components. An API allows a mostly transparent usage of either local or remote workers. WebWorker's access limitations also apply within this approach. The systems *River Trail* [14] and *TigerQuoll* [15] pursue a similar goal of providing a more flexible parallelization mechanism than that offered by WebWorkers. River Trail allows data-parallel programming. Parallel tasks have immutable access to their parent's state. JavaScript code is compiled to OpenCL [16] which can then be run in parallel on the GPU. TigerQuoll is an event based API and JavaScript runtime providing shared memory. Shared data synchronization is handled automatically, avoiding a great source of error such as locks or race conditions. The system follows JavaScript's event based, asynchronous programming style. The proposed API is similar to server-side JavaScript, e.g. Node.js[2]. Though these approaches, especially TigerQuoll, offer potential for future JavaScript engines, they are proprietary implementations, requiring special JavaScript engines (not usable in current browsers) or do not address issues concerning shared memory.

Erbad et al. [17] describe a system for parallelizing JavaScript applications, which itself is written in JavaScript. *DOHA* is an execution layer dealing with concurrency issues, such as state management and load-balancing. It utilizes WebWorkers for JavaScript concurrency but still relies on their share-nothing memory model and API access limitations.

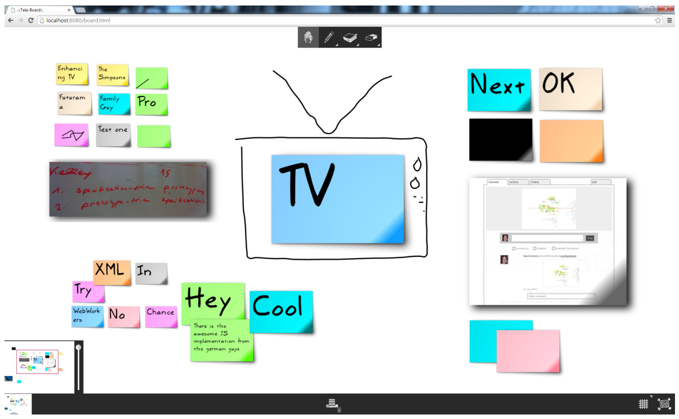


Figure 2. Tele-Board whiteboard client in the web browser. Modifications on elements such as sticky notes, handwriting or images are synchronized via a central server.

Rather than implementing a new concurrency model into existing JavaScript engines, we want to show how a common use case such as parallel background I/O processing can be handled within current web browsers despite their WebWorker native API access limitations. A similar approach to our solution, but for usage on the server side, is *JSDOM*[3]. It is a JavaScript implementation of the DOM, for use with Node.js. On the client side, which is our focus, there are some concrete implementations that closely relate to our use case of XML processing. The popular *jQuery* library[4] provides XML processing capabilities. However, jQuery relies on native functions provided by `DOMParser` [18], which is a property of the global `window` object and not accessible within WebWorker scope. Same applies to JavaScript XML libraries such as *JSXML*[5] and *X2JS*[6]. An exception is *JSXML XML Tools*[7], that uses regular expressions for parsing XML strings.

[1]http://strophe.im/strophejs/

[2]http://nodejs.org/
[3]https://github.com/tmpvar/jsdom
[4]jQuery v2.1.1 - http://jquery.com/
[5]http://jsxml.net/
[6]https://code.google.com/p/x2js/
[7]http://www.petetracey.com/jsxml/

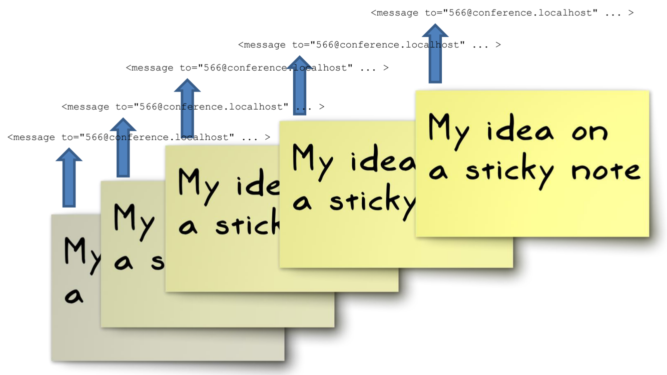
Though, regular expression API is available in WebWorkers, the library's XML string parsing result syntax tree object provides a proprietary interface. It does not implement DOM core [19] `Node` and `Document` interface properties and methods expected by components relying on `DOMParser` parsing results. Using JSXML XML Tools would require considerable changes in existing applications expecting `DOMParser` return object signature. A complex regular expression for splitting an XML string into a list of its individual markup and character data strings is provided in [20]. In contrast to JSXML XML Tools, the XML string is not translated to a syntax tree object, i.e. the actual parsing step is not provided. However, the regular expression is very sophisticated regarding the XML specification's permissions for allowed characters, comments, processing instructions etc. It is a useful basis for a regular expression based XML parser.

