Journal of Web Engineering, Vol. 9, No. 3 (2010) 207–226

PARTITIONING WEB APPLICATIONS BETWEEN THE SERVER AND THE CLIENT

JANNE KUUSKERI
Department of Software Systems, Tampere University of Technology, P.O. Box 553
Tampere, 33103, Finland
janne.kuuskeri@tut.fi

TOMMI MIKKONEN
Department of Software Systems, Tampere University of Technology, P.O. Box 553
Tampere, 33103, Finland
tommi.mikkonen@tut.fi

Received June 21, 2009
Revised January 14, 2010

Web 2.0 and rich Internet application technologies are offering more and more sophisticated means for building compelling applications. At the same time the development of applications is becoming increasingly complex. While web applications are commonly relying on server side processing, we aim at implementing a "fat client" and running applications mostly on the client. With this in mind we derive a set of guidelines on how the applications should be partitioned between the server and the client. By following these directives and leaning on the traditional principles of good software development, we address the issues of complexity that have lately emerged in web development.

Keywords: Web Application, AJAX, JavaScript, Comet

1 Introduction

Web application development is in the middle of a paradigm shift. Users are getting used to web applications with dynamic content and enhanced user experience. User interfaces are no longer updated the whole screen at a time, and servers are able to feed data to them spontaneously. From the users' point of view, web applications are thus becoming more and more like traditional desktop applications.

While the user interfaces of web applications are becoming more usable, the underlying standards and protocols are not evolving at the same pace. Existing technologies are thus pushed to their limits, and consequently web applications become more and more complex to build [1]. There are hundreds of toolkits and frameworks that alleviate the burden for the developers. Still, there is considerable impedance mismatch between HTTP's request/response model and the way user interfaces are implemented in desktop applications.

Most of the current web frameworks focus on simplifying the implementation and maintenance of web applications by providing a set of tools and conventions for the developers. This helps developers to organize and deploy their source code, binaries, and resource files, but they do not really improve the technology behind them. The produced layout of the applica-

tion is still pretty much the same had it been done manually. Applications are still rendered on the screen using HTML [2], CSS [3], and JavaScript [4], while the back-end on the server provides the main processing facilities by handling all the requests and generating responses in HTML, JSON [5], or XML [6]. However, well-established conventions for partitioning the application logic to the client and the server are still missing, apart from the approach where everything is run on the server and the client simply renders data.

We start with a premise that in order to lower the boundary between the server and the client, a complex web application should be built using fewer technologies and a single programming language. This allows the execution of the same code on the server and on the client, which in turn leads to uniform development experience. Applications should be run mostly on the client but using the server for persistence and networking, such as collaborating with other clients. Applications will still need the server but far less. Moreover, with the common code base for the client and the server the development cycle becomes much closer to the traditional software development. However, it is also possible to balance performance with relative ease, as only deployment must be revisited, not the whole implementation.

In this paper we focus on how to partition web applications between the server and the client. This covers both conceptual and technical issues. We present guidelines for deciding which parts of an application should be run on the client and which parts on the server. We also offer techniques for efficiently implementing the communication between the two. Security issues are intentionally left outside the scope of this paper.

This paper is structured in the following fashion. In Section 2 we give an overview of the current status of asynchronous features of web applications and the technologies behind them. Key principles and an overview of our approach are given in Section 3. This is followed by a more detailed description of the implementation in Section 4 and an example in Section 5. In Section 6 we present products and technologies related to ours and in Section 7 we provide some final conclusions to our paper.

2 Background and concepts

Traditionally web applications have been implemented using a synchronous approach. Synchronous user interfaces issue a request and freeze until the data eventually arrives. When the data is received, the whole page is reloaded and displayed on the screen in full. However, asynchronous user interfaces have been getting a lot of attention lately. Due to its presence in all major browsers, JavaScript is the most often used technology for asynchronous user interfaces.

2.1 Asynchronous technologies

Much of the Web 2.0 advent is due to wide adaptation of asynchronous technologies in client server communication. Clients request data from servers asynchronously and without halting the user interface. Once the response arrives, the client can update only the parts of the page that really pertain to the data. Asynchronous techniques may also be applied the other way around by feeding data spontaneously to the client. However, this is much less known and utilized form of the web communication.

Ajax. Asynchronous technologies for the web have been around for years but only in the recent years their popularity has grown exponentially. At present, Ajax [7] is the dominant

technique for implementing asynchronous services. It stands for *Asynchronous JavaScript and XML*, and as the name suggests, it enables dynamic page updates in web applications [8]. However, Ajax is not a single technology but more like an umbrella term covering several existing technologies and the way they are used together to produce dynamic content on web pages. Ajax, like JavaScript itself, suffers from browser incompatibilities. Many toolkits and libraries have emerged to make asynchronous calls to server easier, including for instance The Dojo Toolkit (http://www.dojotoolkit.org/), jQuery (http://jquery.com/) and The Yahoo! User Interface Library (http://developer.yahoo.com/yui/).

HTTP push. In the World Wide Web, clients and servers communicate using HTTP, which is all about requests and responses; if the client wants something from the server, it has to explicitly request it. HTTP push is not part of the protocol, hence there is no standard way of sending spontaneous messages from the server to the client. In order to bypass this limitation, several techniques have been applied over the years. In 2006 Alex Russell coined the term *Comet* to cover all HTTP push techniques [9]. Most popular Comet techniques are *forever frame* and *long-polling* [10]. The forever frame is based on an `IFrame` tag that has infinite length and thus is always loading. When events occur on the server, new script tags are fed into the `IFrame` for the browser to execute. The long-polling is an Ajax-request that is left open until there is new data for the client. When the server sends the data, the client immediately issues another request. Although all variations of Comet work – some admittedly better than others – they still are extensions to HTTP/HTML and are therefore more or less hacks. HTML 5 is set out to fix all this by defining *server-sent DOM events* in the form of `event-source` element [14], but we are far away from the point where all major web browsers support this feature.

