Department of Creative Informatics
Graduate School of Information Science and Technology
THE UNIVERSITY OF TOKYO

Master Thesis

# Redom: A Distributed Object based Server-Centric Web Application Framework

Redom： 分散オブジェクトに基づいたサーバー中心のウェブフレームワーク

**Eki Ko**
胡　益

Supervisor:　Professor Chiba Shigeru

January 2013

# Abstract

Nowadays, dynamic web applications have been widely developed and used for people to retrieve information and interact with remote nodes. AJAX is the most used technique to create dynamic web applications so that data from a server can be retrieved from a server asynchronously and the web page can be refreshed without reload the whole page repeatedly. However, due to the fact that JavaScript dependency and distributed development on both browser and server sides in a AJAX-based web development, it is complicated and inefficient to create dynamic web applications.

We proposed Redom, a distributed object based server-centric user-friendly web application framework. Redom enables developers to write all application logics at server side easily using both browser-side and server-side libraries. We have implemented the framework using Ruby language. Redom provides distributed objects published by browser in natural Ruby syntax so that developers can access browser-side objects directly. In order to avoid overhead of accessing distributed objects, we have introduced techniques such as Batched futures and Ruby-to-JavaScript compilation.

In this paper, we discuss the problems in current web development and then describe the design and implementation of our proposed web framework which aims to solve these problems. We also show the result of productivity and performance evaluation of our web framework.

# 概要

今日、動的な Web アップリケーションは、リモートノードと情報をやりとりするために広く普及し、開発されている。その中でも AJAX は、サーバーと非同期にデータのやりとりを行い、Web ページ全体を何度も読み込み直すことなく、更新が行えることから、Web アップリケーションを開発する手法として広く採用されている。しかし、AJAX は JavaScript の知識を要する上に、Web ブラウザと Web サーバーそれぞれのプログラムを実装する必要があるため、Web アップリケーションを効率的な開発の防げとなっている。

本研究では Web フレームワーク Redom を開発した。Redom の特長は、分散オブジェクトをベースにしていること、Web アップリケーションの実装がサーバー内で完結していること、Ruby のメソッド呼び出しによって Web ブラウザ側のオブジェクトのメソッドを直接呼ぶ ’ ことがきでることである。これらの特長により、Web アップリケーションの開発を容易かつ効率的に行うことができる。また、分散オブジェクトの呼び出しのオーバーヘッドを軽減するため、Batched futures や Ruby から JavaScript へのコンパイルなどの手法を使用した。

本稿では、まず既存の Web フレームワークの問題点について議論し、我々が提案する Redom の設計及び実装について述べ、RPC 及び DOM 操作のパフォーマンスを改善する手法について述べる。そして、生産性と性能について評価して結果を述べる。

# Contents

# Chapter 1

# Introduction

We have developed a single-language and server-centric[1] web framework by using distributed objects[2], which makes web development simple and productive. Web applications are becoming more and more important nowadays because of the need for information sharing and remote interacting. Therefore, solutions to enable an effective web application development have been studied and proposed. In this chapter, we give an overview of problems in current web application development and our approach to solve these problems.

## 1.1 Problems in web development

With the development of Internet technologies, more and more web applications are used for people to retrieve information and interact with web services. User experience has been also greatly improved thanks to the advanced web technologies that enables developers to create dynamic, expressive and interactive web application[3].

One of the most popular technique for creating dynamic web application is AJAX (Asynchronous JavaScript and XML)[4]. By using the XMLHttpRequest object, web applications can send data to and retrieve data from web server asynchronously without refreshing and rebuilding the whole page repetitively. Combined with the DOM[5] manipulation APIs, developers can create desktop-like web applications which take users' action as input and change the DOM on the page dynamically and naturally. AJAX changed the way of web development from only server-side program which returns whole page, to server-side program collaborating with browser-side program which changes the page dynamically.

However, the separation of web development into server and client has several disadvantages, which cause productivity issues in the development. First, it is unavoidable to use JavaScript as the browser-side programming language. For a non expert in JavaScript, programming becomes difficult because of the distinct characteristic of JavaScript from other normal object-oriented programming language. Second, the isolated development over browser and server imposes a heavy burden on developers. Developers have to use different semantics, libraries from different programming languages. Code written in different languages can not be shared between browser and server. Also, developers are responsible for coding the browser/server communication details as well as data format conversion. Finally, debugging is difficult because there are no easy ways for error tracing.

## 1.2 Current approaches to support web development

A lot of approaches have been studied and proposed to overcome the disadvantages of web development which is performed on both server and client. These approaches usually

turn out to be implemented as libraries or web frameworks that support the development of web application. Each approach aims to solve one or more problems mentioned above but also has its own disadvantages.

To simplify client-side programming, there are many JavaScript libraries which provides simple APIs for developers to write programs in JavaScript easily. Moreover, other approaches aim to allow developers to write browser-side program in other programming language, which can be compiled to JavaScript and executed on browser. These approaches facilitate browser-side programming but do not contribute much to the whole web development in which server-side programming plays an important role.

Approaches that focus on simplifying both browser and server side programming in web development have been proposed more and more. Server-centric web frameworks move all application logic to the server, hiding most or all client-side programming from the developer. Single-language web frameworks allow developers to create web applications in a single programming language for all aspects. Although these web frameworks provide good support in web development, but they still have limitations or disadvantages. We will discuss the details about various web frameworks in chapter 2.

## 1.3 Goal and proposal

Our goal is to design and implement a web framework to solve all the problems mentioned above. We named this web framework Redom. Redom is a single-language server-centric web framework based on Ruby. To develop such web framework, we proposed to use distributed objects from browser which can be accessed by RPC[6] easily from server-side program. Redom hides all server/browser communication details from developers and provides simple API for browser/browser communication. It is possible for developers to use browser-side JavaScript libraries from server-side Ruby program. Moreover, the debugging becomes easy due to the debugging support by Redom. By using Redom, developers can create web applications using only one programming language and put all application logics at server side. Therefore, a web development is simplified and the productivity is improved.

However, there is a performance problem in such distributed object system. Every time an RPC occurs at server side, the result of RPC must be retrieved from browser side through a server/browser communication. Because the overhead of communication is extremely high, frequent RPCs will bring a long latency to user and affect the performance of the whole system as well as user experience.

We have improved the performance of Redom by applying batched futures to RPCs in Redom. Batched futures allows the process to continue without synchronization by using future objects. A synchronization is triggered only when the result of RPC is needed for subsequent execution. As a result, the communications between server and browser is reduced and the performance is improved.

We have further improved the performance of Redom by introducing Ruby-to-JavaScript compilation in Redom. We have found another fact that affects the performance of Redom, which is centralized DOM manipulation. Because DOM manipulations cause server/browser communication, when DOM manipulation occurs frequently at server side, the large number of RPCs will cause the system to perform bad. To solve this problem, we proposed to compile code of centralized DOM manipulation, which requires no server-side resources when executed, to JavaScript code and executed at browser side. As a result, there will be only single communication and the further process of DOM manipulation will be executed at browser side without communication.

We implemented the web framework Redom based on the principles above. WebSocket

is used in Redom as the browser/server communication protocol to establish a full-duplex connection. We used multi-threading model in Redom to handle requests from browser at first. However, the overhead of creating a thread in Ruby is very high, and numerous browsers connecting simultaneously to a web server cause C10K problem[7]. Therefore, we changed our framework to use *reactor pattern*[8] combined with *thread pool pattern*[9] so that it is able to tolerate large number of connections with high performance. Another problem is that the overhead of server/browser communication caused by the synchronization of RPC is so high that the latency of an execution will become extremely high. To solve this problem, we implemented *batched futures* in our proposed framework using Ruby *Fiber*[10], a coroutine support in Ruby, and *future*[11] mechanism. Further, we implemented Ruby-to-JavaScript compilation using Opal so that developers can compile centralized DOM manipulations into JavaScript for a higher performance.

## 1.4 Organization of this paper

This paper gives an overview of existing web frameworks and current techniques used in these web frameworks in Chapter 2. We introduce the usage and features of Redom in chapter 3. We describe the proposal of improving performance of our proposed web framework in Chapter 4 and the implementation details in Chapter 5. The evaluation result of our web framework is shown in Chapter 6. And there is a summary of our research in the last chapter.

# Chapter 2

# Backgound

In this chapter, we discuss the background of our research including related works. We also describe the techniques we used in our proposed web framework.

## 2.1 Problems in AJAX-based web development

Nowadays, a web application usually refers to a client/server application over a network which uses a web browser as the client. The web has been changed from a system for serving static documents to a large platform for deploying interactive applications as a result of the rapid development of web technologies. *Interactive* can be explained in two ways, one is that a web application is able to respond to users' interaction with the browser, the other meaning is that browser is able to interact with a web server for exchanging data.

There have been several techniques for interactive web application development such as Java Applet, which have been substituted by a widely used technique now, AJAX. AJAX is not a brand new single technique but a group of interrelated web development techniques used at browser side, which contains XMLHttpRequest, asynchronous JavaScript, DOM manipulation APIs, XML or JSON and so on. With the use of XMLHttpRequest object and asynchronous JavaScript, plain text or structured data formatted in XML or JSON can be sent to or retrieved from a web server asynchronously. Then the content of web page can be created or changed using DOM manipulation API dynamically without refreshing the whole page for a good user experience.

AJAX changed the web application to be interactive and expressive to users. However, the web development based on AJAX is getting difficult for developers. The reasons are:

- Multi-language programming
  Compared to various server-side programming languages such as Java, PHP, Python, Ruby, etc., ECMAScript[12] is almost the only scripting language used for browser-side execution, the superset of which in most browsers is known as JavaScript. Because AJAX is a group of technologies in which JavaScript plays the major role, developers have to know JavaScript to make use of AJAX.
- Distributed development
  Because AJAX is just only technique used for creating browser-side programs, developers have to write server-side programs as well to create a dynamic web application. The distributed development of a web application at both browser side and server side increases developers ' burden in creating, reusing, debugging and maintaining the source codes. Also, developers have to concern about the browser/server communication details, which makes the development complicated.
- Difficulty in debugging
  Due to the fact that programs are isolated at both browser and server sides, when

an error occurs in an AJAX-based web application, it is difficult to determine that from which part of programs the bug comes from.

## 2.2 Existing approaches for supporting web development

As mentioned in section 1.2, there are many existing approaches that aim to provide support in web development. These approaches can be generally separated into two categories, browser-side approaches and server-side approaches. Server-side approaches generally refers to web frameworks. According the different focuses on supporting web development, we can put web frameworks in two categories, server-centric web framework and single-language web framework. In this section, we analyze the advantages and disadvantages of these approaches and make a comparison.

### 2.2.1 Browser-side approaches

Because JavaScript is the only standard browser-side programming language, developers have to know JavaScript grammar to write scripts that run at browser side. However, not every developer knows JavaScript well, and it is sometimes difficult for a developer to learn JavaScript well because JavaScript has its own grammar which is quite different from other most used object-oriented programming language. The difficulty of using JavaScript makes it inefficient to create web applications. Therefore, there are many approaches that focus on simplifying browser-side programming.

One approach to simplify browser-side programming is to provide JavaScript libraries that contain APIs easy to use. These libraries either provide simple APIs for Browser/Server communication, or features that support developers to manipulate DOM easily. There are several famous JavaScript libraries that are widely used in current web application development.

jQuery jQuery is a fast and concise JavaScript Library that simplifies HTML document traversing, event handling, animating, and Ajax interactions for rapid web development. jQuery is designed to change the way that you write JavaScript.[13]

Prototype A foundation for ambitious web user interfaces. Prototype takes the complexity out of client-side web programming. Built to solve real-world problems, it adds useful extensions to the browser scripting environment and provides elegant APIs around the clumsy interfaces of Ajax and the Document Object Model.[14]

These libraries to some extent reduce the coding by free developers from being involved in the complexity of JavaScript but the fact that using JavaScript for browser-side development is not changed.

Another approach is to write programs in other programming languages and compile them into JavaScript. It is easy and efficient for developers to write programs with their familiar programming language. Using the same programming language for writing both server- and browser-side programs also makes it possible to share code between browser and server.

CoffeeScript CoffeeScript is a little language that compiles into JavaScript. Underneath all those awkward braces and semicolons, JavaScript has always had a gorgeous object model at its heart. CoffeeScript is an attempt to expose the good parts of JavaScript in a simple way. The golden rule of CoffeeScript is:

"It's just JavaScript". The code compiles one-to-one into the equivalent JS, and there is no interpretation at runtime. You can use any existing JavaScript library seamlessly from CoffeeScript (and vice-versa). The compiled output is readable and pretty-printed, passes through JavaScript Lint without warnings, will work in every JavaScript runtime, and tends to run as fast or faster than the equivalent handwritten JavaScript.[15]

Opal Opal is a ruby to javascript compiler. It is source-to-source, making it fast as a runtime. Opal includes a compiler (which can be run in any browser), a corelib and runtime implementation. The corelib/runtime is also very small (10.8kb gzipped).[16]

It is a effective way to improve the productivity of browser-side development that various programming languages can be chosen for browser. However, the fact that web development is separated into two parts is not changed. Developers also have to concern about server-side logics and the communication details between browser and server.

### 2.2.2 Server-centric web framework

Server-centric web framework refers to a web framework that hides most or all browser-side programming and client/server communication details from the developer by moving all application logics to the server side. In a sever-centric web framework, the server maintains the view state and manages the control to browser-side interactions. Because all programs are centralized on the server side, developers are able to create web applications without less concern about the distributed nature of web applications.

Ruby on Rails Rails is a web application development framework written in the Ruby language. It is designed to make programming web applications easier by making assumptions about what every developer needs to get started. It allows you to write less code while accomplishing more than many other languages and frameworks.[17]

CloudBrowser CloudBrowser is a web application framework that supports the development of rich Internet applications whose entire user interface and application logic resides on the server, while all client/server communication is provided by the framework. CloudBrowser thus hides the distributed nature of these applications from the developer, creating an environment similar to that provided by a desktop user interface library. CloudBrowser preserves the user interface state in a server-side virtual browser that is maintained across visits.[18]

Server-centric web frameworks simplify web development because developers do not have to write client-side programs. However, the server-side programs are usually created with a mix of HTML, JavaScript and server-side programming language, which makes developers suffer from using different language for different parts of a web application. Further, developers are often not able to use JavaScript libraries in server-side programs.

### 2.2.3 Single-language web framework

To increase the consistency of programming, one effective way is to allow the development of all aspects of web applications to be performed in a single language. Developers can create web applications with less effort when using a single language, because they don't have to change the programming habit between different programming language and code is reusable.

Node.js Node.js is a platform built on Chrome's JavaScript runtime for easily build-

ing fast, scalable network applications. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices.[19]

Node.js enables developers to write programs on the server side in JavaScript.

Google Web Toolkit (GWT) Google Web Toolkit (GWT) is a development toolkit for building and optimizing complex browser-based applications. Its goal is to enable productive development of high-performance web applications without the developer having to be an expert in browser quirks, XMLHttpRequest, and JavaScript.[20]

Orca Orca framework allows developers to work collaboratively on all aspects of Web applications in a single object-oriented language. Orca reduces the dominant difference between client and server by letting developers describe client parts, server functionality, and client-server communication in the language and development environment of the server.[21]

Links Links is a new programming language designed to make web programming easier. Links eases the impedance mismatch problem by providing a single language for all three tiers. The system generates code for each tier; for instance, translating some code into Javascript for the browser, some into a bytecode for the server, and some into SQL for the database.[22]

HOP Hop incorporates all the required Web-related features into a single language with a single homogeneous development and execution platform, uniformly covering all the aspects of a Web application: client side, server side, communication, and access to third-party resources. Hop embodies and generalizes both HTML and JavaScript functionalities in a Scheme-based platform that also provides the user with a fully general algorithmic language. Web services and APIs can be used as easily as standard library functions, whether on the server side or the client side.[23]

It becomes easier for developers to use a single programming language in web development. However, there are still some problems when using single-language web frameworks. First, some single-language web frameworks are not hiding the distributed nature of web development, such as Node.js. Second, developers have to learn a new programming language, such as Orca, Links and HOP. Third, it is not possible to use JavaScript libraries from server-side program, such as GWT. Last, there is seldom debugging support in these web frameworks.

### 2.2.4 Problems in existing web frameworks

From the web frameworks we described in previous section, we can outline the problems in existing web frameworks as follow:

- Programming separately on both client side and server side.
- Using different programming language in a single web development.
- Developers have to learn a new programming language to use a web framework.
- Unable to use JavaScript libraries.
- Mixing HTML in server-side program which makes it difficult to design the page.
- Difficulty in debugging.

## 2.3 Remote Procedure Call (RPC)

The Remote Procedure Call (RPC) is a popular paradigm for inter-process communication between processes in different computers across the network, which is widely used in various distributed systems[24]. RPC allows a procedure to be executed on another computer over the network without the details of communication being coded explicitly in program. Therefore, the programer can write the same program no matter if it is a local execution or a remote interaction. In a distributed system, RPC is usually triggered at client side. Then the client sends the request message, which contains the procedure name and parameters, to the server to inform the server to execute the specified procedure. The server return the result to the client after the execution.

RPC can be classified into two types, synchronous RPC and asynchronous RPC, depending on whether the RPC blocks the caller (client). When a synchronous RPC is invoked, the caller ' s process is blocked until the RPC process is complete. The implementation of a synchronous RPC system is easy but on the other hand the performance of such a system could be extremely low. Compare to blocking caller ' s process at every RPC invocation, asynchronous RPC do not block the caller (client) and the replies can be received when they are needed, which allows the client execution to proceed locally in parallel with the server invocation. Asynchronous RPC can be classified into two types depending on whether the call returns a value[25].