To our knowledge, we are the first to address the issue of limited WebWorker API access at the example of the presented use case.

III. Requirements for parallel XMPP message processing

Network message processing is an integral part of our Tele-Board software system. Working synchronously on the same whiteboard content requires continuous data exchange among all connected browser based clients. A virtual whiteboard surface is called *Panel* in our application. A panel can hold an arbitrary number of whiteboard elements, such as sticky notes or handwritings (see Figure 2). Users can modify those elements from all connected clients equally, regardless which user created them. Whenever an element is manipulated on a panel, the respective item is translated to its XML representation and transferred to the other participants in the session via XMPP messages. Receiving a modification message, the client updates its user interface accordingly [7]. Examples for such an XMPP message is shown in Listing 1. As Tele-Board is a real-time collaboration system, those XML based XMPP messages occur on every content modification resulting in lots of messages to be processed, e.g. when a sticky note is moved or scaled or a panel is scrolled as shown in Figure 3. Remote participants can see elements moving or the creation of handwritten notes to better keep track what is happening on the panel. This way, network message processing and UI rendering have to run in parallel in order to prevent interruption of user interaction when whiteboard content is synchronized simultaneously.

The traditional way of "simulating" parallel JavaScript tasks in the browser was the definition of callback functions that were put into browser's event queue with the help of the asynchronous `setTimeout()` and `setInterval()` methods [21]. The event queue is processed by browser's single-threaded event loop [22]. Every event and its associated function is processed completely until the next one can be executed. Neither two functions from the queue can run simultaneously nor a running function can be pre-empted giving an other function the chance to run [22], [23]. Hence, if a function takes too long to complete, the processing of the event queue is blocked resulting in an unresponsive website. The only way to avoid this would be to split up callback function code to multiple calls to `setTimeout()` method, transferring a task to application level that is normally better handled by the thread scheduler on operating system level.



<message to="566@conference.localhost" ... >
<message to="566@conference.localhost" ... >
<message to="566@conference.localhost" ... >
<message to="566@conference.localhost" ... >
<message to="566@conference.localhost" ... >
My idea on a sticky note

Figure 3. Each movement of a sticky note on a panel triggers an appropriate XML message. This way, a large number of synchronization messages have to be processed.

```
<message to="566@conference.localhost" type="
    groupchat" xmlns="jabber:client">
  <body>
    [[[[WHITEBOARD_SYNC_ALL]]]]
  </body>
  <properties xmlns="http://www.jivesoftware.com/xmlns/
      xmpp/properties">
    <property>
      <name>panelid</name>
      <value type="integer">566</value>
    </property>
    <property>
      <name>whiteboard_data</name>
      <value type="string"></value>
    </property>
  </properties>
</message>
```

Listing 1. An XMPP message sent to the collaboration server on session start requesting the state for panel with identifier "566".

JavaScript WebWorkers solve these problems as they are the first and only mechanism providing real multithreading in client side JavaScript applications. Unfortunately there are downsides associated with WebWorkers' share-nothing memory model that do not appear in the traditional approach: (1) the above mentioned native JavaScript API access limitations and (2) data exchange is only possible via message passing. The latter restriction has no great impact on our use case since networking and rendering are separated components. The necessary data interchange consists of plain JavaScript objects representing whiteboard elements that can be serialized to *JavaScript Object Notation* (JSON) [24] easily. The first limitation does apply to our application. These issues are summarized in Figure 4. In the following we propose a solution for our specific use case.

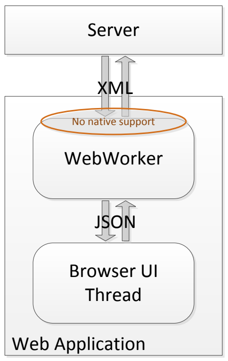


Figure 4. Message communication setup in our application. We focus on the missing native XML processing support within WebWorker scope.