2.2 JavaScript

JavaScript was originally developed to allow client side scripting of HTML pages. For a long time, that is exactly what it was; small snippets of JavaScript code embedded into HTML pages to create content that is more dynamic and responsive. Since then, JavaScript has spread from the browser into various environments.

2.2.1 Client side JavaScript

JavaScript's seamless integration with the Document Object Model (DOM) [15] made it very popular in its early days. The DOM is an object model for representing HTML documents, and it allows programmers to modify web pages from their scripts implemented in JavaScript. Also on all major browsers, JavaScript is the only programming language that can be used without plugins. All other technologies, such as Adobe Flash, Java Applets, and Microsoft Silverlight, require users to install corresponding plugins for their web browsers. Unfortunately different browsers implement different versions of the DOM and, to make things worse, different versions of the JavaScript language itself. This has been a source of grief for many web developers and has spawned multitude of third party JavaScript libraries that try to hide the differences between different JavaScript and DOM implementations.

2.2.2 Server side JavaScript

One of the environments that JavaScript has been brought into, is *Server Side JavaScript*. In the absence of DOM, these host environments always provide some other object model

to access their services. There are some server side JavaScript engines but so far they have not gained much momentum in the perspective of all the existing web frameworks out there. SpiderMonkey [16] and Rhino [17] are two commonly used server side JavaScript engines. SpiderMonkey is also the first ever implementation of JavaScript. It is most notably used in the Mozilla Firefox browser but also many of the JavaScript enabled servers embed it as their JavaScript engine. SpiderMonkey is written in C and is meant to be embedded as part of a host application. Rhino, on the other hand, is written entirely in Java. This means that Java libraries can be called from JavaScript and vice versa, thus bringing the vast collection of existing Java libraries into the hands of JavaScript developers.

One of the biggest inconveniences in server side JavaScript development is the lack of standard libraries and ecosystem that programming languages usually have. Essentially this means that access to system resources such as file system, network interface and even other JavaScript libraries are always vendor specific. So, a JavaScript program written in Rhino environment has little chance of running successfully in say, SpiderMonkey. Lately, there has been a good work to address this issue by the CommonJS (http://wiki.commonjs.org/wiki/CommonJS) group. Their goal is to make JavaScript more accessible for server side development by standardizing and implementing all the standard libraries that programming languages usually have. These include interfaces for file access, sockets, threading and many others. Eventually these interfaces would then be implemented by all JavaScript engines. While this is very valuable work and will pave the way for JavaScript to enter the server realm, it is still very much work in progress and at the time of writing this, very few interfaces have been agreed upon, let alone implemented.

3 Our approach

Our assumption is that when building a complex web application with a lot of user interactivity, it is best to introduce a fat client. By doing so, the responsiveness and robustness of the user interface can be increased significantly because the client can handle as many user events as possible. Furthermore, it makes the development much more straightforward. This can be achieved by choosing a common language for the whole application, partitioning it wisely and then implementing a transparent communication gateway between the server and the client. Moreover, by extending the use of JavaScript to the server makes the development experience more natural. Ideally, for a given web application, the web server could only serve a single entry page in HTML and everything else after that would be handled by the two JavaScript programs: one on the client and another on the server. There is no point in building a complete web server in JavaScript because some kind of JavaScript engine, written in another language, is in any case needed. Therefore, we should use an existing web server, build a small wrapper on top of it and then write all the request handlers in JavaScript.

3.1 Guidelines for client-server partitioning

We start with an assumption, unusual in the context of the web, that by default, everything possible should be executed on the client and only the parts the client cannot handle are left for the server. Technically, it is fairly easy to see which parts must be executed on the server but that is not always the whole picture. Depending on the application, there might be some usability factors that make certain operations more suitable to be run on the server.

3.1.1 Mandatory for the server

In web applications, there are always certain operations that are dependent upon remote resources and thus cannot be executed without the server. Granted, with the help of tools like Google Gears (http://gears.google.com/), these operations can be built to run in offline mode, which is certainly a step forward, but for many applications this is not enough. In Google Gears, web applications become usable even without access to network but depending on the application they also run in more or less restricted mode. In the following, we will address the most common server responsibilities.

Persistence refers to the data that is not erased at the termination of the program that created it. Persistence has to be handled by the server because applications running in web browsers are not allowed to access local resources such as the file system or database. For example, if a web application needs to do database lookups, reading a file or storing user's data into database, it will have to do it on the server.

Client-client communication takes place between one or more web applications running on different clients. JavaScript follows the *same-origin* policy, which prevents clients from connecting to any other peers besides the server they were originally loaded from [18]. By allowing networking via the originating server, we make it possible for the applications to load data from remote servers or develop collaborative applications such as chat rooms.

Shared data means data that is accessible to more than one client simultaneously. As mentioned earlier, the same origin policy prevents data sharing to be implemented peer to peer. However, the server can be used to provide a centralized and shared database among all clients.

Scheduled tasks and tasks that keep on running even after the user has logged off need to be executed on the server. For instance, web application providing administrative interface to an underlying system could leave some scheduled tasks running on the server.

Intellectual property such as secret algorithms and estimating functions must run on the server in order to preserve the integrity of the business sensitive information. While JavaScript code can be obfuscated to make the code practically unreadable to human eyes, there are tools to reverse the obfuscation and make it readable again.

3.1.2 Considerations for the server

Some things may be worth running on the server even if they could be run on the client. While we prefer the client to host the following operations, we have to take into consideration the nature of the application at hand and its targeted users.