## 2.4 Distributed object system

Distributed object can be considered as a extension of remote procedure call in object-oriented programming. A distributed object is an object residing in a different address space, such as multiple computers over the network or different process on the same computer. As same as RPC, a distributed object can be accessed from remote process while the programmer do not have to know how the communication is performed. A distributed object system is a system that consists of various processes, on which distributed objects from a process (usually a server) can be accessed from other processes (usually clients)[26]. A distributed object system hides the details how objects are distributed from user so that distributed objects can be used as local objects. Distributed object system is usually used to create distributed applications which need to exchange data in the form of object or manipulate objects remotely.

For example, dRuby is a distributed object system which is written in pure Ruby and use its own protocol[27]. dRuby allows an distributed object published at server side to be accessed from other processes either on the same machine or different machines over the network. It is very easy for programmers to create distributed applications with dRuby because the details of method invocation or property reference on an distributed object are hidden from programmers. However, the performance of dRuby is not high because the access to distributed objects is performed synchronously while blocking the client-side process.

## 2.5 Ruby programming language

Ruby is a dynamic, open source programming language with a focus on simplicity and productivity. It has an elegant syntax that is natural to read and easy to write[28]. Ruby is widely used in different areas nowadays as a result of its ease of use, especially in web development. Because Ruby is a high productivity programming language, a lot of web

frameworks have been developed based on Ruby for agile web development. Therefore, we choose Ruby to implement our proposed web framework for a high productivity in web development. Further, no extra effort is needed for current developers who use Ruby web frameworks to create web applications using Redom.

# Chapter 3

# Redom web framework

We have developed a web framework Redom, to provide a single-language server-centric development environment for developers to create dynamic web applications with less effort. In this chapter, we describe the features of Redom and how we design it to be a productive web framework with high performance.

## 3.1 Goals and proposal

As presented above, the main challenge in web development is how to improve the productivity. This leads to the following goals of Redom to solve the problems existing in current web frameworks outlined in section 2.2.4.

- Single-language web framework
  Using just one programming language for both browser- and server-side programming will reduce the complexity of creating a web application. Developers do not have to change their programming habit between different programming languages. The code written can be reused at both sides. Furthermore, there is no need to convert data created in one language into another format for the future use in another language.
- Server-centric web framework
  A server-centric web framework means that all application logics are at server side. Developers can write a single program that contains both server-side resource accessing and browser-side objects manipulation, in a more natural process flow style. Developers do not have to concern about the browser/server communication details because they are hidden from developers by the framework.

To achieve the goal, we proposed a distribute object system to publish all browser-side objects as distributed objects which can be accessed from server-side Ruby program. Though the techniques we used in our proposal are not brand new approaches but the combination of them makes an novel approach. There are three key ideas in our proposal.

- RPC from server to browser
  It is common in an web framework that browser-side program can call server-side procedures when a browser-side event occurs. But usually there is no support for server-side program to call browser-side procedures when necessary. Our approach makes it possible that server can call browser-side procedures or methods at anytime. Therefore, it is easy for developer to make web applications which need server-push operations.
- All browser-side objects are distributed objects by default
  Generally in a normal distributed object system, it requires programmers to specify an object to be a distributed object explicitly before they can access it remotely.

This increases coding work and makes the program less readable. In our approach, all browser-side objects published as distributed objects so that developers can use them without declaration like they are local objects.

- Based on Ruby
  Ruby has become a famous and widely used object-oriented scripting language because of its simple syntax and high productivity. More and more web applications are created using web frameworks for Ruby. So we choose Ruby to implement our approach to provide a productive web framework.

## 3.2 System overview

Redom consists of two components, Redom server and Redom runtime3.1. The Redom server is working upon a Ruby web server. The Redom runtime is embeded in a Web page as a JavaScript library at browser side, which is responsible for interacting with DOM, for example, listening to browser events or executing DOM manipulations. Also, the Redom runtime is responsible for communicating with Redom server to receive data or sent request. The web server is used to create and keep connections between server and browsers. The Redom server engine is responsible for creating RPC message when RPC is triggered in application, and receiving events from browser and dispatch them to event handlers defined in application.

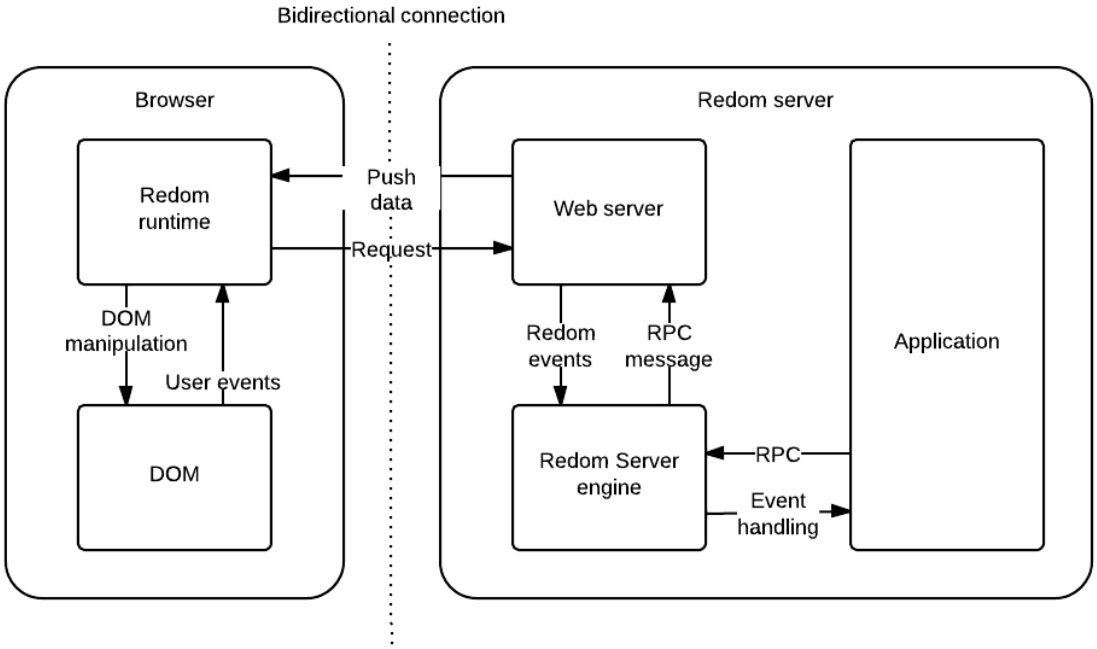


Fig. 3.1. Overview of Redom web framework

## 3.3 Redom by example

We present an example application to give an intuition of developing Web applications with Redom. We created a web chat application in minutes that shows the basic usage of Redom. This chat application allows users to input messages and send them to every-

one who is currently connecting to the server (figure 3.2). All the source codes of this application is shown in listing 3.1 and listing 3.2.

Listing 3.1 shows the browser-side code of the chat application which is a HTML document. There are several elements defined in the document with which we want to interact, such as a *send* button. We include the Redom runtime *redom.js*, as well as a JavaScript library *jQuery*, in this HTML file.

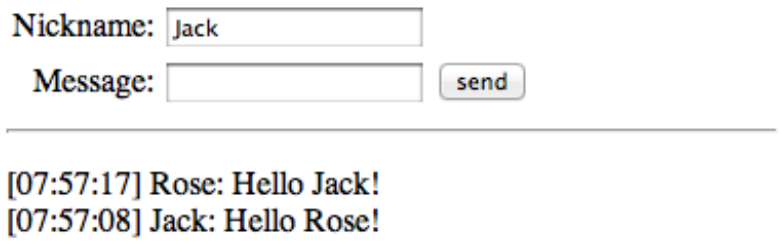


Fig. 3.2. Chat web application

```
<HTML>
  <HEAD>
    <TITLE>Redom Chat</TITLE>
    <SCRIPT type="text/javascript" src="jQuery.js"></SCRIPT>
    <SCRIPT type="text/javascript" src="redom.js"></SCRIPT>
    <SCRIPT type="text/javascript">
      $(function() { Redom("ws://localhost:8080/websocket").open("ChatConnection"); });
    </SCRIPT>
  </HEAD>
  <BODY>
    Nickname: <INPUT type="text" id="nickname" value="" />
    Message: <INPUT type="text" id="message" value="" />
    <INPUT type="button" id="btn" value="send" />
    <DIV id="chats"></DIV>
    </BODY>
</HTML>
```

Listing 3.1. chat.html

Listing 3.2 shows the server-side program of the chat application. It is a Ruby script file that can be executed by a Ruby interpreter.

```
class ChatConnection
  include Redom::Connection

  def send_chat_msg(msg)
    jQuery("#chats").prepend jQuery("<div>#{msg}</div>")
  end

  def on_open
    jQuery("#btn")[0].onclick { |event|
```

```
10        nickname = jQuery("#nickname").attr('value')
11        message = jQuery("#message").attr('value')
12        sync{}
13        connections.each { |conn|
14          conn.async.send_chat_msg("#{nickname}: #{message}")
15        }
16      }
17    end
18  end
```

Listing 3.2. chat.rb

The browser-side program contains just a single row of JavaScript code:

JavaScript

```
$(function() { Redom("ws://localhost:8080/websocket").open("ChatConnection"); });
```

This line of code tells the browser to connect to Redom server after the document have been loaded. As shown in the source code of chat.html, there is no other JavaScript code written in the browser-side program, because we moved all the application logics to the server side.

To write the server-side program, we first need to make a class which include *Redom::Connection*. Then we define the event handlers in the class to tell the browser what to do when certain event occurs.

In method *on_open*, which is called when the connection between browser and server is established, we add an *onclick* event handler to the *send* button using a block:

Ruby

```
jQuery("#btn")[0].onclick { |event|
  ...
}
```

Within the event handler onclick, we can access browser-side object using RPC:

Ruby

```
nickname = jQuery("#nickname").attr('value')
```

As we can see, we can also use browser-side JavaScript library, which is jQuery in this web application. To get the RPC result from browser, we can trigger a synchronization by using *sync*{}. Then we can send the message to other browser by invoking method *send_chat_msg(msg)* on other connection objects. All the connection objects can be retrieved using method *connections* provided by Redom.

Ruby

```
connections.each { |conn|
  conn.async.send_chat_msg("#{nickname}: #{message}")
}
```

To start this web application, we can use the command `redom chat.rb`. After the Redom server has started, we can open the chat.html in a web browser and enjoy the chat. When a user has input the message and clicked *send* button, the event handler onclick at server-side will be invoked. Then the server will retrieve the message from that

browser and send it to other browsers, as what we defined.

## 3.4 Features

By the example of Chat web application, we have describe how to create a web application using Redom. In this section, we describe the features of Redom that allow developers to create web applications easily.

### 3.4.1 Benefit from all Ruby advantages

It is free to use all Ruby features (some features with limitation). Therefore, developers can benefit from all the advantages of Ruby.

For example, in JavaScript, the definition of a class is a little confusing because its prototype-based semantic which makes it less readable.

JavaScript

```
JavaScript
function Person(name) {
  this.name = name;
}

Person.prototype.talk = function() {
  console.log("Hello, I'm " + this.name);
}

john = new Person("John");
john.talk(); // Hello, I'm John
```

On the other hand, the class definition in Ruby is in a more common form that developers with knowledge of object-oriented programming can read easily.

Ruby

```
class Person
  def initialize(name)
    @name = name
  end

  def talk
    puts "Hello, I'm #{@name}"
  end
end

john = Person.new "John"
john.talk # Hello, I'm John
```

### 3.4.2 Simple RPC mechanism

Redom provides simple RPC mechanism that allows server-side program to call browser-side methods or functions. As long as the named method exists at browser side, it is able to call that method as if it is a local method defined at server side, without any definition

or declaration beforehand.

For example, there is a function *alert* defined in window object (the top level object in JavaScript that represents the opened window in a browser). When we want to call the method, we call it like a normal method invocation.

Ruby
```
window.alert "Hello"
```

Furthermore, we have set the default scope to the window object, which is the same as how it is in browser-side JavaScript. Therefore, *window* can be omitted when calling a method which is defined under the window object. The code above can be abridged to:

Ruby
```
alert "Hello"
```

### 3.4.3 DOM manipulation

Unlike some other web frameworks, Redom does not provide DOM manipulation APIs wrapped in server-side programming language, which is Ruby in our case. Because we can call browser-side methods using very simple RPCs, we can do DOM manipulation by using those browser-side APIs effortlessly.

For example, the following DOM manipulation can be done easily using Ruby from server.

Ruby
```
# Get the input value from the text box whose id is "tb"
text = document.getElementById("tb").value

# Add a text node whose content is current url to the page
document.body.appendChild document.createTextNode(location.href)
```

### 3.4.4 Free to use browser-side JavaScript libraries

Not only the embedded APIs, but also JavaScript libraries can be used in Redom. Developers can use all the features provided by the loaded JavaScript libraries as needed in order to simplify and accelerate the development.

For example, developers can use jQuery for DOM manipulation.

Ruby
```
# Get the input value from the text box whose id is "tb" using jQuery
text = jQuery("tb").attr("value")

# Add a text node whose content is current url to the page using jQuery
jQuery("body").append(location.href)
```

### 3.4.5 Event-driven programming

We introduced event-driven programming into Redom because it is the most natural and widely used model for web framework. A Redom server listens to both browser and

server sides. When an event occurs at no matter browser or server side, its relevant event handler will be invoked if defined. There are three pre-defined events handlers that can be overridden by developer shown in table 3.1.

| Event handler | Description |
|---|---|
| on_open | Called when the server/browser connection is established. |
| on_close | Called when the server/browser connection is closed. |
| on_error(err) | Called when an error occurs. |

Table. 3.1. Pre-defined event handler

Usually, developer should override *on_open* to be the entrance of an web application. All initialization is supposed to be placed here. And *on_close* is the place to put finalization process while *on_error* can be used for handling .

Besides the event handlers provided by Redom, developers are able to defined event handlers to browser-side events. When such event is triggered at browser side, the Redom runtime will perform a RPC to invoke the event handler at server side. Developers don't have to worry about how the event handler is invoked because Redom takes care of it.

There are two ways to defined a event handler. One way is to assign the event handler with a symbol or method object.

Ruby

```
# Define a event handler
def on_click(event)
  alert "Clicked."
end

# Assign the event handler to the "onclick" event of a button,
# using a symbol
document.getElementById("btn1").onclick = :on_click

# The method object which represents the defined event handler
m = method("on_click")
# Assign the event handler to the "onclick" event of a button,
# using a method object
document.getElementById("btn2").onclick = m
```

The other way is to pass the event handler using a block.

Ruby

```
# Add the event handler to the "onclick" event of a button,
# using a block
document.getElementById("btn1").onclick { |event|
  alert "Clicked."
}
```

### 3.4.6 Support for debugging

As we mentioned in previous section, developers can catch runtime error by using event handler *on_error*. When an error occurs, no matter at client side or server side, an error event is triggered and the *on_error* method is invoked automatically. With the error

information passed to the method as a parameter, developers can easily find out that in which line of the source code the bug exists.

### 3.4.7 Browser-to-browser communication

Sometimes, there is a need for communicating or exchanging data between browsers. In most current web frameworks, there is no easy way to achieve such purpose. However, Redom provides a easy API for developers to create web application which browsers can interact with each other. As we described before, every server/browser connection is instantiated as a Redom::Connection object. We provide the method *connections* for developers to get all active connections that server is currently handling. Developers are able to operate on these connections objects such as make a method invocation and thus, one browser can be manipulated by another one.

We have shown the example of chat application which uses *connections* method to send message to other users. We can also use a block to do this.

Ruby

```
# Execute the block on every active connection
connections.each { |conn|
  # Invoke the block asynchronously
  conn.async {
    # Process that is to be executed
    alert "Hello!"
  }
}
```

As we can see, we used *async* to tell the server to call the block asynchronously. Developers can also call a method or block synchronously using *sync*. Unless special needs, we suggest to call a method asynchronously because the current process will not be blocked by doing this and the execution will be perform parallel.

### 3.4.8 Embedded WebSocket server in Redom

Redom needs a WebSocket server to handler the WebSocket requests. Though we used EventMachine in Redom as our default WebSocket server, other WebSocket server is also available in Redom. It is very easy to embed a WebSocket server in Redom because what the developers should do is to dispatch the WebSocket requests to Redom.

For example, if we want use WebSocket Rack[29], we can config it like this:

Ruby

```ruby
class RedomApp < Rack::WebSocket::Application
  def on_open(env)
    Redom.on_open self
  end

  def on_message(env, msg)
    Redom.on_message self, msg
  end

  def on_close(env)
    Redom.on_close self
  end

  def send(msg)
    send_data msg
  end
end
```

## 3.5 Easy distributed programming

The most important characteristic of our web framework is that it is very easy to use it to write distributed programs over server and browser. In this section, we describe the simplicity of Redom for distributed programming.

### 3.5.1 Easy programming for server/browser communication

Redom hides the server/browser communication details from developers, hence it is very easy for developers to write distributed program that access resources at both browser and server sides.

In a interactive web application, there are usually a lot of server/browser communications for exchanging data between server and browser. Developers have to write programs to specify what data is to be exchanged and how to use these data after exchange. Moreover, the details of communication, that is how to send or retrieve data, are also needed to be specified in the program.