The differences regarding JavaScript feature availability between WebWorkers and scripts directly associated with the web page are shown in detail in Figure 5. WebWorkers run in an different global context, the so-called `WorkerGlobalScope` [6] which differs from the current web site's window scope. Within this scope, there is no access to the `window` object and most of its properties and methods. Furthermore, some fundamental interfaces are not available, i.e. `Document`, `Element` and `Attr` (Attribute) [19], [21], [25]. As a consequence, native mechanisms for XML processing that rely on these interfaces are not available. In particular, the `DOMParser` interface method `parseFromString`, which parses an XML string and returns a `document` object, cannot be accessed. Another possibility for acquiring a parsed XML `document` object is to use `XMLHttpRequest` for *Asynchronous JavaScript and XML* (AJAX) functionality. Though the `XMLHttpRequest` interface is available within WebWorkers, its `responseXML` property, usually returning a `document` object, always returns `null` in a WebWorker scope [25].

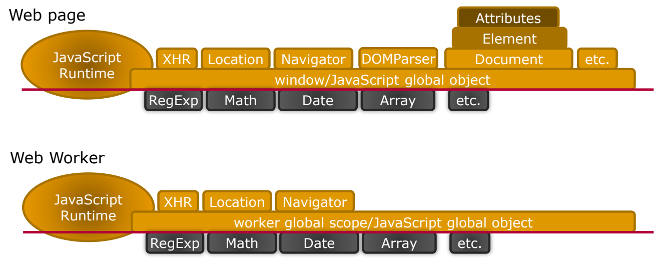


Figure 5. JavaScript features available to WebWorkers and scripts directly associated with the web page [21].

In order to create a networking component, that handles XML based XMPP messaging in our application within WebWorkers, we need two mechanisms to be implemented using JavaScript: (1) an XML parser and (2) classes implementing the missing `Document` and `Element` interfaces for enabling a transparent processing of parsed XML data by structures, which typically expect native parsing `document` object return values. We have implemented these mechanisms for our network component. A detailed description follows in the next chapter.

IV. System components implementation

The collaboration server synchronizes all connected browser based clients via XMPP messages. These XML strings have to be processed by the client application. For outgoing messages from the client to the server, whiteboard element data, which consists of plain JavaScript objects, can be assembled to XML strings by simple string concatenation operations. On the other hand, XML strings within incoming messages have to be decomposed, in order to extract the whiteboard element data. Since there is no possibility to use native XML parsing methods inside a WebWorker, accessible functionality is limited to string operations and regular expression API. Using string operations, such as concatenation or substitution for XML string analysis is cumbersome. Regular expressions ease this process. In our XML parser implementation, we use the regular expression provided in [20] to split a given XML string to a list of its markup and text items. Afterwards, a tree object structure is built upon these items. This step includes a verification whether the given XML string represents a well formed XML document, i.e. whether markup and text elements are nested correctly. Furthermore, our parser is namespace-aware, adding respective checks during the parsing process, e.g. correct element and attribute prefix to URI mapping.

The result of the parsing operation is a JavaScript tree object structure, implementing main parts of the W3C standard [19] core level 2 interfaces, in order to be seamlessly processed by existing, browser API relying components within WebWorker scope. This way, XML string documents are transformed to a JavaScript `Document` object. A class diagram of our DOM implementation is shown in Figure 6. During implementation, we focussed on elemental interfaces for processing common XML documents as these appear also in our XMPP networking use case. Interfaces for representing special XML elements, e.g. `ProcessingInstruction` are not implemented in the current version of our solution. According to the W3C standard, `Node` interface is the basis for other DOM elements, which inherit properties and methods from that interface. During XML string parsing, appropriate objects, e.g. `Element` or `Attr`, are created which represent XML elements and attributes in the document tree structure. The essential functionality for parsing XML strings and serializing DOM documents are encapsulated in the classes `Parser` and `Serializer`. These provide the same `parseFromString` and `serializeToString` methods as their web browser native counterparts. The `Document` class provides DOM `document` interface methods for creating, importing and accessing `Node` objects. The whole functionality is encapsulated in a single JavaScript file. It can be used by means of an `importScripts('XML.js')` statement at the top of a WebWorker script. In order to prevent overwriting global `window` and `document` objects when loaded within web page scope, we encapsulated our implementation in a global `XML` object. `XML` object's `makeGlobal` function provides global `window` and `document` objects making DOM API available in WebWorker scope.