Browser navigation happens using browser's back and forward buttons. People are used to traditional web applications where they can safely navigate back and forth using the browser. This is lost if the whole web application is implemented as one dynamic page. One solution is to compromise and create a hybrid of these two approaches where the application is divided into few dynamic pages and each of these pages comprises one "screen" of the application.

Heavy calculations or other CPU-intensive operations may be better run on the server where we know the hardware and how to best harness it. Also on the server, there's a possibility to use more computationally efficient language than JavaScript, say C++, to do the actual number crunching. Of course, this decision is very specific to the application in

question and in some cases it might be even useful to distribute these calculations over many clients.

Reliability can be increased by running operations on the server. For example, users may close their web browsers in the middle of the operation, or operations depending on the current time may behave oddly if the clock on the client is running late.

3.2 Guidelines for client-server communication

In order to maintain application's responsiveness, the communication should be minimized, but when communication is needed it should be asynchronous. For the client, this means using Ajax for all the requests to the server. For the server, long-polling Comet, which is also implemented using Ajax requests, is the way to go. But this is only the end user's perspective, we will have to bear in mind the developers' usability as well. To lower the boundary between the server and the client, functions provided by the server should be callable from the client and the other way around. To accomplish this we provide proxy functions that seamlessly take care of the communication.

In long-polling Comet, the client maintains one pending `XMLHttpRequest` open at all times. This can be started when the page loads and is then left open for server's spontaneous messages. The connection is closed when a message from the server is received or connection timeout occurs. After a lost connection, the client is responsible for immediately issuing another pending request.

Continuously keeping one connection open raises some issues that should be taken into account when implementing the application. Browsers tend to restrict the number of simultaneous HTTP connections, so some kind of multiplexing is required if there is more than one listener on the client waiting for events from server. Server's main concern is the number of threads it uses if there are a lot of concurrent clients. Some web servers, such as Jetty, are already customized for Comet, and they do not create a separate thread for each pending request.

4 Implementation

To prove the concepts presented so far, we next describe some of the implementation details. First, in order to enforce reusability, we will divide the implementation into two parts: the platform and the web application. The platform is a generic place for hosting JavaScript applications. On the server, it integrates tightly with the web server and on the client side it sits between the browser and the web application. We use Java platform and a Java based server along with Rhino JavaScript engine. There is a good number of Java based web servers implementing Comet available, and we choose Jetty for it's high embeddability and easy configuration. Figure 1 gives an overview of the system.

In the following subsections we will give an overview of different components involved in the implementation. We will start by addressing some of the challenges faced with when doing server side JavaScript. Then we move on into Jetty specific stuff and see how it is embedded and bootstrapped. Finally, we look at some of the implementation details of the platform itself.

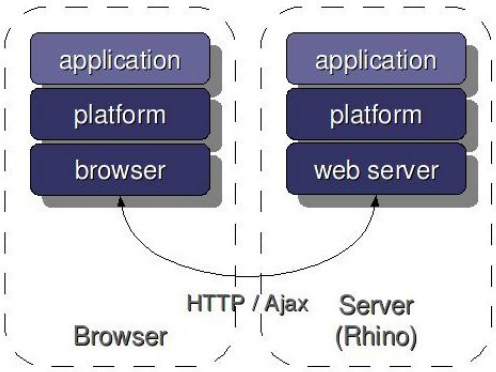

Fig. 1. System overview

4.1 Rhino

Rhino is a pure Java implementation of the JavaScript interpreter and runtime. It is an open source project that is maintained by the Mozilla Foundation. Rhino executes JavaScript code by first compiling it to Java classes and then running the compiled code within the Java Virtual Machine. For the JavaScript program this means that it can make use of all the existing Java libraries and for the Java program this means that it can harness the flexibility of JavaScript by, for example providing scriptable extension points to the Java application. It should be noted that the Rhino JavaScript engine does not implement the DOM or any other browser objects and it is thus meant to be run outside of browser.

4.2 Introducing Server Side JavaScript

As mentioned earlier, server side JavaScript lacks the standard libraries and the ecosystem programming languages usually have. The focus of almost all existing JavaScript libraries is on how to make web development easier on the browser. Therefore they are also dependent upon the browser and for the same reason, useless on the server. Nevertheless, in order to enforce code reusability and other good programming practices we need some basic elements such as modules, logging facility and maybe a test framework. In our implementation we have followed the discussion and decisions outlined in the ServerJS group and filled in the missing pieces from our own experiences.

Because there are no modules or namespaces in JavaScript, all top level variables and functions go into the global namespace. Also, the language itself does not provide any mechanism for importing other JavaScript libraries or modules. This makes modules one of the most missed feature in JavaScript. It is so essential part of a programming language that we practically need them before anything else. Furthermore, it would be nice to be able to keep functions that are only used by the module internally, from cluttering the global namespace. Bearing in mind these requirements, the Listings 1 and 2 give an overview of the usage of modules and later on the Listing 3 shows the implementation of the modules in server side JavaScript.

```
// idgenerator.js - idgenerator module

// visible to other modules
exports.getId = function (key) {
    if (!keys.hasOwnProperty(key)) {
        keys[key] = 0;
    }
    return nextId(key);
};

// not visible to other modules
var keys = {};

// not visible to other modules
var nextId = function (key) {
    return keys[key]+=1;
};
```

Listing 1 Module implementation

```
// db.js - database module

var id = require('idgenerator');
print(id.getId('jedi'));
```

Listing 2 Module usage

```
/// evals the loaded module
var __eval__ = function (code, exports) {
    eval(code);
};

/// returns the global require function
require = function () {

    var moduleExtension = '.js', loadedModules = {};

    /// the global require function
    return function (module) {
        var exports_, module_code, evaled_string, module_file;

        module_file = module + moduleExtension;
        if (!loadedModules.hasOwnProperty(module)) {
            // load the module
            exports_ = {};
            module_code = readFile(module_file);
            evaled_string = '(function (exports){\n' + module_code +
                '\n}).call(exports, exports);';
            // step out of the lexical scope while doing the eval
            __eval__(evaled_string, exports_, module);
            loadedModules[module] = exports_;
        }
        return loadedModules[module];
    };
}();
```

Listing 3 Module implementation

By default, all top level variables and functions are visible only inside the module and only the functions that are explicitly exported by adding them to the `exports` object become visible to other modules. In the example the `idgenerator` module has one public function, one private function and one private variable. The `database` module imports the `generator` module by using the `require` function and storing the result in a variable. The `require` function could also be used to import single functions and objects.