For example, assuming that we want to create such simple web application, *Note*. The UI is shown in figure 3.3. When we enter the title of the note in text box Title and click the button Load, the content of a server-side text file with the same name as title will be read and shown in text area Text if the file exists. The last modified time will also be shown at bottom. If the file does not exists, and alert will be displayed. After editing the text of the note, we can save it to the server-side text file by clicking Save button.

If we create this web application using jQuery and PHP, we need create two files, browser-side note.html and server-side note.php. The source code is shown in listing 3.3, listing 3.4 and listing 3.5.

```
1 <HTML>
2   <HEAD>
3     <TITLE>Note</TITLE>
4     <SCRIPT type="text/javascript" src="jquery.js"></SCRIPT>
5     <SCRIPT type="text/javascript" src="redom.js"></SCRIPT>
6     <SCRIPT type="text/javascript">
```

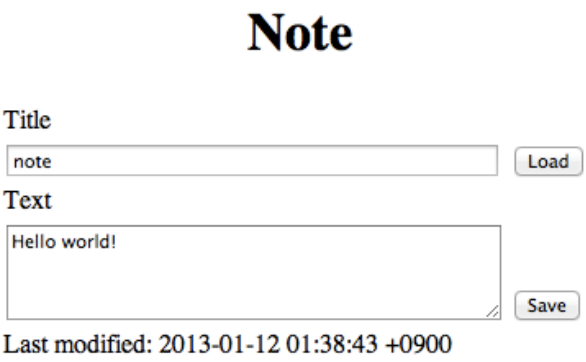


Fig. 3.3. Note web application

```
      // Browser-side program here
    </SCRIPT>
  </HEAD>
  <BODY>
    <TABLE width="100%" height="90%" border="0">
      <TR>
        <TD align="center" valign="middle">
          <TABLE>
            <TR><TD colspan="2"><H1>Note</H1></TD></TR>
            <TR><TD colspan="2">Title</TD></TR>
            <TR valign="bottom">
              <TD width="85%">
                <input type="text" id="title" size="55">
              </TD>
              <TD><input type="button" id="load" value="Load"></TD>
            </TR>
            <TR><TD colspan="2">Text</TD></TR>
            <TR valign="bottom">
              <TD>
                <TEXTAREA id="text" cols="40" rows="4"></TEXTAREA>
              </TD>
              <TD><input type="button" id="save" value="Save"></TD>
            </TR>
            <TR><TD id="lastModified" colspan="2"></TD></TR>
          </TABLE>
        </TD>
      </TR>
    </TABLE>
  </BODY>
</HTML>
```

Listing 3.3. note.html

We extract the browser-side JavaScript program from the HTML file, which is shown in listing 3.4.

```
$(document).ready(function() {
  $("#load").on("click", function() {
```

```
3       load();
4     });
5
6     $("#save").on("click", function() {
7       save();
8     });
9   });
10
11  function load() {
12    $.ajax({
13      type: "POST",
14      url: "note.php",
15      data: { title: $("#title").attr("value") },
16      dataType: "json",
17      success: function(data) {
18        if (data) {
19          $("#text").attr("value", data.text);
20          $("#lastModified").text("Last modified: " + data.lastModified);
21        } else {
22          alert("File not found!");
23        }
24      }
25    });
26  }
27
28  function save() {
29    $.ajax({
30      type: "POST",
31      url: "note.php",
32      data: {
33        title: $("#title").attr("value"),
34        text: $("#text").attr("value")
35      },
36      dataType: "json",
37      success: function(data) {
38        $("#lastModified").text("Last modified: " + data);
39      }
40    });
41  }
```

Listing 3.4. Browser-side program using jQuery

The server-side PHP program is shown in listing 3.5.

```
1   <?php
2     function load($title) {
3       if (file_exists($title)) {
4         $data = array(
5           "text" => file_get_contents($title),
6           "lastModified" => date("F d Y H:i:s.", filemtime($title))
7         );
8         echo json_encode($data);
9       } else {
10        echo "";
11      }
12    }
```

```
13
14   function save($title, $text) {
15     file_put_contents($title, $text);
16     echo json_encode(date("F d Y H:i:s.", filemtime($title)));
17   }
18
19   $title = $_POST["title"];
20   $text = $_POST["text"];
21   if (isset($text)) {
22     save($title, $text);
23   } else {
24     load($title);
25   }
26 ?>
```

Listing 3.5. note.php

As we can see from the source code, developers have to write both browser- and server-side programs in different programming language. As a result, the application logics is separated into two parts and the flow of process becomes unclear. Moreover, developers have to use the AJAX API provided by jQuery to send data to server and retrieve data from server.

However, it is very easy to create such web application using Redom. No browser-side program is needed except a single line of JavaScript code for connecting to the server. The JavaScript code is shown in listing 3.6.

```
1 $(function() { Redom("ws://localhost:8080/websocket").open("NoteConnection"); });
```

Listing 3.6. Browser-side program using Redom

Listing 3.7 shows the server-side program *note.rb* for the note application written using Redom.

```
1  class NoteConnection
2    include Redom::Connection
3
4    def on_open
5      jQuery("#load")[0].onclick { |event| load }
6      jQuery("#save")[0].onclick { |event| save }
7    end
8
9    def load
10     title = jQuery("#title").attr("value").sync
11     if File.exists? title
12       jQuery("#text").attr("value", File.read(title))
13       jQuery("#lastModified").text("Last modified: #{File.mtime(title)}")
14     else
15       alert "File not found!"
16     end
17   end
18
19   def save
20     title = jQuery("#title").attr("value")
21     text = jQuery("#text").attr("value")
22     sync{}
23     open(title, "w") {|f| f.puts text}
```

```
24      jQuery("#lastModified").text("Last modified: #{File.mtime(title)}")
25    end
26  end
```

Listing 3.7. Server-side program using Redom

From the source code, it is obvious that the work of coding has been reduced much. Further, the program becomes highly readable. The figure 3.4 and figure 3.5 show the difference clearly.

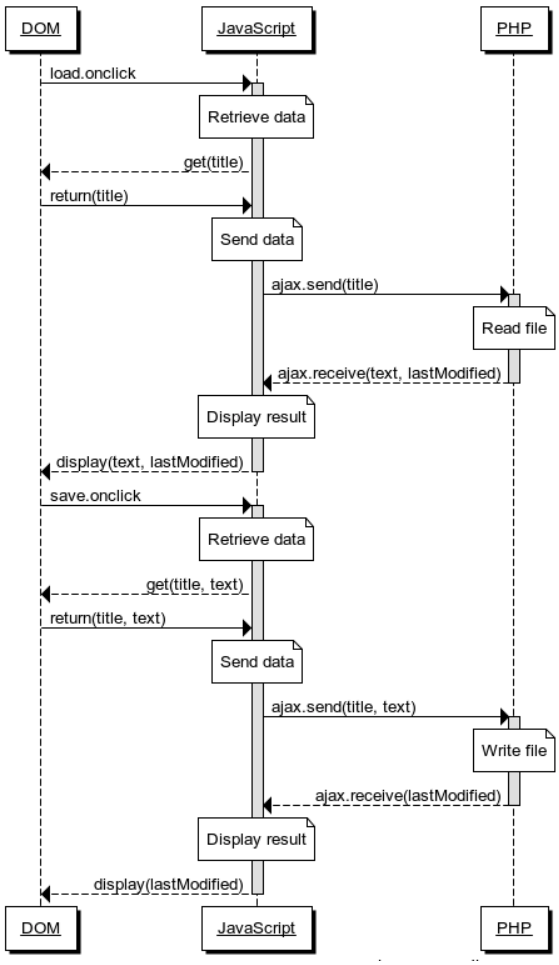


Fig. 3.4. Note by JavaScript and PHP

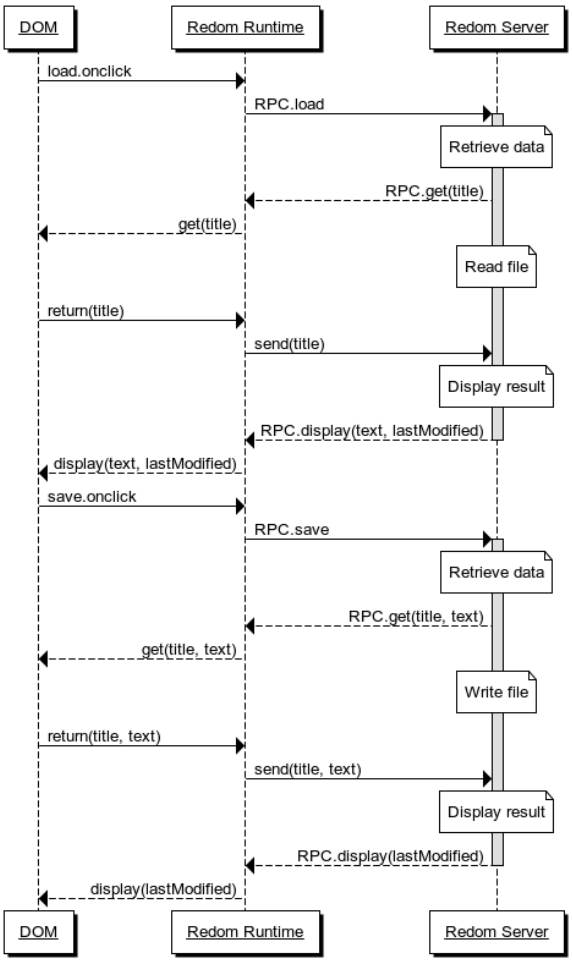


Fig. 3.5. Note by Redom

Figure 3.4 shows the process flow of the program written with JavaScript and PHP. As we can see, the program is divided into two parts over the network. Developers also have to program how to send and receive data. While using Redom, as shown in figure 3.5, only server-side program is needed. Moreover, all communication details are hidden from developer. That is to say, Redom hides the distributed nature of a web application development. Therefore, it is very easy to create web applications use Redom.

### 3.5.2 Easy programming for browser/browser communication

Sometimes a web application may want to send data from one browser to another, for example, a chat web application. A real-time browser-to-browser communication is required

for such operation. Unfortunately, there is no technique that supports direct communication between browsers. Although there are several techniques being researched and developed, such as WebRTC[30], but they are not supported by most browsers and seem to be a little far from practical use. However, the server push allows us to perform a near real-time communication between browsers. Redom uses WebSocket to implement such feature and provides simple API for developers to use.

There is no good solution to browser/browser communication in a traditional AJAX-base web application development because the HTTP connection from browser to server is unidirectional and transient. If a browser wants to communicate with another browser, it has to send data to server and the data must be persisted at server side. Then another browser can retrieve the data by server polling. The lack of server push makes this approach non-real-time and inefficient. However, with server push techniques such as WebSocket, we can perform a near real-time browser/browser communication. Once the server received some data from a browser, it can push the data to another browser immediately, so that it looks like the data is transmitted from a browser to another directly. Figure 3.6 and figure 3.7 describe that how a browser/browser communication is performed with techniques mentioned above.

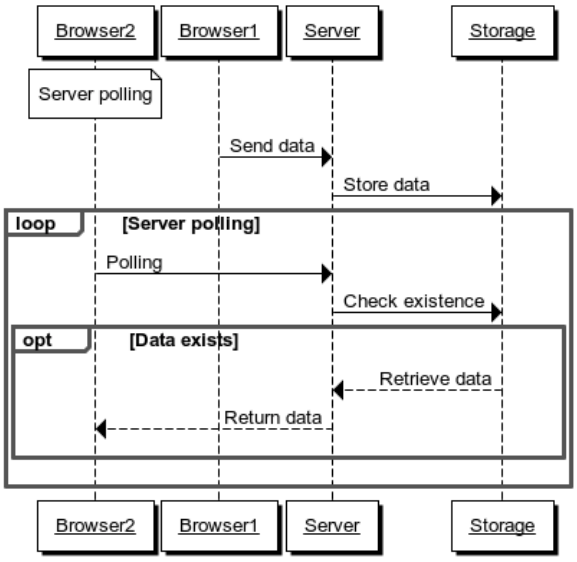


Fig. 3.6. Browser/Browser communication by AJAX

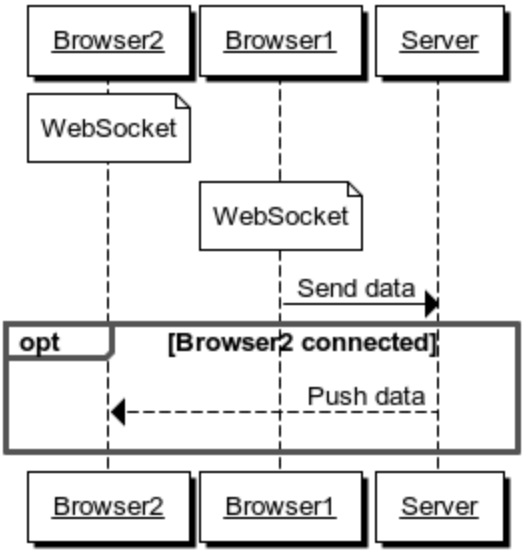


Fig. 3.7. Browser/Browser communication by WebSocket

Since we use WebSocket in Redom, it is possible for developers to write program to perform near real-time browser/browser communication. As is described in section 3.4.7, the coding for browser/browser communication is as easy as a local method invocation. We have shown a example of chat web application in section 3.3, which shows how to perform a browser/browser communication. We show another simple web application here as a example to explain the simplicity of web development using Redom.

We call this web application PingPong. There is only a single text box in the center of the page. Assuming that there are two users connected to the server. An user can type characters in the text box. When an enter key is typed, the text box is disabled and the input text will be sent to the other browser to be displayed in its text box. Then the other user can do the same thing to send back a message. This application can be implemented using WebSocket as follow.

```
<HTML>
  <HEAD>
    <TITLE>PingPong</TITLE>
    <SCRIPT type="text/javascript" src="jquery.js"></SCRIPT>
    <SCRIPT type="text/javascript" src="redom.js"></SCRIPT>
    <SCRIPT type="text/javascript">
      // Browser-side program
    </SCRIPT>
  </HEAD>
  <BODY>
    <TABLE width="100%" height="100%" border="0">
      <TR>
        <TD align="center">
          <input type="text" id="text" size="50">
        </TD>
      </TR>
    </TABLE>
  </BODY>
</HTML>
```

Listing 3.8. pingpong.html

```
$(function() {
  var ws = new WebSocket("ws://localhost:8080/websocket");f
  ws.onmessage = function(event) {
    $("#text").removeAttr("disabled").attr("value", event.data);
  };
  $("#text").on("keydown", function(event) {
    if (event.keyCode == 13) {
      ws.send($("#text").attr("disabled", "").attr("value"));
    }
  });
});
```

Listing 3.9. Browser-side program using WebSocket

```
require 'em-websocket'

EventMachine.run do
  connections = []

  EventMachine::WebSocket.start(:host => "0.0.0.0", :port => 8080) do |ws|
    ws.onopen {
      connections << ws
    }

    ws.onmessage { |msg|
      connections.each { |conn|
        conn.send(msg) unless conn == ws
      }
    }
  end
end
```

Listing 3.10. Server-side program using WebSocket

With Redom, the same program can be written within a single server-side Ruby file as follow.

```
class PingPongConnection
  include Redom::Connection

  def on_open
    jQuery("#text")[0].onkeydown { |event|
      if event.keyCode.sync == 13
        connections.each { |conn|
          unless conn == self
            conn.async.text = jQuery("#text").attr("disabled", "").attr("value").sync
          end
        }
      end
    }
  end

  def text=(text)
    jQuery("#text").removeAttr("disabled").attr("value", text)
  end
end
```

Listing 3.11. Server-side program using Redom

From the comparison of the two implementations in figure 3.8 and figure 3.9, we reach the same conclusion that Redom simplifies the development by centralized program.

### 3.5.3 Easy to debug Redom web application

In a traditional AJAX-based web application, because application logics are separated, it is difficult for developers to find out where the bug comes from when the application works incorrectly. Developers usually have to check both server-side program and browser-side script to locate the code which causes error.

In a Redom application, the debugging is very easy because Redom provides *on_error* method which helps developers to find out the code that causes an error. No matter whether the error occurs on browser side or server side, the error message will be sent to server so that it can be caught within method *on_error* (figure 3.10).