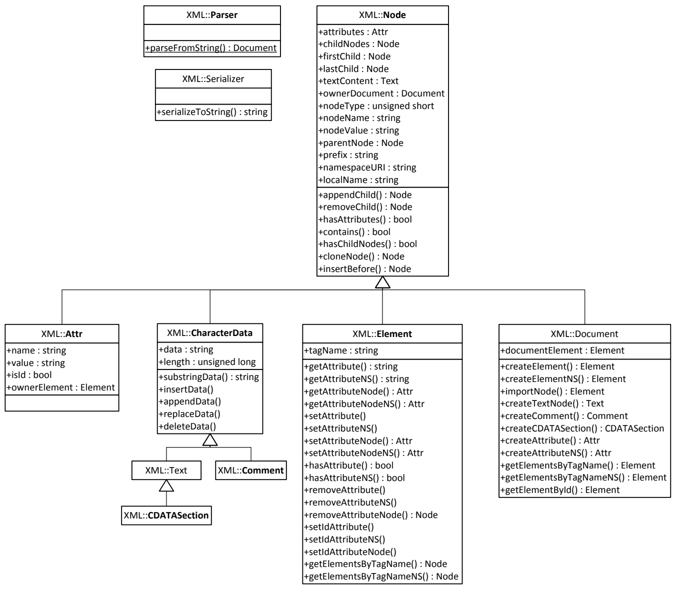


Figure 6. Class diagram of our solution. The implemented classes provide DOM interface features within `WorkerGlobalScope`.

With the help of our implementation, missing interfaces and functionality are transferred to WebWorker scope. The extended features providing their web page counterparts, now available within `WorkerGlobalScope`, are shown in Figure 7. XML processing libraries requiring native parser APIs can now be used within WebWorkers, which was not possible before. In our networking scenario, we use the XMPP library *Strophe.js*. On the basis of our implementation, the library can be utilized within a WebWorker.

V. Integration of Existing XMPP Library

Within the Tele-Board software system the Openfire[8] XMPP server component coordinates all communication among the connected browser based clients. Strophe.js is a JavaScript library that implements the XMPP communication protocol. The library supports TCP based WebSocket [26] as well as HTTP based Bidirectional-streams Over Synchronous HTTP (BOSH) [27] protocols. Openfire server offers an HTTP binding that allows clients using the HTTP protocol to connect to the server over BOSH.

Strophe.js version 1.1.3 uses native browser XML parsing. Therefore, it cannot be used in a WebWorker scope. With the help of our mentioned DOM implementation, the library also works inside a WebWorker. It requires (1) a `document` object for creating XML elements and (2) XML parsing functionality. Since our solution provides both with a standards compliant API, Strophe.js library can be used without any adaptions.

[8]http://igniterealtime.org/projects/openfire/

```
fixXHR = function () {
 var parser;
 // overwrite XMLHttpRequest and its responseXML
     property
 if (XMLHttpRequest) {
  parser = new XML.Parser();
  XMLHttpRequest = (function (origXHR) {
    return function () {
      var newXHR = new origXHR();
      Object.defineProperty(newXHR, "responseXML", {
        enumerable: true,
        get: function () {
          if (this.responseText) {
            return parser.parseFromString(this.
                responseText);
          }
          return null;
        }
      });

      return newXHR;
    };
  })(XMLHttpRequest));
 }
};
```

Listing 2. Changed `XMLHttpRequest` object's `responseXML` property. It returns a parsed XML document on the basis of the object's `responseText` property.

Within the library, two methods make direct use of our solution. The `_makeGenerator()` method provides a DOM `document` object, which is used by the library to create XML element objects. That object, which is normally not available within WebWorkers, is provided by our JavaScript DOM implementation. Since the `Document` object, as well as the objects (e.g. `Node` and `Text`) created by its methods, provide native API's method and property signatures, our DOM XML implementation and native API components can be used interchangeably. The second method to be considered is Strophe's `getResponse()` method, where incoming XMPP messages are parsed. HTTP based BOSH communication is handled by Strophe with the help of AJAX. In order to parse incoming message data, `XMLHttpRequest` object's `responseXML` property is used. As mentioned above, the value of this property always returns `null` within `WorkerGlobalScope`. Hence, `XMLHttpRequest` object's behavior has to be changed while preserving its native functionality. Within our solution, we overwrite `responseXML` property with the help of a `get` function as shown in Listing 2. The self executing function stores the original `XMLHttpRequest` object in a closure. Afterwards, `XMLHttpRequest` is assigned a new function, which, when called as a constructor, creates an instance of the original `XMLHttpRequest` stored in closure variable `origXHR` and defines a `responseXML` property to return a parsed XML document instead of `null`. To overwrite the property, an explicit call to our `XML.makeGlobal()` method is necessary, which in turn executes the shown internal `fixXHR()` function.

The Strophe source file can be deployed without any adaptions together with our XML DOM implementation via an `importScripts('strophe.js', 'XML.js')` statement in our network component running as a WebWorker.