Many existing client side JavaScript libraries already have support for modules. The listing here presents a naive version of how modules could be added into server side JavaScript. Everything is wrapped inside a closure in order to keep the namespace footprint down to a minimum. What is added into the global scope are the `require` function, the `exports` object and an `__eval__` function, which is used only by the module loader and not by the loaded modules. For the sake of keeping the code listing short and concise we have left out things such as nested modules, automatic reloading of modules if a newer version is available and module search order via `JSPATH` environment variable.

4.3 Embedding Jetty

Jetty is an open source web server and servlet container implemented purely in Java. It has been specifically designed with flexibility and embeddability in mind. Launching Jetty from a Java application is as simple as including Jetty libraries (JAR files) in the Java classpath and then in the program code creating a server instance and starting it. For the developers Jetty provides several extension points where the application code can for example, configure the way Jetty serves content, provide proprietary request handlers and register for listening on various events. Furthermore, because Jetty is fully implemented in Java and nicely packaged into Java libraries, it can be embedded into server side JavaScript too. In our case, the Rhino is first started from, for example, command line and it loads the given JavaScript application which, in turn, configures and starts Jetty. In the Listing 4 we present a sample JavaScript for configuring and launching Jetty from JavaScript.

```
// create server instance
var server = new Packages.org.mortbay.jetty.Server();

// create connector
var connector = new Packages.org.mortbay.jetty.nio.SelectChannelConnector();
connector.setPort(2345);
server.setConnectors([connector]);

// register our own http handler
server.setHandler(
    new Packages.org.mortbay.jetty.Handler(
        {
            handle: function (target, request, response, dispatch) {
                response.setStatus(200);
                response.setContentType('text/plain');
                var writer = response.getWriter();
                writer.write('Hello from server side JavaScript');
                writer.close();
                request.setHandled(true);
            }
        }));
```

```
// start the server                                                    23
server.start();                                                        24
server.join();                                                         25
```

Listing 4 Bootstrapping Jetty

When launching this script using Rhino, the application starts listening on port 2345 for incoming connections. When an HTTP request arrives, our `handle` function is executed and the user sees our greeting message. Using similar approach we can build our platform by adding handlers for remote method invocations, application loading, comet requests and static content.

4.4 Building the Platform

There are many different components involved in the implementation but in the scope of this paper two of them are more interesting than others. One is the networking component which implements the remote function calls and the other is the module that loads and partitions the applications between the client and the server. We will next address these two components in more detail, and after that we will have a quick look on how applications can benefit from all this.

4.4.1 Application partitioning

Application partitioning is implemented using the same export/require approach as normal module loading. This makes its usage intuitive and straightforward. Applications running in the browser require modules as described in subsection 4.2, and they can invoke exported functions of these modules as if they were local. In order to accomplish this, the code of the loaded application needs to be inspected and instrumented by the platform. The inspection and instrumentation occurs at application load time. When the application is first loaded by the browser the platform on the server intercepts the HTTP request, finds the JavaScript file implementing the application, reads it and instruments it with proper networking capabilities. This sequence is described in more detail in the following list:

1. The browser loads the HTML page containing the application.
2. The browser encounters a `<script>` tag and initiates an HTTP request to load the application.
3. The platform running on the server intercepts the request.
4. The platform finds the JavaScript file that implements the requested application.
5. The platform reads the application code and finds the names of all the modules that the application `require`s.
6. The platform goes through all required modules, finds all their exported functions, generates corresponding proxy functions for them and injects this code into the application code. For the sake of performance, the platform caches exported interfaces and the generated code of loaded modules.
7. The platform adds all the necessary Ajax and Comet related networking code into the application code.

8. The instrumented application is returned to the browser.

9. The browser runs the application and starts listening Comet messages.

As described above, remote modules are required just like local modules. So, any JavaScript module is a feasible remote module as well as a local module. Many existing JavaScript toolkits (such as Dojo) already provide a mechanism for requiring modules in the client side code but the key difference here is that modules required on our platform stay on the server; only the proxy functions are transferred to the client. Moreover, applications do not need to require whole module but they can require individual functions should they choose to do so. The platform provides some common modules for applications to require. These include the following:

- **db** – primitive access to relational database.
- **file** – access to the file system.
- **log** – logging facility.
- **cron** – scheduled tasks.
- **friends** – send and receive messages from other clients.

While more modules may be added later, application developers are also free to add their own modules by simply placing them into a directory that can be reached by the module loader (defined by the JSPATH environment variable) of the platform. After that they automatically become remote modules and applications running in the browser are able to require them. Naturally, providing remote access to resources such as file system and the database raises severe security concerns. Therefore, the platform and the applications it hosts would need to be authenticated and sandboxed.

4.4.2 Remote function calls

To allow client applications to invoke server's functions we implement two networking libraries into the framework; one for the client and one for the server.

As described earlier, the client-side JavaScript library provides functions that are actually proxies to server side counterparts. These proxies create URIs pointing to the implementations of the functions on the server and uses Ajax to invoke them. All function parameters and return values are transferred in JSON encoded format in the body of the HTTP POST request and response respectively. The server side JavaScript library handles all Ajax requests and forwards them to appropriate functions based on the URI that was requested. When the functions finishes, the return value is sent back to the browser via the HTTP request. An example of this kind of a sequence is presented in Figure 2 where a user performs login using an application that is hosted by the platform.