For example, we have a program as follow:

```
class TestConnection
  include Redom::Connection

  def on_open
    text = document.getelementbyid("text").value
  end

  def on_error(error)
    puts error
  end
end
```

Listing 3.12. test.rb

The program will not run correctly because there is a spell miss in method name *getelementbyid*, which is supposed to be *getElementById*. We also define the method *on_error* so

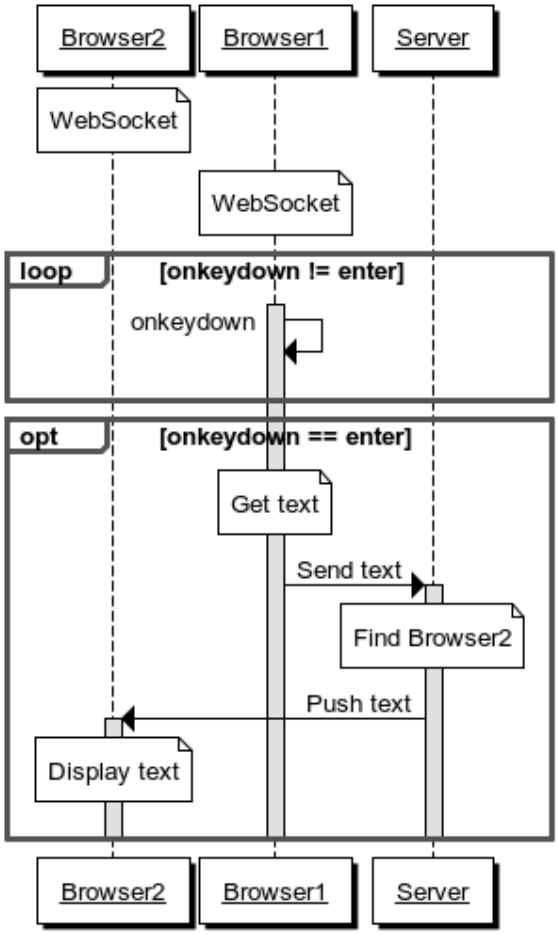


Fig. 3.8. PingPong by WebSocket

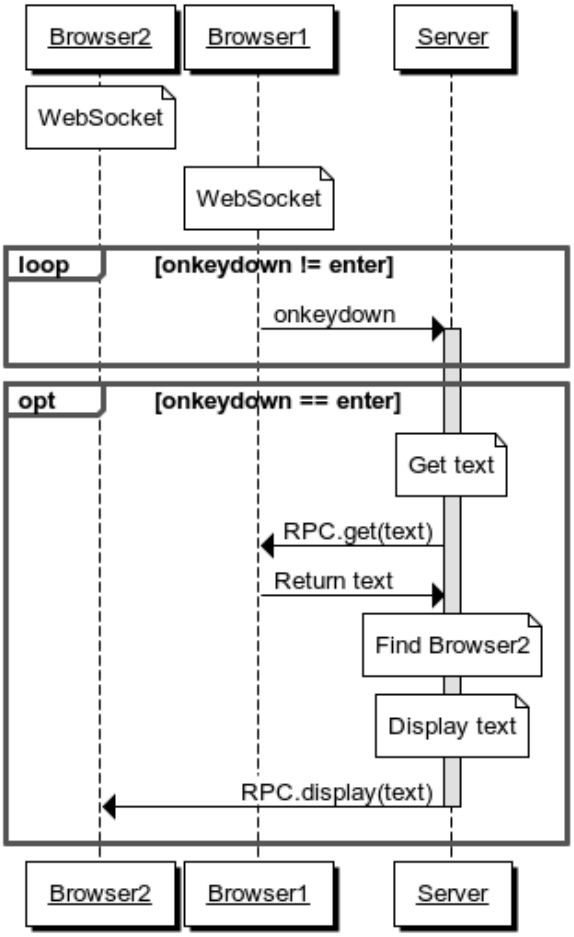


Fig. 3.9. PingPong by Redom

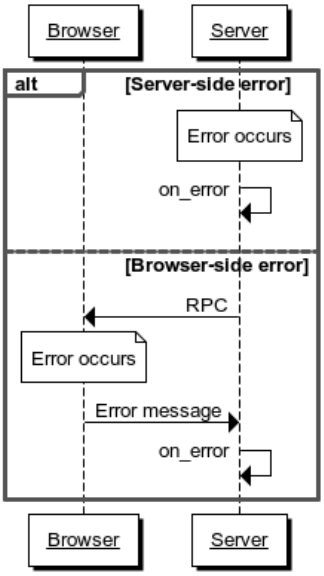


Fig. 3.10. Error handling in Redom

that we can catch the error. When the program is executed, we will get an error message as follow:

Console

```
> No such method 'getelementbyid'.
> /Users/redom/example/test/test.rb:5:in `on_open'
```

We can see that the error message shows what kind of error occurred as well as the location of code which caused the error. Therefore, developers can correct the code easily.

## 3.6 Use cases

The simplicity of Redom for distributed programming outlined in the previous section can become real benefit to developers during the development of web applications. In this section, we describe some use cases that may benefit a lot from Redom.

### 3.6.1 Redom over network

Redom is a web framework that can be used to create almost all kinds of web application. Due to the features provided by Redom, developers can benefit a lot from the simplicity of development with Redom when they create three kinds of web applications shown in figure 3.11.

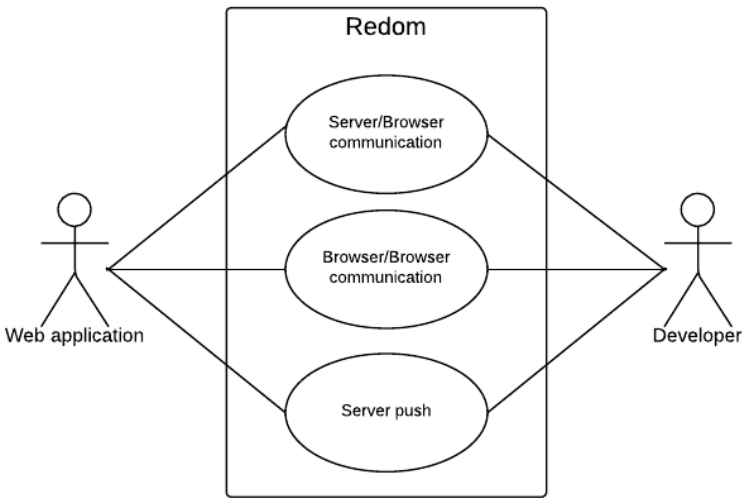


Fig. 3.11. Web applications that are easy for developers to create with Redom

We present example use cases for each kind of web application mentioned above.

- Server/Browser communication
  A simple example of such kind of web application is an online dictionary web application (figure 3.12). User inputs the word they want to know in the browser and then the word is sent to the server. After looking up the word in the database, the server returns the result to the browser. Finally, the result is displayed to the user. By using Redom RPC, such operations are very easily to be implemented by developers.
- Browser/Browser communication

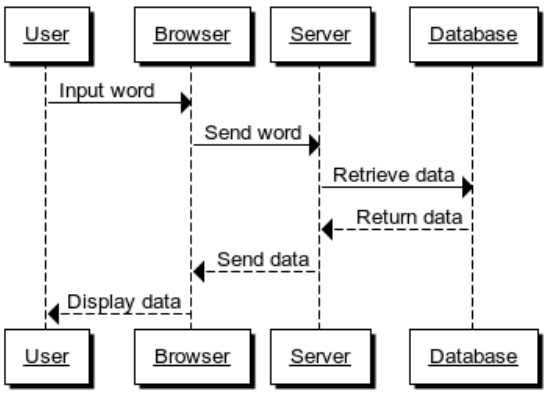


Fig. 3.12. Dictionary web application

A chat web application belongs to such type of web application. Moreover, multi-player game over network is also such web application. For example, a chess web application (figure 3.13). Every time the a piece is moved by a player, the new position of that piece should be sent to the other player's browser. Developers can use Redom API to achieve the purpose easily.

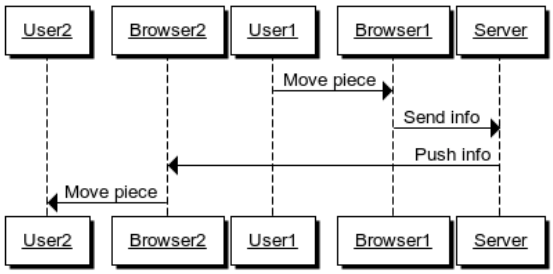


Fig. 3.13. Chess web application

- Server push
  Server push brings better user experience in many cases. A typical example is a auction web application (figure 3.14). Bidders can bid for the goods at any time and the highest price should be informed to all bidders immediately as soon as the price changes. Therefore, it is hard to create such a web application without server push. Redom makes a server push very easy to be performed because Redom keeps all connections from browser so that data can be sent to browsers at any time using simple RPC.

Of course, practical web applications, which may be a combination of several types above, are much more complicated than the examples. However, in either case, Redom will be a good choice for developers to create interactive web applications.

### 3.6.2 Redom in a local environment

Besides the common use of Redom to create web applications, Redom can be used to do some amazing things in a local environment.

- Browser as GUI of Ruby program

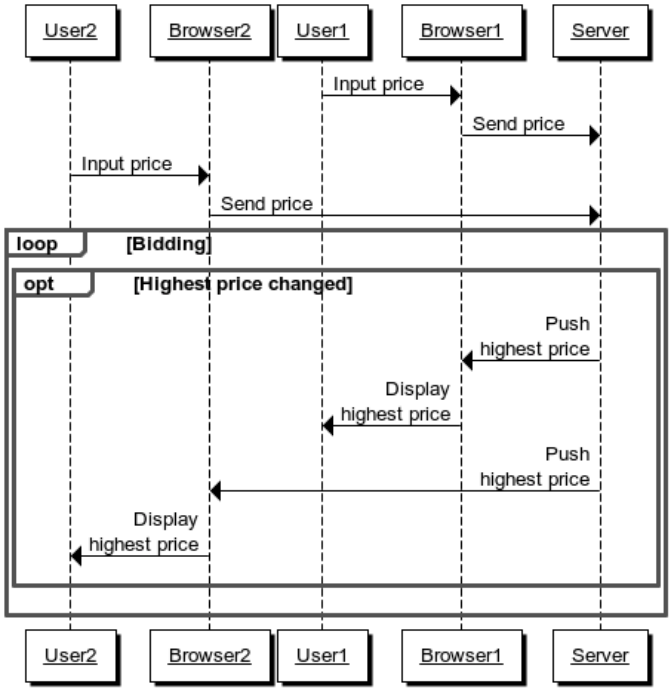


Fig. 3.14. Auction web application

Although there is extension libraries in Ruby for GUI programming, but it is still a good choice to use browser as GUI of Ruby programs (figure 3.15). The reasons are: 1) Browser is platform-independent, 2) Easy to use HTML to create GUI components, 3) Much more expressive GUI using CSS, 4) Powerful JavaScript graphic libraries. Redom enables Ruby program to manipulate browser-side DOM using simple RPC. Therefore, user input can be retrieved from browser and the output of program can be displayed on browser.

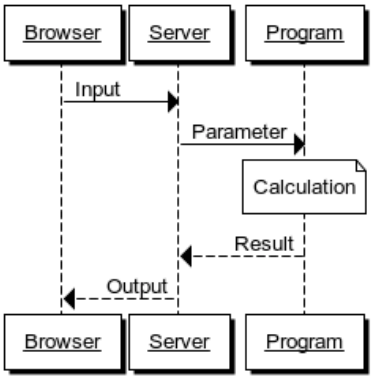


Fig. 3.15. Browser as GUI for Ruby program

- Automatic test for web application
  To test a web application if it works correctly, testers usually apply operations on the web page and check if the browser-side or server-side status changes as expected. It will be a tedious, fallible, time-consuming work for testers to perform

such tests manually. However, Redom provides a easy way to perform such tests (figure 3.16). Since it is possible to manipulate browser-side DOM from server-side program using Redom, testers can write Ruby programs that simulate the user inputs, such as clicking a button. Then testers can check if the alteration to the web page is correct by access browser-side DOM from server-side Ruby program. Once testers have written such test programs, the tests can be performed automatically on multiple browsers across multiple platform, and the test results can be collected at server side for testers to evaluate.

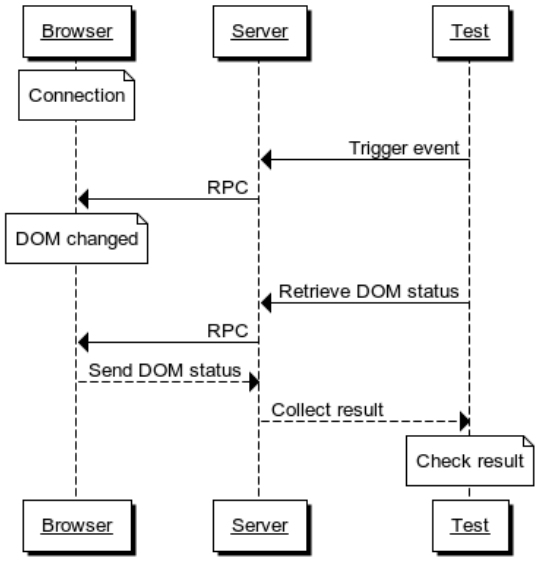


Fig. 3.16. Automatic test for web application

# Chapter 4

# Improving performance of Redom

In last chapter we introduced Redom, a web framework we developed to simplify and accelerate web development. We have shown the features of Redom which lead to a high productivity in web development. However, we encountered a performance issue in Redom. In this chapter, we describe the problem encountered and the effort we made to solve it.

## 4.1 Improving performance of RPC

This section presents the performance issue while performing RPC from server to browser, together with the solution we used.

### 4.1.1 Performance issue with synchronous RPC

A synchronous RPC refers to an RPC that blocks current process after sending message to the remote node and will not relieve the block until the result of execution on the remote node is received. Therefore, every synchronous RPC is accompanied by a server/browser communication. As is known to all, the overhead of a server/browser communication is quite high. If the synchronous RPC occurs frequently, the performance of the web application will be affected greatly due to the overhead of considerable communications.

For example, we have a piece of code as below:

Ruby

```
text = document.getElementById("text1").value
document.getElementById("text2").value = text
```

This piece of code copies the content in text box with id "text1" to the text box with id "text2". When executed, the code will cause four RPCs from server to browser, which are *document.getElementById("text1")*, *value*, *document.getElementById("text2")* and *value=*. Four times of communication between server and browser will be performed to complete these RPCs. We compared the execution time of the operation above using JavaScript and RPC. The result is shown in table 4.1.

| | How many times executed | | | |
|---|---|---|---|---|
| | 1(4 RPCs) | 2(8 RPCs) | 4(16 RPCs) | 8(32 RPCs) |
| JavaScript | <1ms | <1ms | <1ms | <1ms |
| Synchronous RPC | 372ms | 740ms | 1491ms | 3117ms |

Table. 4.1. Execution time of DOM manipulation using JavaScript and RPC

From the result, we can see that the overhead of synchronous RPC is extremely high. The high overhead makes user feel a long latency and affects the performance of the entire web application greatly.

### 4.1.2 Batched futures

In order to solve the problem, we introduce *batched futures* into Redom. The definition of batched futures in [31] is as follow:

> The basic idea is that certain RPCs are not performed at the point the server triggers them, but are instead deferred until the server actually needs the value of a result. By that time a number of deferred calls have accumulated and the calls are sent all at once, in a " batch ". In this way we can turn N server/browser communication into one, and user code runs faster as a result. Our mechanism makes the batching transparent to server applications and allows later calls to make use of the results of earlier RPCs.

Every time an RPC is detected in Redom, a future object with the information of the RPC will be created. Then any subsequent operation to this future object, such as an method invocation, will be taken as an RPC. All future objects are stored in a queue in order. When the queue is full, the synchronization starts. The accumulation of RPCs is performed as shown in figure 4.1.

When the synchronization starts, the server will serialize all future objects into a single message and send the message to browser. After receiving the message, the browser first unserializes the message and retrieves the RPC information contained in the future objects. Then these RPCs are performed in the order when they are created. The results are stored in a proxy queue for future use. The results will also be serialized and sent back to the server. The server gets results from unserialized message and replace the future object with corresponding result. At last, the future object queue is emptied. The process flow of batched futures is shown in figure 4.2.

### 4.1.3 Synchronization with browser

The server have to communicate with browsers to get the results of RPCs, which we called a synchronization. The timing for a synchronization is shown below.

- When the amount of future objects have reached a limit
  As we mentioned in previous section, when the queue is full of future objects, a synchronization will be performed to retrieve the result of RPCs. The size of the queue can be defined when the server starts.
- When it has reached the end of a task
  When a Redom task has been completed, for example, the process inside an event handler have been completed, a synchronization will be performed so that the DOM manipulations can be executed immediately.
- When the value of RPC result is needed
  Sometimes, the server need the value of RPC result for follow-up process, for example, a database query need the user input. At such a moment, a synchronization is required to retrieve the result of RPC immediately. However, Redom has no idea when the value of RPC is needed. Therefore, developers have to use *sync*{} to trigger the synchronization explicitly in program.