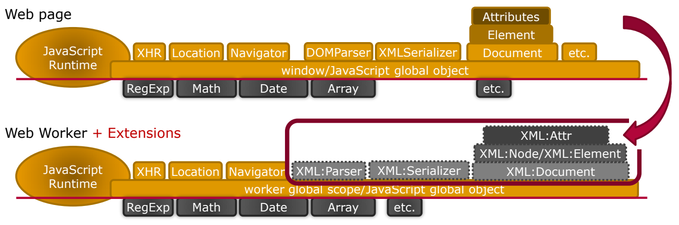


Figure 7. `WorkerGlobalScope` extended with our DOM interface implementations. DOM interface relying components can now be used inside of WebWorkers.

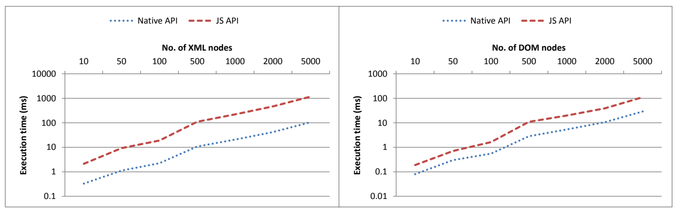


Figure 8. Performance of XML parsing and serialization in desktop Chrome browser. Native as well as custom APIs have linear execution time complexity.

VI. Evaluation of the Implementation's Performance

Our described implementation of an XML parsing/serialization and DOM API for replacing non-existent native functionality within WebWorker scope is written in JavaScript. Hence, it is not platform specific compiled code and therefore inevitable slower than native code. In this chapter, we elaborate our solution's performance compared to web browser's native API functions.

We measure execution times of the main functionality used in our application, which are XML parsing and serialization and DOM navigation in dependence on the number of XML respectively DOM nodes to be processed. For that, we compare the performance of `XML.Parser.parseFromString()`, `XML.XMLDocument.serializeToString()` and `XML.XMLElement.getElementsByTagName()` methods with their native counterparts `window.DOMParser.parseFromString()`, `window.XMLSerializer.serializeToString()` and `window.document.getElementsByTagName()`.

In our test setup, we use the following web browsers:

- Chrome 38
- Firefox 32
- Internet Explorer 11

Tests are run on two devices:

- Desktop PC
  - CPU: Corei5 750@2.66 GHz
  - RAM: 6GB
  - Operating System: Windows 7 Enterprise SP1 x64
- Tablet device Asus Nexus 7 (2012)
  - CPU: Nvidia Tegra 3@1.3 GHz
  - RAM: 1GB
  - Operating System: Android 4.4.4

The results of XML parsing and document serialization are shown in Tables I and II. Depending on the number of XML and DOM nodes, execution time increases linear in our solution and in native API functions. This can be seen in all tested browsers. As we expected, application runs faster on desktop hardware. The native performance speedup is about 10. During the tests with Internet Explorer 11, the measured values strongly varied as indicated by the high standard deviation. We did not figure out why this happens especially in this browser. We saw similar values also on other machines. Nevertheless, we chose to include the measured values in order to show their linear course, which is similar to the results of the other browsers. For better visualization, the XML parsing and serialization results for Windows 7 Chrome browser are shown in Figure 8. In both cases, execution time complexity is linear.

The `getElementsByTagName()` operation performance differs between native API and our JavaScript solution as it is shown in Table III. The native API provides constant time access to DOM nodes regardless the number of contained nodes in the document. It is likely that the native DOM utilizes mechanisms like HashMaps to assign actual DOM nodes to tag names and therefore provide constant time access. In our approach we decided to prefer a fast parsing and DOM creation, since the usage of a caching mechanism slowed down our parsing performance. The `getElementsByTagName` operation must provide a list of element nodes in document order, i.e. in order of the occurrence of their start-tag in the XML [19]. This corresponds to a pre-order depth-first traversal of the DOM tree structure. The effort for keeping this list consistent, grows with the document size. The `appendChild()` operation for inserting a DOM node, therefore takes more time in larger documents. Measuring `appendChild()` operation in desktop Chrome browser, we could confirm this behavior. The operation's time consumption increases linear with the number of nodes in the document. In our solution, the `appendChild()` operation takes constant time for inserting a new DOM node. In return, the implementation of our `getElementsByTagName()` operation has a linear execution time complexity, since there is no caching mechanism. In our networking scenario there are many short XML messages (see Figure 1) to process. Hence, we preferred a fast XML parsing, since in our use case it is more important than later constant time DOM node retrieval.