1. User clicks the submit button.

2. The application calls `get()` function of the `db` module, which is a proxy function generated by the platform.

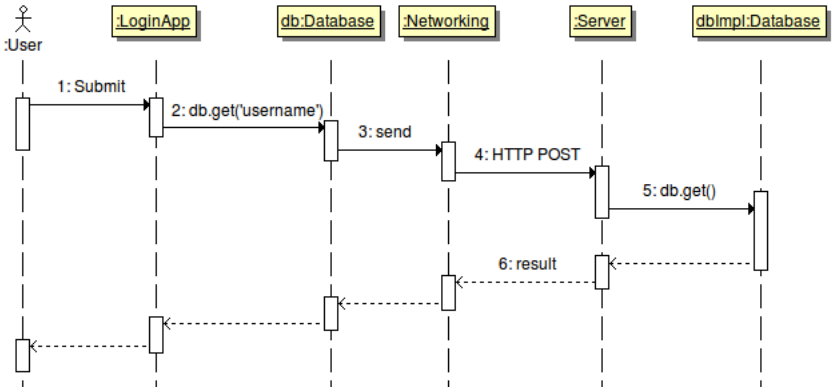

Fig. 2. Sequence of a remote invocation

3. The proxy function converts the function call into a URI – in this case '/modules/db/get' – and encodes all parameters into a JSON string.

4. Ajax request is sent to the server using HTTP POST method.

5. The platform handles the request, decodes the function parameters and finds the correct module and function based on the requested URI.

6. The platform encodes the result returned by the database, encodes it into a JSON string and writes it into the HTTP response.

One of the main principles for the communication library is that it should impose as little overhead to application developers as possible. Having said that, there is some overhead as the communication link may be asynchronous whereas normal function calls are synchronous. So far, we have only demonstrated synchronous remote function calls which looks exactly similar to calling a local function but if an asynchronous request is desirable an additional callback parameter needs to be passed to the function being invoked. The callback function then gets executed when the server comes back with an answer. This does not really add any application code as the result would have to be handled somewhere anyway. Rather, it just moves the handling of the result into a separate function. An example of this behavior can be seen in Listing 5 on lines 4 and 5. In this case the callback function is implemented as an unnamed function inside the invocation.

```
// require database module to the browser                                1
var db = require('db');                                                  2
                                                                         3
// update high score to db                                               4
db.update({                                                              5
  type: 'highscore',                                                     6
  data: {score: score},                                                  7
  constraint: {player: playerid}},                                       8
                                                                         9
  // we want to do this asynchronously, so give callback function       10
  function(response) {                                                   11
    alert('High score saved');                                           12
```

```
});                                                                    13
```

Listing 5 Remote call to server

In our implementation, Comet has been made more accessible to developers by wrapping it inside the `friends` module. This module exports functions for accessing other users; seeing who is online, sending and receiving private messages to them and also sending multicast messages to all users. On the server side we rely on Comet facilities provided by Jetty and on the client we have implemented our own comet library. When a client wants to send a message to other user, it just uses a function in the `friends` module and the message is sent to server side Comet implementation which in turn publishes it to the recipients channel.

4.4.3 Trying to be RESTful

Representational state transfer (REST) [19] is an architectural style for composing web services in a resource oriented way and accessing those resources using standard methods provided by HTTP. Furthermore, web services that adhere to the rules of REST are said to have *resource oriented architecture* (ROA). In ROA, the service is split into identifiable resources and each resource must have at least one URI pointing to it. These resources are then accessed and manipulated by using the methods defined in the HTTP protocol, most commonly, GET, POST, PUT, DELETE and HEAD. Another key principle behind RESTful web services is the statelessness; the state of the application should only be stored in the client. Only resources and their states are stored in the server. This means that each HTTP request should operate in complete isolation from any previous or future requests of the same client. Moreover, this means that each request must include any application state required in order to carry out the request.

In the realm of our platform, we have followed the principles of REST as much as was reasonable. We identified all modules and functions hierarchically as resources and assigned corresponding URIs for them. For example, fetching data from database can be performed by issuing a request to `/modules/db/get`. Also, the platform does not store any application data into the server; every method invocation results in self-contained HTTP request. The part where the platform deviates from the REST model is the usage of HTTP methods. Because the platform has no way of knowing about the semantics of the modules and the functions it presents as resources, it also cannot figure out which HTTP method is appropriate for a given remote invocation. This is why we have decided to use POST method for all requests and place JSON encoded function parameters into the body of the request. Of course, it could be argued that this approach does not break any REST principles as we are POSTing data (parameters) to functions which are, after all, the resources of our web service.

4.4.4 Using the platform

Writing an application to be hosted on our platform is much like writing any other client side application. As a matter of fact, if the application runs solely on the client and does not need any services provided by the server, there is no *special* code to be added into the application.

When the application does use modules provided by the server, there are still very few things to take into account. If remote functions are invoked synchronously, it is just a matter of requiring the modules using the `require` function provided by the platform and then calling the functions as if they were local. If remote functions are invoked asynchronously, the

caller needs to pass a callback function which handles the return value of the asynchronous invocation as already described in subsection 4.2. The part that requires a bit more knowledge is the Comet inside the `friends` module as it has few functions that the application may need to invoke during setup. These are functions like setting the username and registering to different channels. We however, feel that these are very moderate things to learn when utilizing a new platform.