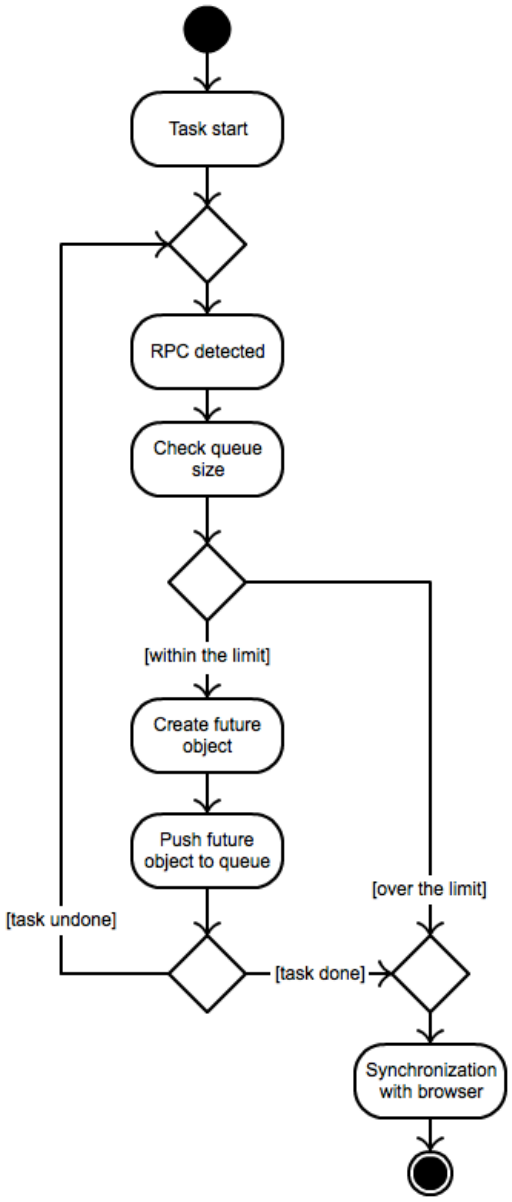


Fig. 4.1. Accumulation of RPCs

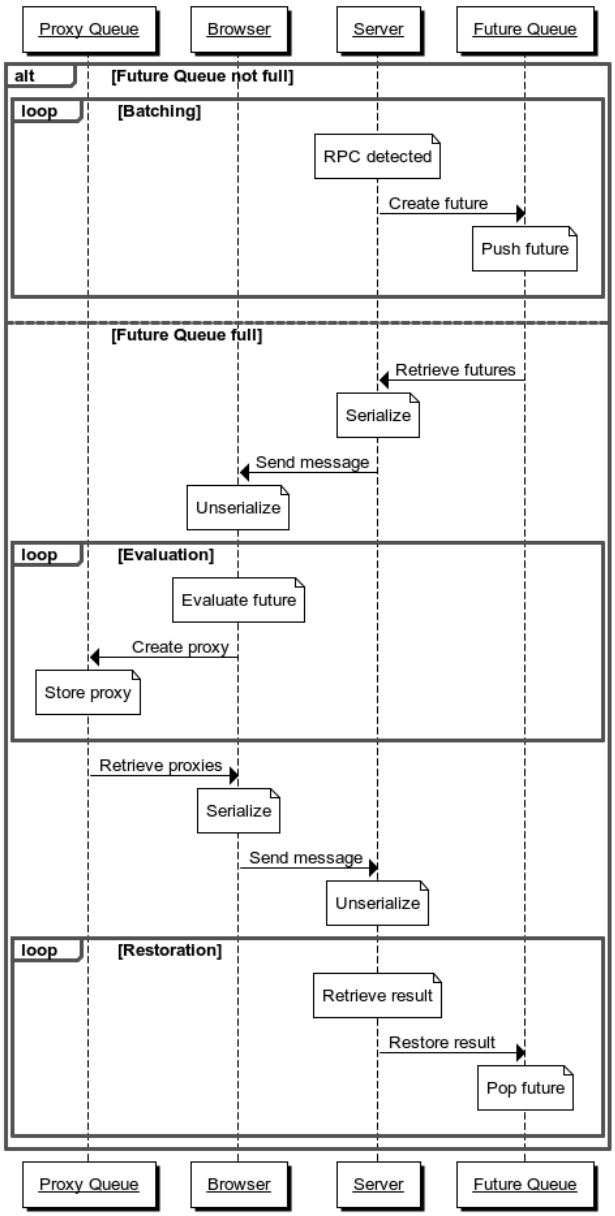


Fig. 4.2. Process flow of batched futures

## 4.2 Improving performance of DOM manipulation

By using batched futures, the performance of Redom has been greatly improved. Developers can use Redom to create a web application easily with adequate performance. However, it is possible for developers to make a little more effort to obtain better performance. We have noticed some facts of DOM manipulation that can be used to improve the performance of Redom. In this section, we describe these facts and how we make use of them to improve the performance of Redom.

### 4.2.1 Performance issue with DOM manipulation

By creating web applications using Redom, we have found that the DOM manipulation is an important factor that affects the performance of Redom web application. The reason is because in the server-side Redom program, DOM manipulations cause server/browser communications.

Another important factor that affects the performance is frequent browser-side user events. Once a browser-side event occurs, the server-side event handler (if defined) will be invoked via RPC, which cause a browser/server communication.

For example, there are two text boxes on the page. Every time a character is typed in one text box, the same text will be shown in the other text box. The program written with Redom are as follow.

Ruby

```
def copy_text(e)
  text = jQuery("#text1").attr("value")
  jQuery("#text2").attr("value", text)
end
jQuery("#text1")[0].onkeydown = :copy_text
```

Because *keydown* event is triggered every time user types a character in the text box, when user types characters continuously, there would be a lot of RPCs from browser to server to invoke the event handler *copy_text*. Each RPC requires a browser/server communication (figure 4.3). Therefore, there may be a obvious latency to user.

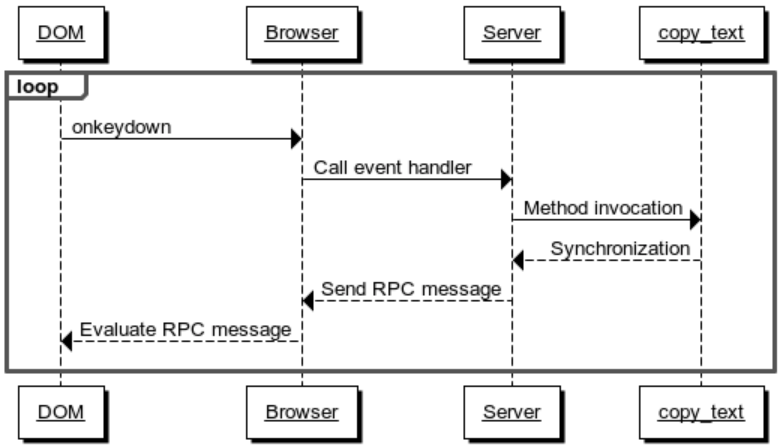


Fig. 4.3. Event handling in Redom

We noticed another fact that the DOM manipulations inside method *copy_text* do not use server-side resources, namely access to server-side API, methods, variables, etc. If we define the event handler at browser side using JavaScript, it will cause no browser/server communication when it is called (figure 4.4).

### 4.2.2 DOM manipulation can be centralized

An event handling in web application is generally processed as follow:

1. Receive an event object.

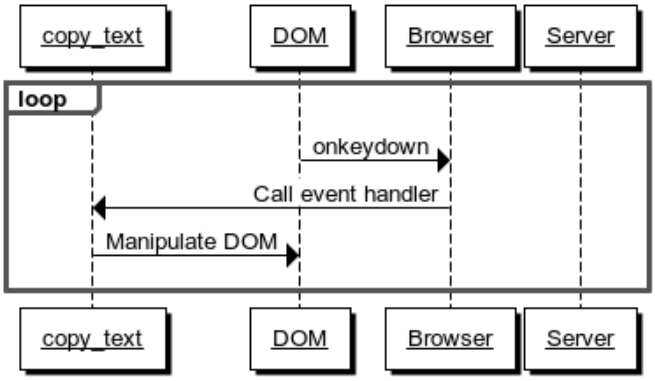


Fig. 4.4. Event handling in JavaScript

2. Get user input from the event object or DOM (DOM manipulation).
3. Retrieve data from server or send data to server.
4. Make change to DOM (DOM manipulation).

We can see that there are DOM manipulations in step 2 and 4. However, there is not always a step 3 in an event handling, that is to say step 3 is not required in many cases. The example code in section 4.2.1 illustrates this point. Therefore, if we use JavaScript to handle such events, there will be no browser/server communication and the performance can be improved.

Because Redom is a single-language web framework based on Ruby, we introduce Ruby to JavaScript compiler into Redom so that developers can write all code in Ruby while a portion of the code can be compile to JavaScript to be run at browser side. However, we can not compile all Ruby code to JavaScript. Because JavaScript is a browser-side programming language, there is no support for JavaScript to access server-side resources. Moreover, it is impossible for JavaScript to access browser-side resource from browser directly for security reasons. Therefore, we have to use Ruby to do server-side operations, such as access to the database, file system and so on. We only compile the code of DOM manipulation that requires no server-side process to JavaScript.

### 4.2.3 Compile Ruby to JavaScript using Opal

We used Opal[16] in Redom to compile Ruby code to JavaScript code. Opal is a source-to-source ruby to javascript compiler which has an implementation of the ruby corelib as we introduced in section 2.2.1. It is very easy to use *Opal.parse* to parse Ruby to JavaScript. For example, the following code

Ruby

```
src = %Q{
  jQuery("#text1")[0].onkeydown {
    text = jQuery("#text1").attr("value")
    jQuery("#text2").attr("value", text)
  }
}
puts Opal.parse(src)
```

will output JavaScript code as follow.

JavaScript
```
(function() {
  var __opal = Opal, self = __opal.top, __scope = __opal,
  nil = __opal.nil, __breaker = __opal.breaker, __slice = __opal.slice;
  var __a, __b;
  return (__b = self.$jQuery("#text1")['$[]'](0),
    __b.$onkeydown._p = (__a = function() {

    var text = nil;

    text = this.$jQuery("#text1").$attr("value");
    return this.$jQuery("#text2").$attr("value", text);
  }, __a._s = self, __a), __b.$onkeydown())
})();
```

Then we can use *window.eval* to execute the JavaScript code at browser side.

As we mentioned in previous section, we can not compile all Ruby code to JavaScript. Only Ruby code of DOM manipulations without access to server-side resources may be compiled to JavaScript. There are two cases that developers can consider to compile Ruby to JavaScript to obtain better performance.

- A large number of centralized DOM manipulations
  For example, a piece of code wants to draw one thousand points on a browser-side canvas. If using RPC, although we have batched futures will reduce the server/browser communications to a very low amount, communication will still happen due to the limit size of batched future queue. In such case, using JavaScript code will cause no communication between server and browser.
- Events that occurs frequently but requires no access to server
  We have discussed such situation in previous section. If we defined the event handler of such events at browser side, the event will be handled without any browser/server communication/

### 4.2.4 Allowing browser-side Ruby to access DOM

We can use Opal to compile Ruby code to JavaScript code. However, there is a serious problem that the Opal-compiled code is not able to access DOM. This is because Opal does not provide DOM manipulation APIs. Moreover, all the method names of Ruby methods have been wrapped as a Opal method. For example, if the following Ruby code

Ruby
```
text = jQuery("#text1").attr("value")
```

is compiled to JavaScript by Opal, we will get

JavaScript
```
text = this.$jQuery("#text1").$attr("value");
```

As we can see, the method *jQuery* has been converted into *$jQuery* which does not exists on both server and browser sides. When the JavaScript code is executed, an exception will be raised because JavaScript can not find the method *$jQuery* in Opal runtime though *jQuery* exists in DOM (figure 4.5).

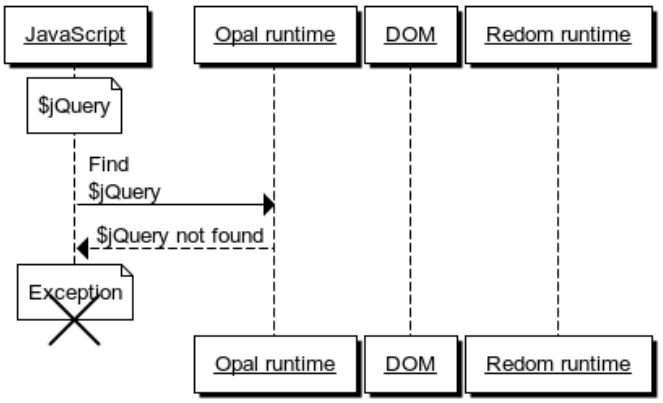


Fig. 4.5. Method invocation in Opal-compiled Javascript code

In order to enable Opal-compiled code to access DOM, we extended Opal to have a method_missing-like feature so that JavaScript is able to find the correct method at browser side. We alter the compiler to wrap every method with a check method named *$redomCall*. The compiled JavaScript is like:

JavaScript
```
text = self.$redomCall('$jQuery')("#text1").$redomCall('$attr')("value");
```

Then when the code is executed, the $redomCall method will get the proper method for JavaScript to evaluate the code as expected (figure 4.6).

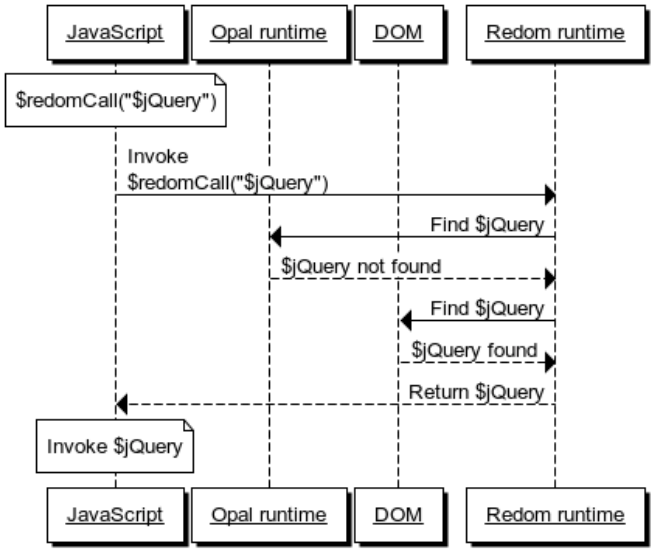


Fig. 4.6. Method invocation in Opal-compiled Javascript code with Redom

### 4.2.5 Code sharing between server and browser

Because the only purpose that we introduce Ruby-to-JavaScript compilation into Redom is to improve the performance further, it should not sacrifice the productivity to achieve

this. Therefore, the Ruby code previously written should be able to work correctly at browser side after being compiled to JavaScript. That is to say, Ruby code can be shared between server and browser.

To enable code sharing, we implemented method *require* in Redom that allow Ruby code to be compiled to load other Ruby code into itself. For example, we have a Ruby file *person.rb* with the class Person defined in it.

```ruby
class Person
  def initialize(name)
    @name = name
  end

  def talk
    puts "Hi, I'm #{@name}."
  end
end
```

Listing 4.1. person.rb

Then we can use this file on either or browser as follow.

```ruby
require './person'

class TestPersonConnection
  include Redom::Connection

  def on_open
    john = Person.new("John")
    john.talk

    src = %Q{
      require './person'
      jack = Person.new("Jack")
      jack.talk
    }
    window.eval(parse(src))
  end

  def on_error(error)
    puts error
  end
end
```

Listing 4.2. test_person.rb

We require *person.rb* at server side and also, in the code to be compiled to JavaScript, we required it as well. When running the application, the output at server side will be:

Console

```
> Hi, I'm John.
```

The output at browser side will be:

Console

```
> Hi, I'm Jack.
```

We can see that the class Person can be reused at both server side and browser side. Therefore, by using this feature, developers can create web applications using Ruby for the prototype. If better performance is required, the Ruby code can be compiled to JavaScript with no extra effort.

# Chapter 5

# Implementation

There are many existing techniques for the development of a web framework, from which we picked several techniques to implement our proposed web framework. In this chapter, we explain the reason why we chose them and how we combined them together to work well in Redom.

## 5.1 Server/Browser connection

The server/browser connection is the foundation of a web application. Without a connection, the communication between server and browser can not be performed and nothing else can be done. In this section, we describe how we implement the server/browser connection.

### 5.1.1 Communication protocol

To enable server/browser communication, we first need a communication protocol to define the way how communication can be carried out. One of the most commonly used application layer protocol for server/browser communication is HTTP protocol (Hypertext Transfer Protocol). Due to its simplicity of use, numerous techniques use HTTP as the protocol to transfer or exchange data over network, one of which is AJAX.

HTTP is a request-response protocol for a client-server model, which means that the server will not return a response message until it receives a request from the client. For a web application using HTTP protocol, requests can be sent from browser (regarded as client) to a server, but not vice versa. In our proposed web framework, there is a need for sending messages from serve to browser so that RPC can be executed. Therefore, if we want to use HTTP in Redom, we need techniques that enable a server to push data to a browser over HTTP whenever necessary.

*Comet*[32] is a term which refers to techniques for server push over HTTP. There are many implementations of Comet. One of these implementations is long-polling. By using a long-lived HTTP request, the server is able to send a response to browser at any time while the HTTP connection is alive. Because AJAX is able to send such HTTP request, a server-push web application can be created easily using AJAX long-polling. However, the shortcomings, for example, the long-lasting HTTP connections take much server resource and the overhead of reconnecting when the previous connection times out is very high, prevent Comet to be widely used in practical web applications.

*HTML5* brings us a new technology called *WebSocket* to achieve the purpose of server push[33]. WebSocket is a protocol that enables full-duplex communication over a TCP connection. We focus on two characteristics of WebSocket which we require to implement Redom. Bi-directional communication is what we want to allow the server to send

messages to browser directly. The other one is high efficiency. Once a WebSocket connection has been established through a HTTP Upgrade request as the handshake sent from browser to server, data is exchanged over a TCP socket instead of HTTP request/responses. Thus, no further HTTP request overhead occurs during the life time of the WebSocket connection. As for the shortcoming of WebSocket, it is that not all browsers support WebSocket. However, at the moment of writing this thesis, we can see from figure 5.1 that most of the latest browsers support WebSocket. Therefore, we choose WebSocket as the communication protocol in Redom.

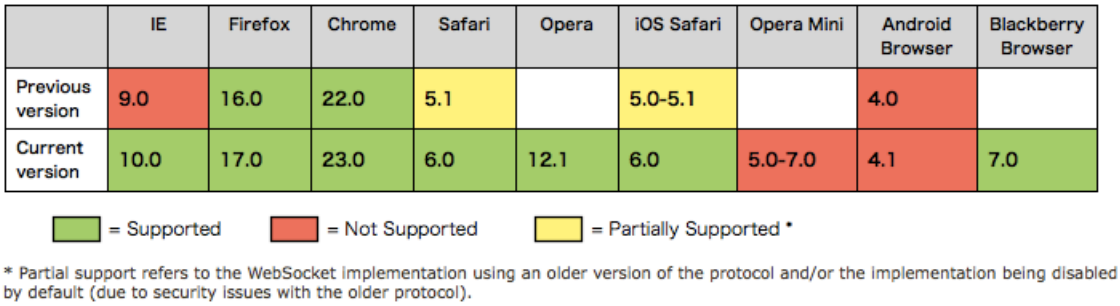


Fig. 5.1. Browser support for WebSocket[34]

Though we used EM-WebSocket[35] in Redom as the default WebSocket server, it is possible to use other WebSocket server instead as we mentioned in chapter 3.