VII. Conclusion

Our proposed implementation covers a common use case of running network communication and XML processing in parallel to the web browsers main thread. This is not possible with standard native browser instruments due to WebWorker's limitations. Providing standards compliant APIs, we could

TABLE I
XML PARSING EXECUTION TIMES (MILISECONDS) AND STANDARD DEVIATION OF OUR JAVASCRIPT BASED SOLUTION (JS) COMPARED TO NATIVE BROWSER API (NATIVE).

	Windows 7x64												Android 4.4.4.			
	Firefox 32				Chrome 38				Internet Explorer 11				Chrome 38			
#Nodes	Native (ms)	SD	JS (ms)	SD	Native (ms)	SD	JS (ms)	SD	Native (ms)	SD	JS (ms)	SD	Native (ms)	SD	JS (ms)	SD
10	0.48	0.04	7.29	2.74	0.33	0.11	2.13	0.19	3.18	7.18	3.37	3.89	2.58	0.62	25.47	27.72
50	1.62	0.07	23.87	1.82	1.13	0.05	9.29	0.27	2.65	0.12	7.79	0.38	9.47	3.66	73.69	20.74
100	3.46	0.14	47.51	4.75	2.24	0.11	18.89	1.11	4.94	0.33	15.81	4.21	16.23	0.39	130.59	5.96
500	17.85	1.11	238.62	13.12	10.84	0.68	110.45	10.67	24.34	1.41	90.1	19.17	76.41	0.86	760.5	45.56
1000	37.82	1.31	457.98	10.1	20.8	0.71	222.61	10.02	47.87	1.55	202.9	37.81	154.5	5.98	1557.12	103.33
2000	85.89	1.82	943.23	27.01	41.9	1.94	465.63	44.62	97.6	2.24	530.62	44.55	311.66	12.3	3126.19	174.82
5000	296.13	3.05	2511.04	49.57	106.11	4.05	1170.76	70.29	246.97	7.49	2393.83	159.27	776.44	23.8	8048.98	428.71

TABLE II
DOCUMENT SERIALIZATION EXECUTION TIMES (MILISECONDS) AND STANDARD DEVIATION OF OUR JAVASCRIPT BASED SOLUTION (JS) COMPARED TO NATIVE BROWSER API (NATIVE).

	Windows 7x64												Android 4.4.4.			
	Firefox 32				Chrome 38				Internet Explorer 11				Chrome 38			
#Nodes	Native (ms)	SD	JS (ms)	SD	Native (ms)	SD	JS (ms)	SD	Native (ms)	SD	JS (ms)	SD	Native (ms)	SD	JS (ms)	SD
10	0.16	0.02	4.54	11.39	0.08	0.01	0.19	0.01	0.29	0.15	0.52	0.36	0.49	0.18	1.17	0.16
50	0.59	0.03	2.21	0.11	0.3	0.02	0.71	0.02	1.07	0.35	1.51	1.45	1.8	0.07	5.46	0.14
100	1.14	0.09	4.38	0.25	0.55	0.03	1.63	0.47	1.88	0.83	1.41	0.05	3.5	0.12	12.24	3.61
500	5.77	0.48	26.14	4.76	2.83	0.37	10.98	3.93	9.83	4.01	20.87	35.04	16.05	0.38	62.25	14.04
1000	11.4	0.64	46.82	1.11	5.38	0.15	19.88	2.38	33.73	43.06	20.11	2.05	31.27	0.37	116.61	7.51
2000	22.04	0.61	101.22	17.9	10.65	0.17	39.06	3.33	72.15	65.04	58.2	56.86	64.6	0.6	259.42	15.32
5000	57.33	1.38	249.22	29.34	29.06	1.03	110.86	16.2	149.86	103.44	237.05	131.78	179.21	9.79	722.9	162.42

TABLE III
DOM NODE RETRIEVAL EXECUTION TIMES AND STANDARD DEVIATION OF OUR JAVASCRIPT BASED SOLUTION (JS) COMPARED TO NATIVE BROWSER API (NATIVE).