5 Example

Next, we demonstrate the platform using an example, where we highlight the steps required to make an application capable of storing data and communicating with other clients. For this, we took an existing web application and enhanced it with collaboration and persistence. The application is a game running on Sun Labs Lively Kernel platform, an open source web programming environment developed at Sun Microsystems Laboratories [20]. The Lively Kernel is an environment for developing and hosting client side JavaScript applications in the browser. It provides a rich set of user interface components and an MVC model for event handling. It should be noted however that there is nothing specific to the Lively Kernel in our platform implementation. We are just using Lively Kernel to easily create nice looking applications for the browser.

As an example application we chose a 3-D maze walking game called CanvasScape, which ships with the Lively Kernel as a demo application. In the game the player walks in a labyrinth trying to find all the blue walls within a given time frame. We then added a collaborative feature to the game: when a wall is clicked the user can enter a fragment of JavaScript code and that code is stored persistently into the wall and the wall turns to red. That information is then stored into the database and replicated to all the other clients currently playing the game. If any of the players hits a red wall, the code stored into it, is executed. A screenshot of the application is shown in Figure 3.

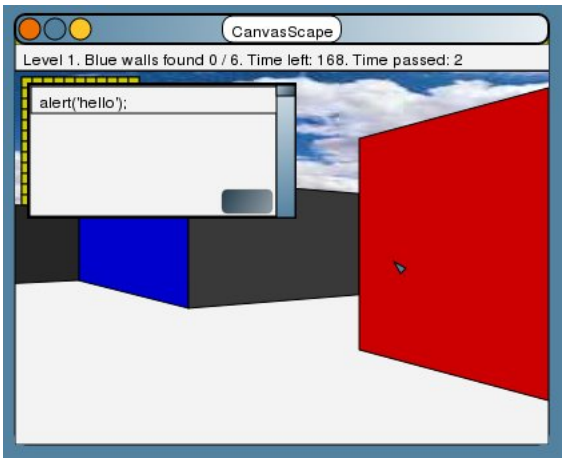

Fig. 3. CanvasScape

The application is implemented in a single file called `CanvasScape.js`. To this file we

added a class[a] to handle red walls and gave it the name `RedWall`. This class has two methods; one for adding code to a wall and one for removing it. Listing 6 shows the code that is executed in the browser while the actual implementations of the database module and the friends list are on the server. On lines 3 and 4 these two remote modules are required, which leads to code instrumentation at application load time. When these modules are used on lines 29 and 34, the code injected by the platform performs the necessary networking operations to carry out the remote function calls. The function starting on line 8 is a Comet callback function for when another player adds code to a wall. It is then the responsibility of that function to add the same code to its local copy of the maze.

```
// ...taken from RedWall-class                                          1
                                                                        2
this.database = require('db');                                         3
this.players = require('friends');                                     4
                                                                        5
                                                                        6
// comet event: some other player has added code to a wall              7
this.codeAdded = function (msg) {                                       8
  this.walls[msg.x][msg.y] = msg.code;                                  9
};                                                                      10
                                                                        11
                                                                        12
// ajax event: callback from asynchronous remote function call         13
this.callback = function (result) {                                     14
  // handle possible error conditions here                              15
};                                                                      16
                                                                        17
                                                                        18
// user event: user has clicked a wall and added some code              19
this.addCode = function (x, y, code) {                                  20
                                                                        21
  var data = {                                                          22
    x: x,                                                               23
    y: y,                                                               24
    code: code                                                          25
  };                                                                    26
                                                                        27
  // store into database                                                28
  this.database.insert(                                                 29
    {type: 'redwall', data: data},                                      30
    this.callback);                                                     31
                                                                        32
  // propagate to all players                                           33
  this.players.sendMulticast(data, this.callback);                      34
};                                                                      35
```

Listing 6 CanvasScape

As already mentioned, the application is implemented in a single file so there is no client side and server side parts of the application. Everything is run in the client but during the application loading some of the functions are replaced with proxies. Figure 4 depicts the flow of control when a player clicks on a wall in order to add code into it. The sequence reflects the code in Listing 6. It clearly shows the asynchronous nature of the application as opposed

[a]JavaScript does not actually have classes but some toolkits and frameworks (such as the Lively Kernel) mimic the behavior of classes.

to the synchronous login screen shown earlier in Figure 2.

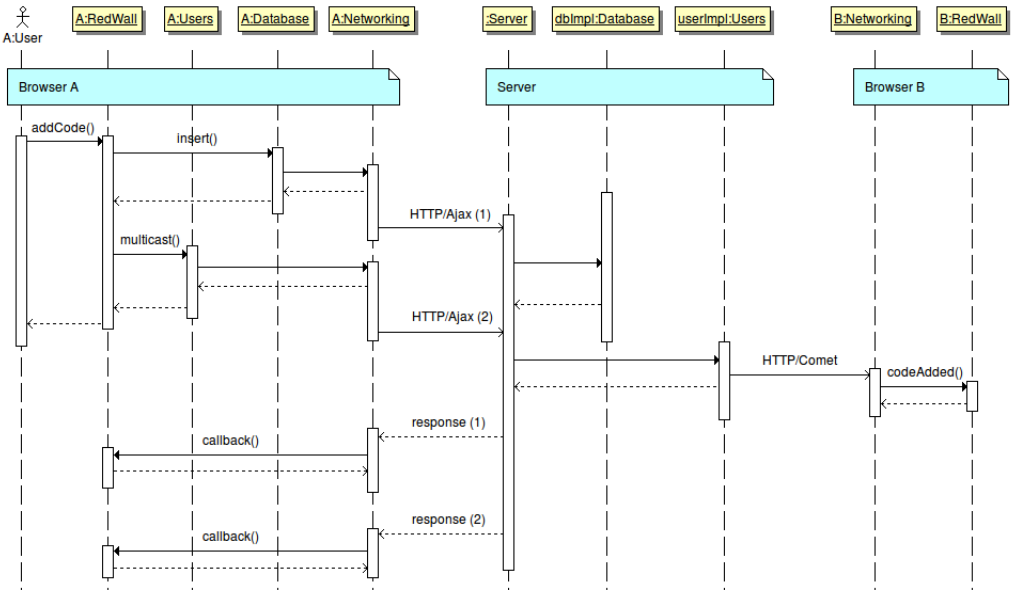

Fig. 4. CanvasScape: Player adds code to a wall

6 Related work

There are some products that share some similar ideas or approaches with the ones we have presented in this paper. Quick introductions of these products are presented in the following paragraphs.