With the WebSocket API standardized by the W3C (World Wide Web Consortium)[*1], we can create and use a WebSocket connection easily. The WebSocket interface is shown below.

```
[Constructor(DOMString url, optional (DOMString or DOMString[]) protocols)]
interface WebSocket : EventTarget {
  readonly attribute DOMString url;

  // ready state
  const unsigned short CONNECTING = 0;
  const unsigned short OPEN = 1;
  const unsigned short CLOSING = 2;
  const unsigned short CLOSED = 3;
  readonly attribute unsigned short readyState;
  readonly attribute unsigned long bufferedAmount;

  // networking
            attribute EventHandler onopen;
            attribute EventHandler onerror;
            attribute EventHandler onclose;
  readonly attribute DOMString extensions;
  readonly attribute DOMString protocol;
  void close([Clamp] optional unsigned short code, optional DOMString reason);

  // messaging
            attribute EventHandler onmessage;
            attribute DOMString binaryType;
  void send(DOMString data);
```

[*1] http://www.w3.org

```
  void send(Blob data);
  void send(ArrayBuffer data);
  void send(ArrayBufferView data);
};
```

Listing 5.1. The WebSocket interface

WebSocket connection is opened by using the constructor in JavaScript. After a WebSocket object is created, the event handlers, *onopen*, *onclose*, *onmessage* and *onerror*, can be used for handling WebSocket events. And data can be transmitted via method *send(data)*. Usually, the server-side WebSocket object to represent the connection has a same or similar interface. For example, a WebSocket connection object in EM-WebSocket provides the same methods for the server-side program to handle WebSocket events at server side and send data to the browser.

### 5.1.2 Data interchange format

In order to perform RPC between server and browser, we need a proper data interchange format to transmit structured data. We considered two of the most widely used formats XML and JSON, which are supported by both JavaScript and Ruby. And we compared them in order to know which format is more proper for us.

The first thing we compared is data size. It is obvious that the larger the data size is, the longer the time it takes for transmitting. Therefore, we need a smaller data size for a better performance. Assuming we have such a *person* object of which the field "name" is "John" and "age" is 20. We formatted it into XML and JSON as shown below.

| Format | Text | Length |
|---|---|---|
| XML(form 1) | `<person><name>John</name><age>20</age></person>` | 47 |
| XML(form 2) | `<person name="John" age="12"/>` | 30 |
| JSON | `{"name":"John","age":20}` | 24 |

Table. 5.1. Data size comparison between XML and JSON

As we can see in table 5.1, the size of XML data is larger than JSON because the markup nature of XML leads to redundant information in the data. In addition, it is very easy to parse JSON data into typed object with either JavaScript or Ruby because JSON supports basic data types in both language, while complicated operations are required only to parse XML data into untyped DOM tree. Therefore, we use JSON as the data interchange format in Redom in order to obtain good performance due to efficient data transmission.

### 5.1.3 Redom connection object

This section describes the actions at server side when a connection is established. When an WebSocket connection is established, a WebSocket connection object is created by the WebSocket server. Any subsequent data transmission is performed via the two WebSocket objects. We wrap that WebSocket object in a Redom connection object. A Redom connection object is created from a connection class which includes Redom::Connection module. Different Redom connection classes correspond to different services. Redom::Connection module provides event handlers described in section 3.4.5 and enhances the connection class to be able to perform RPC. The scenario below describes what happens when a browser connected to the server.

1. After the connection is established, the event handler *onopen* of the browser-side WebSocket object is invoked and a handshake request is sent to the server. The request contains the name of a Redom connection class which defines what should be done to the page.
2. When server-side WebSocket object have received the handshake request, an instance of the Redom connection class specified in the request is created. The connection object wraps the WebSocket object in it and will live until the WebSocket connection is closed. For each WebSocket connection, only one Redom connection object is created.
3. The server stores the created Redom connection object. Then the *on_open* method of the Redom connection object is invoked. The application logics defined in that method will be executed.
4. When an RPC occurs to access browser-side objects while executing, RPC message is sent via the WebSocket object kept in the Redom connection object. The RPC message contains the id of the Redom connection object. And when the result of RPC is sent back to the server, with the id contained in the result, the Redom connection object is retrieved and the execution proceeds using that object.

### 5.1.4 Concurrency

The nature of web application requires that it can serve or interact with several users concurrently[1]. The server should respond to requests from browser in paralle, otherwise the user experience will deteriorate due to the high latency. Multithreading is a technique to achieve the purpose. By using multiple threads created at server side, each request can be processed on a single thread without interfering each other. In our case, for each WebSocket connection, there should be a thread on which application logics can be executed.

Since Ruby supports multithreading, it is possible to implement Redom using a multithreading model. The multithreading model may work well when there are only small number of connections. However, when the web application scales to service a large number of users, we have to face performance issue and the *C10K problem*[7]. The C10K problem refers to the problem of a web server that can not handle a large number of clients simultaneously. The following reasons cause the problems.

- Thread creation is a very expensive operation on Ruby. The overhead of creating a thread whenever a connection is established will be extremely high.
- There is a limit to the number of threads that can be created.
- A large number of threads will consume large amounts of system resources.

To solve these problems, we introduce *Thread Pool Pattern* in Redom. A thread pool is usually a queue which contains a certain number of threads. These threads are created in advance and can be reused. A task to be performed is passed to a idle thread, and then be executed on that thread. Once the thread complete the task, it becomes idle again to wait for next task.

In Redom, a Redom::ThreadPool instance is created when the server starts up. The ThreadPool instance contains an array of Redom::Worker objects. A Redom::Worker object contains a queue of tasks and a thread on which tasks are performed. The number of worker objects to be created can be specified before the server starts up. No more Redom::Worker instance, namely thread, will be created at runtime. Therefore, there is no overhead of thread creation at runtime and the performance as well as scalability of Redom is improved.

## 5.2 Event-driven programming

As wen mentioned before, we used event-driven programming as the programming paradigm in our framework. This is because in a web application created by Redom, the browser can be taken as GUI of the server-side program, and the major work is reaction to user inputs and actions. Thus it is simple for developers to create interactive web application with flexible and readable event-driven programs. In this section, we describe how we implement event-driven programming in Redom.

### 5.2.1 Reactor pattern

There are many methods to implement event-driven programming, one of which is *reactor pattern*. Reactor pattern uses a dispatcher to demultiplexes incoming events synchronously on a single thread, and dispatches them to their associated event handlers. Since we used thread pool to manage worker threads and it is convenient to use reactor pattern to dispatch tasks to a certain worker thread, we choose reactor pattern to implement that how Redom handles events.

The figure 5.2 shows the flow how event is processed.

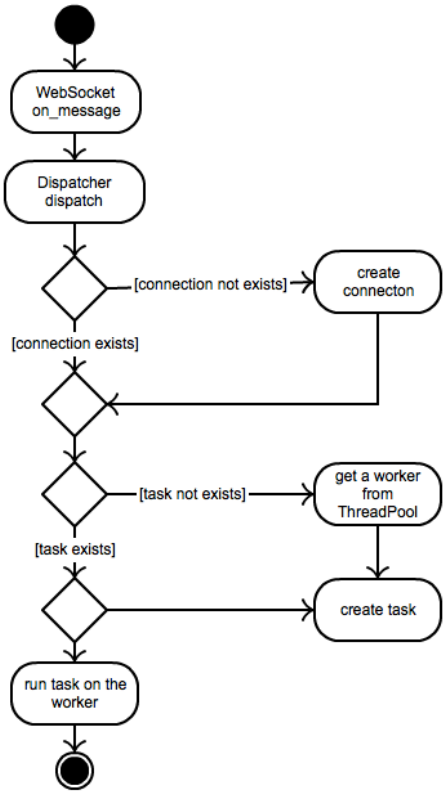


Fig. 5.2. Activity diagram of event handling in Redom

### 5.2.2 Events in Redom

All the events in our framework that the dispatcher receives can be separated into 3 categories.

- HANDSHAKE
  This event occurs when a web page tries to connected to a Redom server. This event contains the name of connection class in which application logics are defined for this page. This event is triggered at browser side.
- RPC
  This event occurs when an RPC is about to be performed. This event contains the id of the Redom connection object of this WebSocket connection, and the method name as well as arguments for this method. All browser-side DOM events are wrapped in this event. Not only browser-side user interaction, but also the action when a connection invokes method of another connection can trigger this event.
- RPC_RETURN
  This event occurs after the end of the execution of a RPC at browser side. This event contains the id of the Redom task object and the result of RPC. This event is triggered at browser side.

### 5.2.3 Event handling

The figure 5.3 shows the class diagram of event handling in Redom.

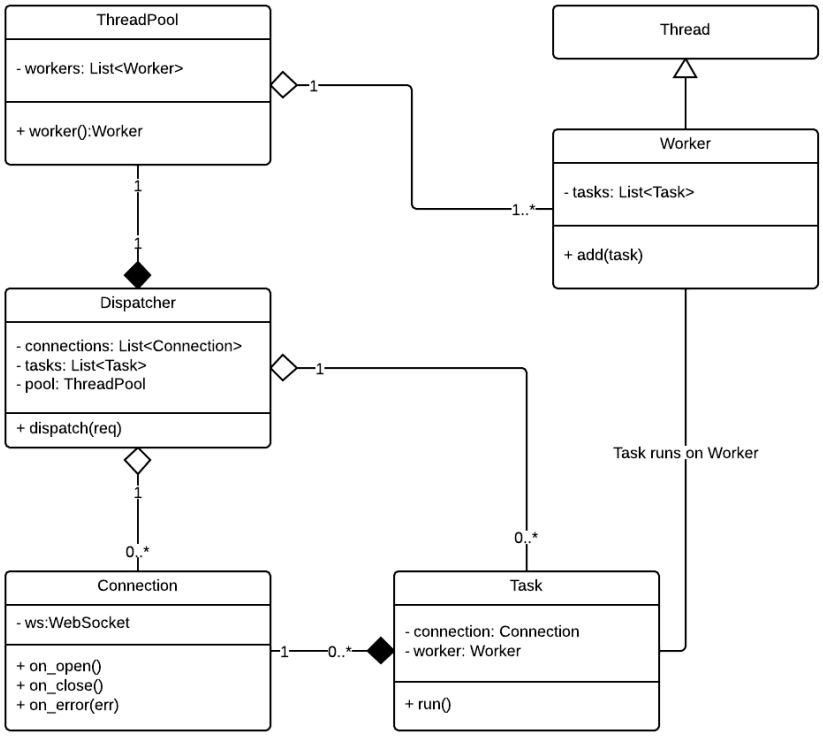


Fig. 5.3. Class diagram of event handling in Redom

## 5.3 Implementation of RPC

RPC is the core part of our framework because it is the only thing connected server-side program with browser-side objects. The most server/browser communications are caused by RPC. Therefore, the implementation of RPC is closely related with the performance of Redom. In this section, we describe how we implement the RPC to be efficient in Redom.

### 5.3.1 Detecting an RPC

As we described before, developers use simple RPC to access browser-side objects or API. The first problem we should solve is how we can tell if a method invocation is an RPC or just a local method call. To achieve this purpose, one of the metaprogramming features in Ruby called *method_missing* helps us a lot.

method_missing is a method which will be called when a method on the receiver can not be found. The name of called method, arguments to the called method and the block passed to the called method are received by method_missing as arguments. Because the browser-side objects or API are not defined at server side (assuming that there is no local method or variable shared the same name), when developers use an RPC to access them, the method_missing will be invoked as a result of invocation of an unknown method. Therefore, we can know that an RPC has been performed. Then we can create an RPC message from the arguments of method_missing to be sent to browser side where the real method invocation is performed.

Just one problem is that if there is a local method or variable whose name is the same as a browser-side object or API, the local method will be invoked even though an RPC is supposed to be performed. However, with just a little attention while programming, it will hardly trouble developers.

### 5.3.2 RPC message

An RPC message is a JSON-formatted string which is sent to browser side and evaluated. The RPC message is serialized from an Ruby array which looks like:

RPC message

```
[id, receiver id, method name, [argument, ...]]
```

The arguments in an RPC message can be any type that JSON supports. Further, in order that arguments can be passed correctly, for example, the event handler can be assigned to browser-side object, we extended JSON to express more types. Table 5.2 shows the type conversion between Ruby/JavaScript types and JSON types.

### 5.3.3 Execution at browser side

The browser-side Redom runtime is responsible for evaluating the RPC message after it have been received. The Redom runtime takes the steps shown in figure 5.4 to evaluate the RPC message.

### 5.3.4 Batched futures

As we mentioned before, we used batched futures to improve the performance of Redom. Every time an RPC is detected, it is not performed immediately. Instead, a future object,

| Ruby/JavaScript type | JSON type | Example |
|---|---|---|
| String | String | "foo" → "foo" |
| Number | Number | 1 → 1, 0.5 → 0.5 |
| Boolean | Boolean | true → true, false → false |
| Nil(Ruby),Null(JavaScript) | Null | nil → null |
| Array | Array | [1,2] → [2, [1,2]] |
| Hash(Ruby), Object(JavaScript) | Object | {"a":1, "b":2} → {"a":1, "b":2} |
| Symbol, Method, Proc(Ruby) | Array | method(:m) → [4, "m"], :foo → [4, "foo"] |
| Redom Proxy(Ruby) | Array | proxy → [1, "id"] |
| Error(JavaScript) | Array | error → [3, "error message"] |
| Undefined(JavaScript) | Array | error → [0] |

Table. 5.2. Type conversion between Ruby/JavaScript and JSON

which contains all information that is needed to perform the RPC, is created. The created future object can be used as the receiver of subsequent method invocation, or an argument of an method invocation. All these future objects are accumulated in a queue and when the queue is full, the synchronization starts to perform RPCs. The class diagram of batched futures is shown in figure 5.5.

### 5.3.5 Suspending and resuming task

Batched futures is performed in a task as shown in figure 5.5. When a synchronization starts, the current task must be suspended to wait for the results of RPCs. After receiving the results of RPCs from browser, the task must be resumed to continue the remaining work. To implement this, we introduce Fiber into Redom. Fiber is a light weight cooperative concurrency in Ruby. Every Task object has a fiber in it. All process in a task is performed in that fiber object. We can use *Fiber.yield* method to suspend the current task when a synchronization is needed. When we received the result of RPCs, we can use *resume* method of the fiber object to resume the task. The process flow of suspending and resuming task is shown in figure 5.6.

## 5.4 Ruby-to-JavaScript compilation

To provide developers a choice of acquire a higher performance, we introduce Ruby-to-JavaScript compilation into Redom. We used Opal as the compiler in Redom, but there is a few problems such as Opal-parsed JavaScript code is not able to access browser-side objects. So we alternated Opal to work with Redom well.

### 5.4.1 Compilation at server side

As we described before, we wrapped every method with *$redomCall* method so that JavaScript can find the correct method to be invoked at browser side. The wrap is performed when parsing Ruby code to JavaScript. There is a Parser class in Opal whose instance is used for parsing. We override the method that parses Ruby method to JavaScript method and alter method to add $redomCall to every method name. Therefore, JavaScript will know how to find the method via $redomCall.