	Windows 7x64												Android 4.4.4.			
	Firefox 32				Chrome 38				Internet Explorer 11				Chrome 38			
#Nodes	Native (ms)	SD	JS (ms)	SD	Native (ms)	SD	JS (ms)	SD	Native (ms)	SD	JS (ms)	SD	Native (ms)	SD	JS (ms)	SD
10	0.01	0.01	0.15	0.3	0	0	0.01	0	0.01	0.02	0.05	0.01	0.02	0.02	0.08	0.02
50	0	0	0.19	0.16	0	0	0.06	0.01	0.01	0.01	0.09	0.01	0.01	0.01	0.41	0.03
100	0	0	0.12	0.01	0	0	0.14	0.01	0.01	0.01	0.25	0.02	0.01	0.01	0.94	0.12
500	0	0	2.44	0.24	0	0	2.53	3.25	0.02	0.04	3.88	0.22	0.02	0	10.67	7.92
1000	0.01	0	12.02	9.5	0.01	0	15.41	25.8	0.05	0.12	28.34	42.38	0.02	0	24.04	9.06
2000	0.01	0	47.87	36.19	0.01	0	10.42	4.93	0.07	0.17	45.41	2.6	0.02	0	60.43	16.4
5000	0.01	0	294.28	49.39	0.01	0	44.65	3.18	0.22	0.63	463.84	63.6	0.02	0	278.29	18.24

use a JavaScript XMPP library (Strophe.js), relying on native browser APIs, within an OS level thread spawning WebWorker script. In performance tests, we could show that the execution time complexity of our JavaScript XML processing implementation has the same linear course as corresponding native browser API functions.

Currently, the presented results show the performance of our implementation in a limited test case. Further tests have yet to reveal the impact of the proposed approach in a collaborative environment. Furthermore, the proposed approach is limited to APIs that can be replaced by implementations using the standard JavaScript APIs available in WebWorker scope. Restrictions remain in cases when access to hardware resources have to be provided, e.g. persistent data storage supported by Local Storage API.

Overall, we show that it is possible to distribute application tasks among separate threads within web browsers, utilizing today's common multicore processors more effectively. The application range of this approach is not limited to our use case of XML processing, but can be broadened by other fields, e.g. image processing. This way, our approach helps to parallelize JavaScript applications by providing APIs to WebWorkers, aided by application level implementations. The condition is thus created for additional libraries that rely on native interfaces to be moved to a dedicated thread.

ACKNOWLEDGMENTS

We thank the HPI-Stanford Design Thinking Research Program for funding and supporting this project. The authors are especially grateful to Lutz Gericke for giving his valuable feedback.

REFERENCES

[1] M. Anttonen, A. Salminen, T. Mikkonen, and A. Taivalsaari, "Transforming the web into a real application platform: New technologies, emerging trends and missing pieces," in *Proceedings of the 2011 ACM Symposium on Applied Computing*, ser. SAC '11. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/1982185.1982357

[2] P. Bright, "Surprise! Mozilla can produce near-native performance on the Web," http://arstechnica.com/information-technology/2013/05/native-level-performance-on-the-web-a-brief-examination-of-asm-js/, May 2013, Ars Technica.

[3] A. Zakai and R. Nyman, "Gap between asm.js and native performance gets even narrower with float32 optimizations," https://hacks.mozilla.org/2013/12/gap-between-asm-js-and-native-performance-gets-even-narrower-with-float32-optimizations/, December 2013, Mozilla Hacks - the Web developer blog.

[4] D. Flanagan, *JavaScript: The Definitive Guide, Sixth Edition*. Beijing; Sebastopol, CA: O'Reilly, 2011.

[5] C. Hewitt, P. Bishop, and R. Steiger, "A Universal Modular ACTOR Formalism for Artificial Intelligence," in *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, ser. IJCAI'73. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1973, pp. 235–245. [Online]. Available: http://dl.acm.org/citation.cfm?id=1624775.1624804

[6] I. Hickson, "Web Workers," World Wide Web Consortium (W3C), Candidate Recommendation, May 2012, http://www.w3.org/TR/2012/CR-workers-20120501/.

[7] R. Gumienny, L. Gericke, M. Quasthoff, C. Willems, and C. Meinel, "Tele-Board: Enabling Efficient Collaboration In Digital Design Spaces," in *Proceedings of the 15th International Conference on Computer Supported Cooperative Work in Design (CSCWD 2011)*. IEEE Press, June 2011, pp. 47–54.

[8] M. Wenzel, L. Gericke, R. Gumienny, and C. Meinel, "Towards Cross-Platform Collaboration - Transferring Real-Time Groupware To The Browser," in *Proceedings of the 17th IEEE International Conference on Computer Supported Cooperative Work in Design (CSCWD 2013)*. IEEE, June 2013, pp. 49–54.