Hilda [21] is a declarative, high-level programming language for developing data-driven web applications. It provides a uniform data model for all tiers of the application. The programming model in Hilda is state-based and the programmer specifies which operations are permitted in any given state of the program. The state of the application is maintained by the server. The server is also responsible for resolving any state conflicts between the client and the server before an update to the database can be made. The Hilda compiler translates the application code into Java Servlets and the application can then be deployed on a Servlet Container.

Further research has been done in extending the Hilda to handle automatic partitioning of web applications [22]. In this research the partitioning has been identified as an NP-hard optimization problem and as a solution, an approximation algorithm is given to optimize the partitioning. The runtime environment for Hilda is implemented as a Java Applet on the client. The Applet takes care of data caching, HTML rendering and keeping the application states (the one on the client and the one on the server) in sync. This makes the partitioning completely invisible for the programmer.

Compared to our approach this research offers a more scientific and automated approach to partitioning, while our motivation has been in practicality and providing an environment web application developers are already familiar with. Also, the Hilda language is designed for data driven applications which is clearly visible in the data modeling and statefulness of

the language. Our goal was to provide a more generic platform for running web applications mostly on the client.

Helma [23] is an open source web application framework that allows web applications running on the server to be written in JavaScript. It is written in Java and uses Rhino to integrate with JavaScript. The framework is built so that it enforces the MVC paradigm [24]. Helma is rather traditional web framework in the sense that applications run mostly on the server producing HTML-content for the browser. On the other hand, there are no limitations in Helma that would prevent from using fat clients and thus we could use it to run our implementation.

DWR [25] is an open source library that enables remote function calls between client and server. In DWR, one configures in an XML-file the functions that are to be published for clients. Based on the configuration file and Java reflection, DWR dynamically produces JavaScript proxies for the browser. DWR is written in Java and uses Java servlet technology to handle remote invocations. So, everything in the client is written in JavaScript and everything on the server is written using Java, thus making it more difficult to naturally partition applications. However, DWR presents some very good ideas and techniques for partitioning web applications.

Jaxer [26] is an Ajax web server, which brings the JavaScript and the DOM into the server. Applications can be implemented fully in JavaScript and the developer specifies which parts of the JavaScript implementation are executed in the server and which parts in the client. This is achieved by adding an additional `runat` attribute to the script tag (see Listing 7).

```
<script runat="server">                                                    1
  document.getElementById('acceptcb').checked = true;                      2
</script>                                                                  3
                                                                           4
<script runat="both">                                                      5
  var validateData = function (data) {                                     6
     // validate data...                                                   7
  };                                                                       8
</script>                                                                  9
```

Listing 7 Jaxer

This way web applications are able to manipulate web pages using the DOM on the server and the resulting page is sent to the browser. The Jaxer framework is then responsible for keeping the two DOM trees in sync. Jaxer embraces some very similar principles to what we have presented in this paper but it is essentially a thin client approach even though the development model is more uniform compared to traditional web application development.

Google Web Toolkit (GWT) [27] is an open source toolkit for developing web applications in the Java programming language. In GWT, the client and the server are implemented in two separate Java packages. The communication between them follows the RPC paradigm where the server provides services to clients by publishing interfaces and functions. These interfaces are used by the client package to call the remote functions while GWT takes care of all the networking details. The user interface is implemented in the client package using Java and is converted into DHTML during the build process. This way, the GWT also provides a uniform development experience by allowing the whole web application to be implemented in

Java. Consequently, compared to our solution, main differences between these two approaches lie in the technologies they use rather than in the concepts or ideas they represent. In GWT, Java branches out from the server into the browser while our solution does the opposite with JavaScript. In addition, while both platforms make functions callable from the client, they are technologically very different. Our solution uses the same `require` mechanism regardless of whether the required module is local or remote, whereas in GWT every remote function is explicitly marked as being remote by defining it within a remote interface.

Echo3 [28] platform bears resemblance to couple of other platforms. From the user interface point of view, it is similar to the Lively Kernel platform in that it encourages rich, single page web applications and it provides a lot of widgets and an event model. Of course Lively Kernel is purely client side platform and thus is written in JavaScript. Echo3 also resembles GWT because applications may be written entirely in Java and the converted into DHTML during the build phase. The main differences the Echo3 platform has over these two – as well as over our solution – is the way the applications are partitioned. Echo3 applications are predominantly run in the server and the platform completely hides the partitioning and the networking from the developer. This means that many user interface events (such as pressing a button) are handled in the server even when the fired action could be executed by the client alone.

The latest version of the user interface part of the Echo3 platform has been ported into JavaScript enabling the developers to write applications is JavaScript using the same APIs they are accustomed to when using the Java API. The trade off is that the uniform development model is lost because in this model the client and the server are implemented in two different languages and the developer has to take care of the communication layer between the two.

Bayeux [29] is a protocol defined to operate on top of a low latency protocol, primarily HTTP. It defines the transportation and encoding of asynchronous messages between clients and servers. In Bayeux, messages are encoded using JSON and it follows the publish/subscribe model via named channels. Ajax and Comet are the key technologies behind Bayeux. It is defined by the Dojo Foundation as an RFC document even though officially it is not one.