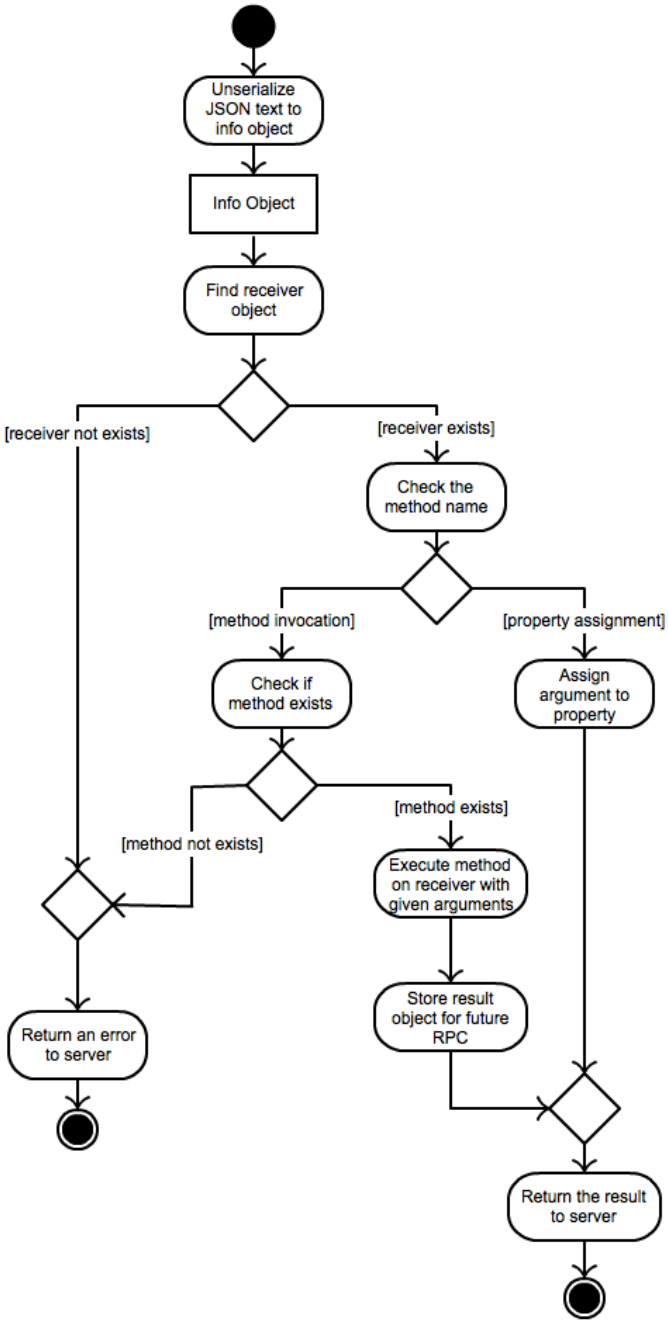


Fig. 5.4. Evaluating RPC message

### 5.4.2 Execution at browser side

We implemented method $redomCall at browser side. The $redomCall method takes a method name string as argument. When $redomCall is called, it will first check if the required method is defined in Opal runtime. If true, that method will be retrieved and executed, otherwise, $redomCall will look for the method in window scope to see if it is defined in a JavaScript library or in embedded JavaScript code inside HTML. If the

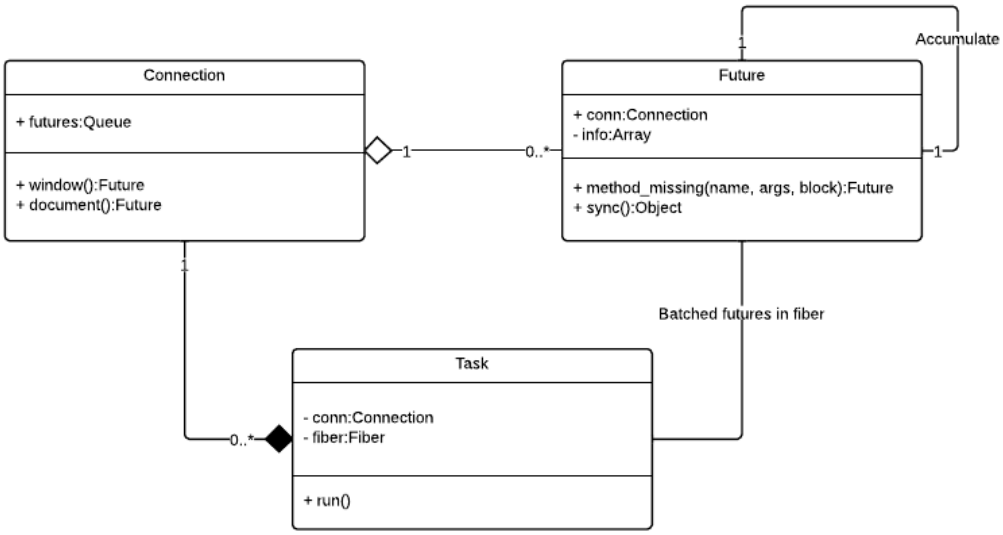


Fig. 5.5. Class diagram of batched futures

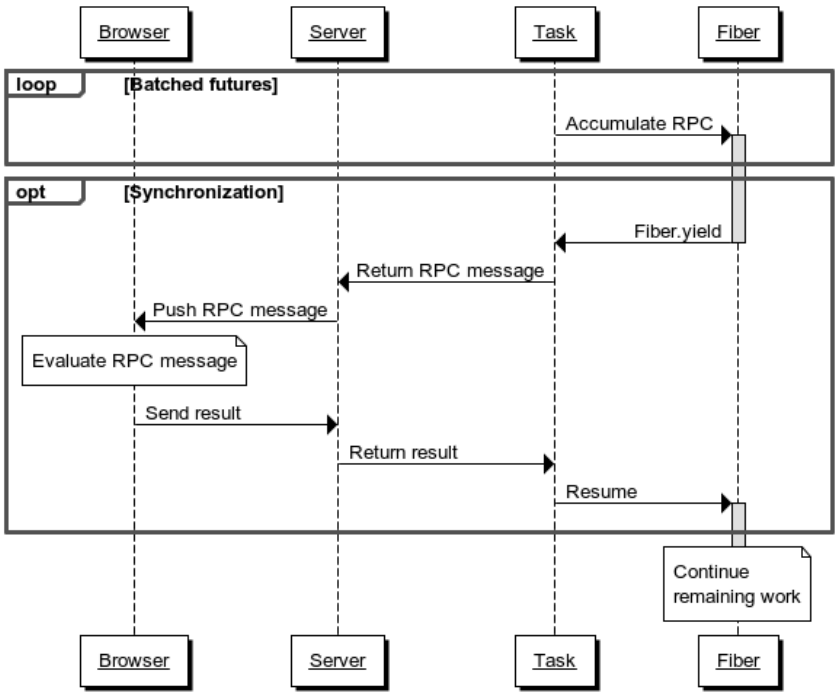


Fig. 5.6. Suspending and resuming task

method is not found, an exception will be raised. The $redomCall method is registered to all objects at browser side by set $redomCall to *Object.prototype*, in order that it will work in a method chain. For example, the following Opal-compiled JavaScript code:

JavaScript

```
self.$redomCall('$jQuery')("#text").$redomCall('$attr')("value");
```

After the execution of *$redomCall('$jQuery')("#text1")*, we will get a result object. Because we have added $redomCall to all objects, the follow-up call *$redomCall('$attr')("value")* of the result object is able to be executed correctly.

# Chapter 6

# Evaluation

This chapter presents a evaluation of Redom. We focus the evaluation on productivity and performance.

## 6.1 Productivity evaluation

In order to evaluate the productivity of Redom, the best way is to let developers to create web application using Redom and evaluate the effort they spent to create the web application. However, it is difficult for us to convene people who have ability to create web applications, to perform such experiment. Therefore, we examined another aspect in web development to evaluate the productivity of Redom, which is the code size. The effort required for creating a web application can be evaluated by measuring the code size because a smaller code size means less effort to develop and maintain the web application. We also compared Redom with other web framework, which shows the advantages in improving productivity.

### 6.1.1 Code size

We created three web applications using Redom and AJAX, which are Chat (described in section 3.3), Note (described in section 3.5.1) and PingPong (described in section 3.5.2). We choose the three web application because they contain all the operations that may occur in web application, such as data exchanging between server and browser, server/browser communication, browser/browser communication, etc. Then we compared the code size of each web application created in two ways. We used two metrics, LOC (physical source lines of code) and LLOC (logical source lines of code) to measure the size of code [36]. The result of comparison is shown in table 6.1.

| | LOC | | LLOC | |
|---|---|---|---|---|
| | AJAX | Redom | AJAX | Redom |
| Chat | 29 | 20 | 18 | 13 |
| Note | 67 | 27 | 45 | 19 |
| PingPong | 28 | 19 | 16 | 11 |

Table. 6.1. Result of code size comparison for web applications using Redom and AJAX

From the result, we can see that the code size is smaller when using Redom to create web applications. Therefore, we can conclude that it is effective to use Redom in web development for a higher productivity.

We also re-created several web applications using Redom, which are originally created

| | LOC | LLOC |
|---|---|---|
| Chat(Node.js)[37] | 111 | 67 |
| Chat(Redom) | 51 | 39 |
| Handler(GWT)[38] | 343 | 116 |
| Handler(Redom) | 22 | 13 |
| Smiple document application(CloudBrowser)[18] | 11 | 7 |
| Smiple document application(Redom) | 12 | 7 |
| Dictionary suggest(Links)[22] | 106 | 73 |
| Dictionary suggest(Redom) | 70 | 52 |

Table. 6.2. Result of code size comparison for web applications using Redom and other web frameworks

by other web frameworks. The source code of each web application created using Redom is shown in Appendix A-D. The code size comparison is shown in table 6.2.

First, compared to Chat in Node.js, there is no need to program the details of communication, such as sending message to the server when the send button is clicked, and displaying the message when received. With Redom, the program is simplified to two DOM manipulations which are retrieving user's input and display the message on other page. Second, compared to GWT, the code size is largely reduced when using Redom. The GWT framework requires programmer to create a lot of Java files to define both the web service and the behavior of browser-side manipulation, however, in Redom, there is no need to define a web service, all DOM manipulation can be done from server. Moreover, the simplicity of Ruby script language makes it further simpler to create a web application. Third, compared to web application created using CloudBrowser, the size of source code is nearly the same. CloudBrowser is a very productive web framework because the browser-side logics are compacted into HTML. However, all application logics at server side in Redom also achieves the same effect. Fourth, compared to Links, Redom shows the high productivity as well. Links allows developer to create web application in a single language for both server-side and browser-side programs, however, developers have to specify whether a function is executed at server side or browser side, which increases the complexity to create a web application.

From the result we discussed above, we can conclude that less or same code is needed to create a web application by using Redom, comparing to other existing web frameworks with high productivity. Therefore, the productivity of web development has been improved by Redom.

### 6.1.2 Comparison with other web frameworks

In this section, we compare Redom with other web frameworks (table 6.3).

From the table, we can see that we have solved all the problems outlined in section 2.2.4. We implemented Redom to be a single-language, server-centric web framework. There is no Ruby to JavaScript compilation so that it is easy to debug the source code. The HTML is separated from application logics in order to separate designing from programming. Moreover, Redom provides features like server push, browser/browser communication, debugging support, etc., which improves the productivity of web development further.

Table. 6.3. Productivity comparison of web frameworks: this table shows whether the problems outlined in section 2.2.4 are solved or not

| | Redom | Ruby on Rails | CloudBrowser | Node.js | GWT | Orca | Links | HOP |
|---|---|---|---|---|---|---|---|---|
| Single-language | Yes (Ruby) | No (Ruby,JavaScript) | Yes (JavaScript) | Yes (JavaScript) | Yes (Java) | Yes (Smalltalk) | Yes (Links) | Yes (extended Scheme) |
| Server-centric | Yes | Yes | Yes | No | Yes | Yes | Yes | Yes |
| No compiling to JavaScript | Yes[a] | Yes[b] | Yes | Yes | No | No | No | No |
| Server push | Yes | No | Yes | Yes | ? | Yes | ? | ? |
| Browser/browser communication | Yes | No | Yes | Yes | ? | Yes | ? | ? |
| Not mix application logic in HTML | Yes | No | ? | Yes | Yes | No | No | No |
| Able to use JavaScript libraries | Yes | Yes | Yes | Yes | No | ? | ? | ? |
| Debugging support | Yes | ? | ? | ? | Yes | ? | ? | ? |

[a] Developers can compile DOM manipulations to JavaScript for better performance
[b] Only when using RJS[39]

## 6.2 Performance evaluation

It is important to evaluate the performance of a web farmework because the performance is directly related to user experience. In this section, we describe the performance evaluation of Redom and show the result.

The experiment environment we used for the evaluation is shown in table 6.4.

| | Local server/client<br>Server over LAN | Client over LAN |
|---|---|---|
| Machine | MacBook Air 11-inch | ThinkPad T400s |
| OS | Mac OS X Lion 10.7.4 | Windows 7 Professional |
| CPU | 1.6 GHz Intel Core 2 Duo | Intel Core2 Duo P9400 2.40GHz |
| Memory | 4 GB 1067 MHz DDR3 | 4.00 GB |
| Browser | Safari 6.0.2 | Google Chrome 19.0 |
| Ruby | 1.9.2p290 | - |

Table. 6.4. Experiment environment

In table 6.4, the local server/client means that the browser/server communication is performed on a single machine. Server/client over LAN means that the communication is completed over LAN. Usually, the browser/server communication on a local environment is faster than that over LAN.

To compare the performance of RPC in Redom with native JavaScript code, we used the following program to perform the experiment.

JavaScript

```
function benchmark(n) {
  for (var i = 0; i < n; i++) {
    text = document.getElementById("text1").value
    document.getElementById("text2").value = text
  }
}
```

Ruby

```
def benchmark(n)
  n.times {
    text = document.getElementById("text1").value
    document.getElementById("text2").value = text
  }
end
```

The two fragments of code do the same thing, DOM manipulation that copies the content in a text box to another text box. The *benchmark* method takes an integer as argument so that we can perform the DOM manipulation several times.

We measure the execution time of the code above in following 7 cases:

- Native JavaScript code runs on browser.
- Synchronous RPC runs on a single machine.
- Batched futures runs on a single machine.

- Opal-parsed JavaScript runs on a single machine.
- Synchronous RPC runs over LAN
- Batched futures runs over LAN.
- Opal-parsed JavaScript runs over LAN.

Synchronous RPC means that every RPC triggers a server/browser communication. Batched futures means that RPC are accumulated until a synchronization occurs. Opal-parsed JavaScript is JavaScript code that is parsed from Ruby code by Opal. For each case, we passed integer from 20 to 200 with a step of 20 to method benchmark and have it executed. We repeated the execution nine times for each $n$. Then we got rid of the largest and smallest execution time, calculate the average and median of the remaining seven execution time. The execution time of each case is shown in table 6.5 and table 6.5. The header of the table shows the argument $n$ passed to method *benchmark*. The data in each cell is shown in the form *avg./med.*, and the unit of time is millisecond (ms).

| | 20 | 40 | 60 | 80 | 100 |
|---|---|---|---|---|---|
| Native JavaScript | 0/0 | 0/1 | 0/1 | 0/1 | 0/1 |
| Synchronous RPC (local) | 101/97 | 162/158 | 216/214 | 287/283 | 406/398 |
| Batched futures (local) | 14/9 | 42/41 | 53/52 | 65/66 | 58/58 |
| Opal-parsed JavaScript (local) | 4/5 | 6/7 | 7/8 | 11/9 | 8/8 |
| Synchronous RPC (LAN) | 438/389 | 1214/1245 | 1774/1829 | 2122/2031 | 2133/2136 |
| Batched futures (LAN) | 60/36 | 61/62 | 101/92 | 108/111 | 127/120 |
| Opal-parsed JavaScript (LAN) | 9/9 | 8/8 | 9/9 | 8/9 | 10/10 |

Table. 6.5. Execution time: The header is the argument $n$ passed to method *benchmark*. The data in each cell is *avg./med.*. The unit of time is *ms*.

| | 120 | 140 | 160 | 180 | 200 |
|---|---|---|---|---|---|
| Native JavaScript | 0/1 | 1/1 | 1/1 | 1/1 | 1/2 |
| Synchronous RPC (local) | 478/485 | 555/541 | 605/601 | 687/694 | 788/782 |
| Batched futures (local) | 69/69 | 88/86 | 99/96 | 122/122 | 101/99 |
| Opal-parsed JavaScript (local) | 9/9 | 10/10 | 10/11 | 19/11 | 12/12 |
| Synchronous RPC (LAN) | 2780/2632 | 3026/3031 | 3816/3475 | 3815/3782 | 4906/4946 |
| Batched futures (LAN) | 147/147 | 155/156 | 300/221 | 285/260 | 339/355 |
| Opal-parsed JavaScript (LAN) | 10/10 | 12/12 | 11/12 | 10/10 | 11/11 |

Table. 6.6. Execution time (continued): The header is the argument $n$ passed to method *benchmark*. The data in each cell is *avg./med.*. The unit of time is *ms*.

From the execution time, we can see that the averages are very close to medians when executed in local environment because the browser/server connection is relatively stable. When executed over LAN, sometimes the average is quite different from median. The reason is that the communication may be affected by the browser/server connection over relatively unstable LAN environment. Therefore, to avoid the impact brought by a very high or low value, we chose median to evaluate the performance of each case.

First, we compare the execution time of native JavaScript, synchronous RPC and batched futures. The result is shown in figure 6.1.

The figure 6.1 shows that it is very fast to perform such DOM manipulation by native JavaScript. Conversely, it is extremely slow to use synchronous RPC to access browser-side objects because of the overhead of browser/server communication. However, by introducing batched futures in Redom, the performance has been greatly improved. As we can see in the figure, using batched futures is more than ten times faster then using synchronous RPC to perform DOM manipulation. The proposal of using batched futures

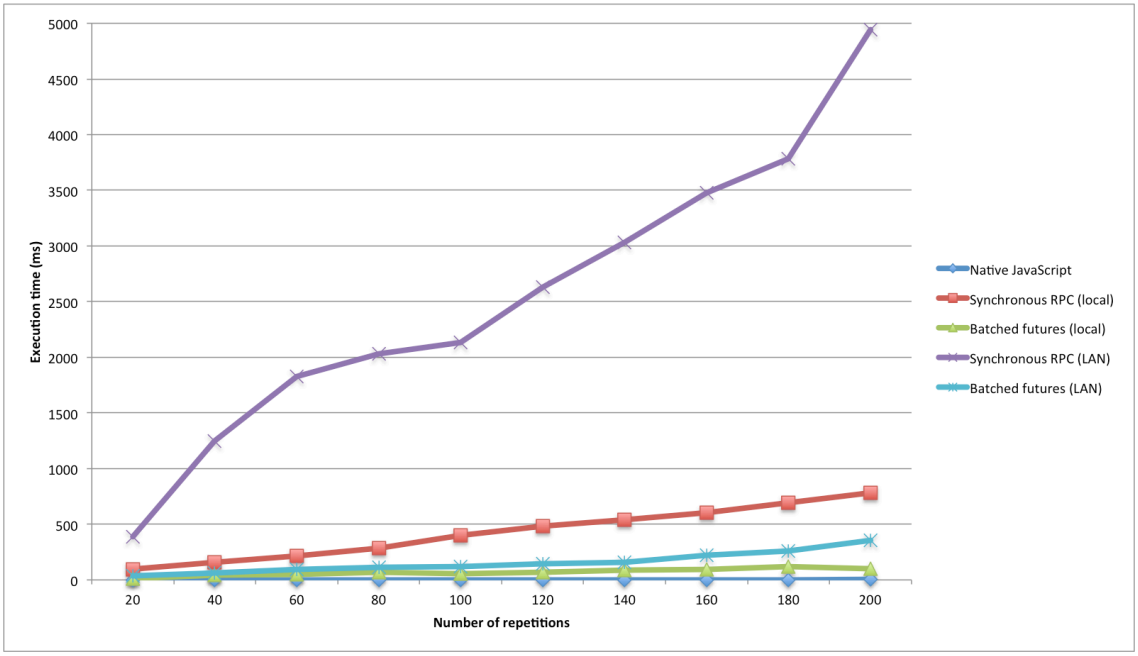


Fig. 6.1. Performance evaluation of batched futures

to improve performance of Redom is effective. However, batched futures is still much slower than native JavaScript. The reason for this is that the need for synchronizations in RPC causes browser/server communications.