[9] P. Saint-Andre, *RFC 6120 - Extensible Messaging and Presence Protocol (XMPP): Core*, Internet Engineering Task Force (IETF), March 2011. [Online]. Available: http://tools.ietf.org/html/rfc6120

[10] L. Gericke and C. Meinel, "Evaluating an Instant Messaging Protocol for Digital Whiteboard Applications," in *Proceedings of the 2011 International Conference on Internet Computing (ICOMP 2011)*. CSREA Press, July 2011, pp. 3–9.

[11] A. Taivalsaari, T. Mikkonen, M. Anttonen, and A. Salminen, "The Death of Binary Software: End User Software Moves to the Web," in *Proceedings of the 9th International Conference on Creating, Connecting and Collaborating through Computing*, ser. C5'2011. IEEE, January 2011, pp. 17–23. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5936687

[12] M. Mehrara and S. Mahlke, "Dynamically Accelerating Client-side Web Applications Through Decoupled Execution," in *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 74–84. [Online]. Available: http://dl.acm.org/citation.cfm?id=2190025.2190055

[13] A. Welc, R. L. Hudson, T. Shpeisman, and A.-R. Adl-Tabatabai, "Generic workers: towards unified distributed and parallel JavaScript programming model," in *Programming Support Innovations for Emerging Distributed Applications on - PSI EtA '10*. New York, New York, USA: ACM Press, 2010, pp. 1–5. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1940747.1940748

[14] S. Herhut, R. L. Hudson, T. Shpeisman, and J. Sreeram, "River Trail: A Path to Parallelism in JavaScript," in *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications - OOPSLA '13*. New York, New York, USA: ACM Press, 2013, pp. 729–744. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2509136.2509516

[15] D. Bonetta, W. Binder, and C. Pautasso, "Tigerquoll: Parallel event-based javascript," *SIGPLAN Not.*, vol. 48, no. 8, pp. 251–260, Feb. 2013. [Online]. Available: http://doi.acm.org/10.1145/2517327.2442541

[16] "OpenCL - The open standard for parallel programming of heterogeneous systems," http://www.khronos.org/opencl/, July 2014, Khronos Group.

[17] A. Erbad, N. C. Hutchinson, and C. Krasic, "DOHA: Scalable Real-timeWeb Applications through Adaptive Concurrent Execution," in *Proceedings of the 21st international conference on World Wide Web - WWW '12*. New York, New York, USA: ACM Press, 2012. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2187836.2187859

[18] T. Leithead, "DOM Parsing and Serialization - DOMParser, XMLSerializer, innerHTML, and similar APIs," World Wide Web Consortium (W3C), Candidate Recommendation, June 2014, http://www.w3.org/TR/2014/CR-DOM-Parsing-20140617/.

[19] A. Le Hors, P. Le Hgaret, L. Wood, G. Nicol, J. Robie, M. Champion, and S. Byrne, "Document Object Model (DOM) Level 2 Core Specification," World Wide Web Consortium (W3C), W3C Recommendation, November 2000, http://www.w3.org/TR/DOM-Level-2-Core/.

[20] R. D. Cameron, "REX: XML Shallow Parsing with Regular Expressions," *Markup Languages*, vol. 1, no. 3, pp. 61–88, 1999. [Online]. Available: http://www.cs.sfu.ca/~cameron/REX.html

[21] D. Rousset, "Introduction to HTML5 Web Workers: The JavaScript Multi-threading Approach," http://msdn.microsoft.com/en-us/hh549259, July 2011, Microsoft Developer Network.

[22] J. Resig and B. Bibeault, *Secrets of the JavaScript Ninja*. Shelter Island, NY 11964: Manning Publications Co., 2013.

[23] "Concurrency model and Event Loop," https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/EventLoop, July 2014, Mozilla Developer Network.

[24] T. Bray, *RFC 7159 - The JavaScript Object Notation (JSON) Data Interchange Format*, Internet Engineering Task Force (IETF), March 2014. [Online]. Available: http://tools.ietf.org/html/rfc7159

[25] "Functions and classes available to workers," https://developer.mozilla.org/en-US/docs/Web/API/Worker/Functions_and_classes_available_to_workers, July 2014, Mozilla Developer Network.

[26] I. Fette and A. Melnikov, *RFC 6455 - The WebSocket Protocol*, Internet Engineering Task Force (IETF), December 2011. [Online]. Available: http://tools.ietf.org/html/rfc6455

[27] I. Paterson, D. Smith, P. Saint-Andre, J. Moffitt, L. Stout, and W. Tilanus, "Bidirectional-streams Over Synchronous HTTP (BOSH)," XMPP Standards Foundation, Draft Standard 1.11, April 2014, http://xmpp.org/extensions/xep-0124.html.