XMPP [11] [12] (formerly known as Jabber) is an XML based protocol for instant messaging. On top of the messaging protocol, the XMPP also includes the presence information. A lot of famous applications such Google Talk (http://www.google.com/talk/), iChat (http://www.apple.com/macosx/what-is-macosx/ichat.html) and the upcoming Google Wave (http://wave.google.com/) use XMPP as their messaging protocol. **BOSH** [13] further specifies the usage of XMPP over HTTP. It introduces a *connection manager* component which sits between the client and the server and takes care of the long lived HTTP connections. This is very similar technique to Comet's long-polling. For this and the fact that many web servers (such as Jetty and Tomcat) already provide a support for Comet, We have not used XMPP over BOSH in our solution. Having said that, there is no reason why we could not use BOSH as a replacement for Comet but we leave this experiment for future work.

HTML 5 [14] is the fifth major revision of the HTML language. It introduces the concept of *Web sockets*, which are essentially a bidirectional communication channel between the client and the server. This would assure a standard way for the server to send data to the client instead of using the variety of the Comet implementations. However, the specification is still

in the works and may remain so for a long time.

7 Conclusions

Building web applications that offer complex functions and a lot of user interactivity is becoming increasingly complex. To alleviate the development process we need to define the structure of a web application more precisely. In doing so, we can also bring back other tried and true principles and methods of software engineering that have gotten somewhat lost in all the different technologies that comprise today's web applications [1].

In this paper we have presented practices for partitioning web applications between the server and the client. We have given the principles for identifying the parts of the application that should be run on the server and on the client. We have also described the technological details of implementing the partitioning using an example application.

Currently, the platform is at development quality and it should be further tested and packaged nicely to make it more attractive to developers. Security concerns should also be studied and addressed properly. So far, no real actions have been taken in order to make the platform truly secure. The research paper [30] investigates security concerns related to web application partitioning thoroughly. Our belief is that while challenging, the security issues are something that can be addressed after we have fully functional and feature complete platform.

Acknowledgements

Financial support from the Academy of Finland grant 115485 is gratefully acknowledged.

References

1. T. Mikkonen and A. Taivalsaari. Web applications - spaghetti code for the 21st century. Technical report, Sun Microsystems, 2007.
2. The World Wide Web Consortium. HTML 4.01 Specification. http://www.w3.org/TR/html401/, 1999.
3. The World Wide Web Consortium. CSS 2.1 Specification. http://www.w3.org/TR/CSS2/, 2009.
4. ECMA, Standard ECMA-262, ECMAScript Language Specification 3rd Edition, 1999
5. D. Crockford. Json. RFC 4627, 2006.
6. The World Wide Web Consortium. Extensible Markup Language (XML) 1.0 (Fifth Edition). http://www.w3.org/TR/xml/, 2009.
7. J. J. Garrett. Ajax: A new approach to web application. http://www.adaptivepath.com/ideas/essays/archives/000385.php, 2005.
8. A. T. H. III. Ajax: The Definitive Guide. O Reilly Media, Inc., 2005.
9. A. Russell. Comet: Low latency data for the browser. http://alex.dojotoolkit.org/?p=545, 2006.
10. A. Russell. Comet: Low latency data for the browser. http://alex.dojotoolkit.org/wp- content/LowLatencyData.pdf, 2006.
11. Jabber Software Foundation. Extensible Messaging and Presence Protocol (XMPP): Core. http://xmpp.org/rfcs/rfc3920.html, 2004.
12. Jabber Software Foundation. Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence. http://xmpp.org/rfcs/rfc3920.html, 2004.
13. XMPP Standards Foundation. XEP-0206: XMPP Over BOSH. http://xmpp.org/extensions/xep-0206.html, 2009.
14. The World Wide Web Consortium. HTML 5. http://www.w3.org/TR/html5/, 2009.
15. WWW Consortium. Document object model. http://www.w3.org/DOM/, 2004.

16. Mozilla Foundation. SpiderMonkey (javascript-c) engine. http://www.mozilla.org/js/spidermonkey/, 2009.
17. Mozilla Foundation. Rhino: Javascript for java. http://www.mozilla.org/rhino/, 2009.
18. Mozilla Foundation. The same origin policy. http://www.mozilla.org/projects/security/components/same-origin.html, 2001.
19. L. Richardson and S. Ruby. RESTful Web Services. 2007. OReilly Media, Inc., 2007.
20. A. Taivalsaari, T. Mikkonen, D. Ingalls, and K. Palacz. Web browser as an application platform: The lively kernel experience. Technical report, Sun Microsystems, 2008.
21. F. Yang, J. Shanmugasundaram, M. Riedewald, J. Gehrke, A. Demers. Hilda: A High-Level Language for Data-Driven Web Applications.
22. F. Yang, N. Gupta, N. Gerner, X. Qi, A. Demers, J. Gehrke, J. Shanmugasundaram. A Unied Platform for Data Driven Web Applications with Automatic Client-Server Partitioning. WWW 2007, May 812, 2007, Banff, Alberta, Canada.
23. C. Zumbrunn. Helma. http://dev.helma.org/, 2009.
24. G. Krasner and S. Pope. A cookbook for using model-view-controller user interface paradigm in smalltalk- 80. Journal of Object-Oriented Programming, 1988.
25. F. Zammetti. Practical DWR 2 Projects. Apress, 2008.
26. Aptana Inc. Jaxer. http://aptana.com/jaxer, 2009.
27. Google. Google Web Toolkit. http://code.google.com/webtoolkit/, 2009.
28. NextApp. Echo3 Web Framework. http://echo.nextapp.com/site/echo3, 2009.
29. A. Russell, G. Wilkins, D. Davis, and M. Nesbitt. Bayeux protocol. Technical report, The Dojo Foundation, 2007.
30. S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, X. Zheng. Secure Web Applications via Automatic Partitioning. SOSP07, 2007, Stevenson, Washington, USA.