Second, we compare the execution time of native JavaScript, batched futures and Opa-parsed JavaScript. The chart is shown in figure 6.2.

As we can see, the Opal-parsed JavaScript runs much faster than batched futures. With the growth of number of executions, the difference of execution time becomes larger and larger. The reason why it is much faster (about ten times) to compile Ruby to JavaScript is that once the parsed JavaScript code is sent to browser side with one time server/browser communication, there will be no subsequent communication when the code is executed. The parsed code is almost executed as efficient as native JavaScript code except a little overhead of finding methods ($redomCall).

From the result of evaluation, we conclude that it is efficient to use batched futures to improve the performance of Redom. Moreover, Ruby-to-JavaScript compilation will provide further better performance.

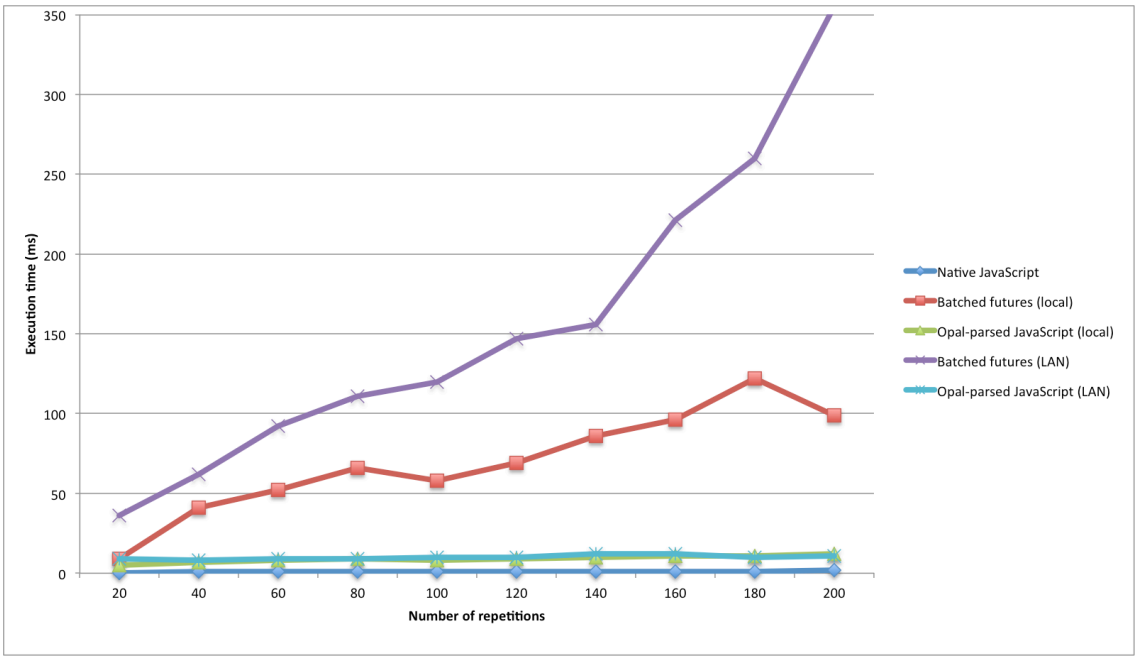


Fig. 6.2. Performance evaluation of parsing Ruby to JavaScript

# Chapter 7

# Conclusion

We designed and implemented a single-language server-centric web framework called Redom for developers to create interactive web applications easily and efficiently. To achieve the purpose of a web framework with high productivity, we used distributed object in Redom so that browser-side objects can be manipulated using simple RPC from server-side Ruby program. Further, we proposed to use batched futures and Ruby-to-JavaScript compilation in Redom to solve the performance problem. Finally, we evaluated the productivity and performance of Redom to show that Redom is able to be used to create practical web applications.

There are several problems in traditional AJAX-based web development, such as JavaScript dependency, distributed programming and so on. A lot of web frameworks have been proposed and developed to improve the productivity of web development. However, there are more or less some problems in those web frameworks, such as unable to use JavaScript library, need to learn a new programming language and so on.

Therefore, we proposed a single-language server-centric web framework Redom to solve these problems. We designed Redom to use simple RPC to access browser-side distributed objects from server-side Ruby program. Therefore, developers are able to write only server-side simple programs to create interactive web applications. We also provides useful API that supports developers to create various web applications with less effort, such as API for browser/browser communication. However, synchronous RPC in Redom causes a performance issue. To improve the performance of Redom, we proposed to introduce batched futures into Redom. By using batched futures, RPC messages are accumulated and a synchronization is performed when necessary. As a result, the server/browser communications are reduced and the performance of Redom is improved greatly. Moreover, we proposed to compile Ruby code of centralized DOM manipulation to JavaScript code to obtain better performance.

We implemented Redom according to the our design principles. We used WebSocket as the communication protocol for a full-duplex communication and used reactor pattern to implement the event-driven programming in Redom. We also implemented batched futures using Ruby fiber and Ruby-to-JavaScript compilation using Opal.

We evaluated the productivity of Redom by measuring the code size of applications created using AJAX and Redom. The result shows that developers can create web applications with less code using Redom. We compared Redom with other web framework to show that Redom solve the problems which affect the productivity in web development. We then evaluated the performance of Redom when using batched futures instead of synchronous RPC. The result shows that batched futures improves the performance of Redom by 5 to over 10 times. Another result of performance evaluation shows that by compiling centralized DOM manipulation to JavaScript, the performance is improved further at least 4 times than using batched futures, and the more DOM manipulations,

the higher performance will be obtained.

In the future, we want to use Redom to create more practical web applications. In current Redom, developers have to trigger a synchronization of batched futures explicitly in program under certain circumstances, and the code of centralized DOM manipulation must be specified by developers. Therefore, we also plan to solve these problems to make Redom further easier to use.

# Publications and Research Activities

(1) Eki Ko, Koichi Sasada. DJS: A distributed object library for Server-WebBrowser communication. 第 88 回プログラミング研究発表会. 15 March 2012.

# References


[1] Iwan Vosloo and Derrick G. Kourie. Server-centric web frameworks: An overview. *ACM Computing Surveys*, 40(2), April 2008.

[2] Peter Eeles. Distributed object patterns. *Rational Architecture Workshop*, May 2000.

[3] Mehdi Jazayeri. Some trends in web application development. In *2007 Future of Software Engineering*, May 2007.

[4] Jesse J. Garrett. Ajax: A new approach to web applications. `http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications`, 2005.

[5] DOM (Document Object Model). `http://www.w3.org/DOM/DOMTR`.

[6] Patrcia Gomes Soares. On remote procedure call. In *Proceedings of the 1992 conference of the Centre for Advanced Studies on Collaborative research*, volume 2, November 1992.

[7] Dan Kegel. The C10K problem. `http://www.kegel.com/c10k.html`.

[8] Douglas C. Schmidt. Reactor: An object behavioral pattern for concurrent event demultiplexing and event handler dispatching. *Pattern Languages of Program Design*, pages 529–545, 1995.

[9] Irfan Pyarali, Marina Spivak, Ron Cytron, and Douglas C. Schmidt. Evaluating and optimizing thread pool strategies for real-time corba. In *Proceedings of the ACM SIGPLAN workshop on Languages, compilers and tools for embedded systems*, August 2001.

[10] 郡司啓. Fiber と Proc —— 手続きを抽象化する二つの機能. *Rubyist Magazine 0034* **号**, 2011.

[11] B. Liskov and L. Shrira. Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, July 1988.

[12] ECMAscript. `http://www.ecmascript.org`.

[13] jQuery. `http://jquery.com`.

[14] Prototype. `http://prototypejs.org`.

[15] Reuven M. Lerner. At the forge: Coffeescript and jquery. *Linux Journal*, 2011(209), 2011.

[16] Opal. `http://opalrb.org`.

[17] Ruby on Rails. `http://rubyonrails.org`.

[18] Brian McDaniel and Godmar Back. The cloudbrowser web application framework. In *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*, October 2012.

[19] Node.js. `http://nodejs.org`.

[20] Google Web Toolkit (GWT). `https://developers.google.com/web-toolkit/`.

[21] Lauritz Thamsen, Anton Gulenko, Michael Perscheid, Robert Krahn, Robert Hirschfeld, and David A. Thomas. Orca: A single-language web framework for collaborative development. In *Proceedings of the 2012 10th International Conference on Creating, Connecting and Collaborating through Computing*, January 2012.

[22] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. In *Proceedings of the 5th international conference on Formal*

*methods for components and objects*, 2006.
[23] Manuel Serrano and Grard Berry. Multitier programming in hop. In *Communications of the ACM*, volume 55, August 2012.
[24] B. H. Tay and A. L. Ananda. A survey of remote procedure calls. In *SIGOPS Operating Systems Review*, volume 24, July 1990.
[25] A. L. Ananda, B. H. Tay, and E. K. Koh. A survey of asynchronous remote procedure calls. In *SIGOPS Operating Systems Review*, volume 26, April 1992.
[26] Roger S. Chin and Samuel T. Chanson. Distributed, object-based programming systems. In *Computing Surveys (CSUR)*, volume 23, March 1991.
[27] 関 将俊. *dRuby* **による分散・** *Web* **プログラミング**. オーム社, 2005.
[28] Ruby. `http://www.ruby-lang.org`.
[29] Websocket Rack. `https://github.com/imanel/websocket-rack#readme`.
[30] Google Inc. WebRTC. `http://www.webrtc.org`.
[31] Phillip Bogle and Barbara Liskov. Reducing cross domain call overhead using batched futures. In *Proceedings of the ninth annual conference on Object-oriented programming systems, language, and applications*, October 1994.
[32] Phil McCarthy and Dave Crane. *Comet and Reverse Ajax: The Next-Generation Ajax 2.0*. APress, 2008.
[33] Saeed Aghaee and Cesare Pautasso. Mashup development with HTML5. In *Proceedings of the 3rd and 4th International Workshop on Web APIs and Services Mashups*, December 2010.
[34] Can i use... Support tables for HTML5, CSS3, etc. `http://caniuse.com`, 2012.
[35] Ilya Grigorik. Ruby & websockets: Tcp for the browser. `http://www.igvita.com/2009/12/22/ruby-websockets-tcp-for-the-browser/`, December 2009.
[36] Source lines of code. `http://en.wikipedia.org/wiki/Source_lines_of_code`.
[37] 【初心者向け】node.js(0.8) + socket.io(0.9x) のサンプルプログラム. `http://d.hatena.ne.jp/replication/20121222/1356154137`, 2012.
[38] Google web toolkit examples, demos and source. `http://code.google.com/p/gwt-examples/`.
[39] RJS. `http://en.wikipedia.org/wiki/List_of_Ajax_frameworks#Ruby`.

# Acknowledgements

I would like to express my heartfelt gratitude to all people who helped me with my research and this thesis.

My deepest gratitude goes foremost to my current supervisor Shigeru Chiba and former supervisor Koichi Sasada. Without their enthusiastic support and advice to my research, I was not able to accomplish what I have done so far. Especially thanks to Koichi Sasada for contributing the origin idea of my research and guiding me along with the development of this project. He also spent his precious time reviewing this thesis and making a lot of useful comments. I gratefully acknowledge all the support he provided to me. I appreciate Ruby committers, especially Yugui and Nahi, for paying attention to my research and their precious advice. Thanks Google Japan for providing me an opportunity to introduce my research. High tribute shall be paid to all members in Chiba research group and former Sasada laboratory for the useful discussion and feedbacks to support me in my research and thesis. Special thanks to *www.lucidchart.com*, *www.websequencediagrams.com* and *creately.com* for the excellent services they provided. The beautiful diagrams in this thesis are created using their service. Last my thanks would go to my beloved family for their loving considerations and great confidence in me all through these years.

# Appendix A

# Source code of Chat using Redom

We ported the Chat application[37] created with Node.js to the following Redom application to compare the source code size (Section 6.1.1).

```
class ChatConnection
  include Redom::Connection

  def send_message(message)
    connections.each { |conn|
      conn.document.getElementById("receiveMsg").innerHTML = message
      conn.sync{}
    }
  end

  def on_open
    console.log("connet");
    document.getElementById("connectId").innerHTML = "ID::#{self.__id__}"
    document.getElementById("type").innerHTML = "METHOD::WebSocket"
    document.getElementById("send").onclick { |e|
      send_message document.getElementById("message").value.sync
    }
    document.getElementById("disconnect").onclick { |e|
      send_message "#{self.__id__} is disconnected."
    }
  end
end
```

# Appendix B

# Source code of Handler using Redom

We ported the Handler application[38] created with GWT to the following Redom application to compare the source code size (Section 6.1.1).

```
class HandlerConnection
  include Redom::Connection

  COLOR = ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9",
"A", "B", "C", "D", "E", "F"]

  def random_color
    color = "#"
    6.times {
      color << COLOR[rand(COLOR.length)]
    }
    color
  end

  def on_open
    jQuery("#left")[0].onclick { |e|
      jQuery("#rightCell").attr("style", "background-color:#{random_color}")
    }
    jQuery("#right")[0].onclick { |e|
      jQuery("#leftCell").attr("style", "background-color:#{random_color}")
    }
  end
end
```

# Appendix C

# Source code of Simple Document application using Redom

We ported the Simple Document application[18] created with CloudBrowser to the following Redom application to compare the source code size (Section 6.1.1).

```
class SDConnection
  include Redom::Connection

  def on_open
    jQuery("#fname")[0].onchange { |e|
      jQuery("#output").text(jQuery("#fname").attr("value").sync +
" " + jQuery("#lname").attr("value").sync)
    }
    jQuery("#lname")[0].onchange { |e|
      jQuery("#output").text(jQuery("#fname").attr("value").sync +
" " + jQuery("#lname").attr("value").sync)
    }
  end
end
```

# Appendix D

# Source code of Dictionary Suggest using Redom

We ported the Dictionary Suggest application[22] created with Links to the following Redom application to compare the source code size (Section 6.1.1).

```
class Dictionary
  CH = ('a'..'z').to_a

  def initialize
    @dic = Hash.new
    100.times {
      word = CH.shuffle[0..(rand(10) + 1)].join
      @dic[word] = word * 5
    }
  end

  def find(word)
    @dic.select { |k, v|
      k =~ /^#{word}/
    }
  end

  def update(word, meaning)
    @dic[word] = meaning
  end

  def delete(word)
    @dic.delete word
  end

  def [](word)
    @dic[word]
  end
end

class DictionaryConnection
  include Redom::Connection
```

```
  def redraw
    word = jQuery("#input").attr("value").sync
    if word.empty?
      jQuery("#definitions").html("")
    else
      words = @dic.find(word)
      definitions = ""
      words.each { |k, v|
        definitions << "<div id=\"#{k}\"><a href='#'>#{k}: #{v}</a></div>
<div id=\"edit_#{k}\"></div><br>"
      }
      jQuery("#definitions").html(definitions).sync
      words.each { |k, v|
        jQuery("##{k}")[0].onclick { |e|
          edit = %Q{
<table>
  <tr>
    <td>Word:</td>
    <td>#{k}</td>
</tr>
<tr>
  <td>Meaning:</td>
  <td><textarea id="meaning_#{k}">#{@dic[k]}</textarea></td>
</tr>
<tr>
  <td>
    <input type="button" id="update_#{k}" value="Update">
    <input type="button" id="cancel_#{k}" value="Cancel">
  </td>
  <td><input type="button" id="delete_#{k}" value="Delete"></td>
  </tr>
</table>
}
          jQuery("#edit_#{k}").html(edit).sync
          jQuery("#update_#{k}")[0].onclick { |e|
            meaning = jQuery("#meaning_#{k}").attr("value").sync
            @dic.update(k, meaning)
            jQuery("##{k}").html("<a href=\"#\">#{k}: #{meaning}</a>")
            jQuery("#edit_#{k}").html("")
          }
          jQuery("#cancel_#{k}")[0].onclick { |e|
            jQuery("#edit_#{k}").html("")
          }
          jQuery("#delete_#{k}")[0].onclick { |e|
            @dic.delete k
            jQuery("##{k}").remove
            jQuery("#edit_#{k}").html("")
          }
        }
      }
    end
```

```
  end

  def on_open
    @dic = Dictionary.new

    jQuery("#input")[0].onkeydown { |e|
      redraw
    }

    jQuery("#add")[0].onclick { |e|
      word = jQuery("#add_word").attr("value")
      meaning = jQuery("#add_meaning").attr("value")
      sync{}
      @dic.update(word, meaning)
      jQuery("#add_word").attr("value", "")
      jQuery("#add_meaning").attr("value", "")
      redraw
    }
  end
end
```