

# The CloudBrowser Web Application Framework

Brian Newsom McDaniel

Thesis submitted to the Faculty of the  
Virginia Polytechnic Institute and State University  
in partial fulfillment of the requirements for the degree of

Master of Science  
in  
Computer Science and Applications

Godmar V. Back, Chair  
Stephen H. Edwards  
Calvin J. Ribbens

May 2nd, 2012  
Blacksburg, Virginia

Keywords: web frameworks, server-centric, JavaScript

Copyright 2012, Brian McDaniel

# The CloudBrowser Web Application Framework

Brian Newsom McDaniel

(ABSTRACT)

While more and more applications are moving from the desktop to the web, users still expect their applications to behave like they did on the desktop. Specifically, users expect that user interface state is preserved across visits, and that the state of the interface truly reflects the state of the underlying data. Unfortunately, achieving this ideal is difficult for web application developers due to the distributed nature of the web. Modern AJAX applications rely on asynchronous network programming to synchronize the client-side user interface with server-side data. Furthermore, since the HTTP protocol is stateless, preserving interface state across visits requires a significant amount of manual work on behalf of the developer.

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 is ideal for single-page web applications, which is the current trend in web development. 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. Furthermore, multiple clients can connect to a single server-side interface instance, providing built-in co-browsing support. Unlike other server-centric frameworks, CloudBrowser's exclusive use of the HTML document model and associated JavaScript execution environment allows it to exploit existing client-side user interface libraries and toolkits while transparently providing access to other application tiers. We have implemented a prototype of CloudBrowser as well as several example applications to demonstrate the benefits of its server-centric design.

# Acknowledgments

I would like to thank my advisor, Dr. Godmar Back, for his guidance and support throughout the development of this thesis. Few professors devote as much one-on-one time to their students as Dr. Back, and I've grown immensely as a researcher and an engineer through our weekly meetings. I sincerely appreciate him giving me the freedom to develop my own ideas and make my own mistakes, while still ensuring that I ultimately traveled down the right path.

I would also like to thank Dr. Calvin Ribbens for serving on my committee, and for offering guidance as I transitioned from an undergraduate to a graduate student. It's obvious that Dr. Ribbens cares deeply about the students at Virginia Tech, and his approachability and kindness ensured I always felt comfortable seeking help while attempting to navigate through endless graduate school requirements.

I am also grateful to Dr. Stephen Edwards for serving on my committee, and for his Programming Languages class. His class gave me a deeper understanding of programming language theory, and the lessons I learned there made it much easier to learn the new languages and technologies required for this thesis. I took Programming Languages during my first semester as a graduate student, and it was a great introduction to the demands and rewards of graduate education.

Finally, I would like to thank my classmates Naren Sundaravaradan, Brian Nicholson, Kyle Morgan, and Michael Woods, without whom I could not have survived graduate school.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Motivation . . . . .	1
1.2	Core Contributions . . . . .	3
<b>2</b>	<b>Background</b>	<b>5</b>
2.1	Web Development Concepts . . . . .	5
2.1.1	The Document Object Model . . . . .	5
2.1.2	Traditional AJAX Applications . . . . .	8
2.1.3	Event-Based Programming . . . . .	10
2.2	Web Development Tools . . . . .	15
2.2.1	Node . . . . .	15
2.2.2	CoffeeScript . . . . .	16
2.2.3	Client-side Libraries . . . . .	19
2.2.4	Server-Centric Web Frameworks . . . . .	23
<b>3</b>	<b>Developing CloudBrowser Applications</b>	<b>26</b>

3.1	The CloudBrowser Paradigm . . . . .	26
3.2	Framework Architecture . . . . .	28
3.3	Application Life Cycle . . . . .	30
3.3.1	Initializing an Application Instance . . . . .	30
3.3.2	Managing Virtual Browsers . . . . .	31
3.4	Components . . . . .	31
<b>4</b>	<b>Example Applications</b>	<b>33</b>
4.1	Simple Document Example . . . . .	33
4.2	Model-View-Controller Chat Room Example . . . . .	36
4.3	Database-backed Phone Book Application . . . . .	40
4.4	Co-browsing Based Scheduling Application . . . . .	43
<b>5</b>	<b>Implementation</b>	<b>49</b>
5.1	System Components . . . . .	49
5.1.1	HTTP Server . . . . .	49
5.1.2	Browser Manager . . . . .	50
5.1.3	Shared Components . . . . .	51
5.1.4	Client Engine . . . . .	51
5.1.5	Server Engine . . . . .	52
5.2	Key Implementation Aspects . . . . .	53
5.2.1	Bootstrapping the Client . . . . .	53

5.2.2	Synchronization Protocol . . . . .	54
5.2.3	Client Event Processing . . . . .	56
5.2.4	JavaScript Execution . . . . .	57
5.3	Component-based Widgets . . . . .	59
5.4	Multiprocess Architecture . . . . .	61
<b>6</b>	<b>Evaluation</b>	<b>63</b>
6.1	Memory Usage . . . . .	63
6.2	Latency . . . . .	64
6.3	Bandwidth Consumption . . . . .	67
6.4	DOM Conformance . . . . .	69
<b>7</b>	<b>Related Work</b>	<b>71</b>
7.1	ZK . . . . .	71
7.2	ItsNat . . . . .	73
7.3	Fiz . . . . .	74
7.4	Ripley . . . . .	74
7.5	Crawljax . . . . .	75
7.6	Opera Mini . . . . .	75
<b>8</b>	<b>Conclusion</b>	<b>76</b>
8.1	Future Work . . . . .	76
8.2	Summary . . . . .	77



# List of Figures

2.1	A DOM Tree . . . . .	6
2.2	Knockout Simple Example Screen Shot . . . . .	23
3.1	CloudBrowser System Architecture . . . . .	29
4.1	Simple Document Example Screen Shot . . . . .	34
4.2	Date Application Screen Shot . . . . .	35
4.3	Chat Room Application Screen Shot . . . . .	37
4.4	Chat Room Application Architecture Diagram . . . . .	38
4.5	Phone Book Application Screen Shot . . . . .	42
4.6	Meeting Time Application Screen Shot . . . . .	44
5.1	YUI3 Date Picker Component Screen Shot . . . . .	59
5.2	Load Balancing CloudBrowser Applications . . . . .	62
6.1	Average Latency without Simulated WAN Delay . . . . .	66
6.2	Average Latency with Simulated Client Mental Processing Delay . . . . .	67



# List of Tables

5.1	Client Engine RPC Methods . . . . .	52
5.2	Server Engine RPC Methods . . . . .	53
5.3	DOM Node Record Format . . . . .	54
6.1	Sizes of Static Bootstrap Files . . . . .	68
6.2	Serialized DOM Snapshot Sizes vs. HTML Text Size . . . . .	68
6.3	jQuery Test Suite Performance . . . . .	69

# Chapter 1

## Introduction

### 1.1 Motivation

The web has become the preferred medium for distributing applications. Web applications can be accessed from any device with a compatible browser, can be updated transparently and instantly by the developer, and allow application state to be offloaded “to the cloud”, where it is backed up and always accessible. As applications move from the desktop to the web, users expect the same, rich experience they are used to with their desktop applications. Existing web application frameworks attempt to enable developers to provide this desktop-like experience for their users, but these frameworks provide a poor experience for developers, and the resulting applications still fall short of user expectations.

The key issue with developing web applications is that they are distributed systems. Part of the application runs in a user’s web browser, while the other part runs on a separate server computer. Furthermore, the underlying HTTP protocol upon which web applications are built is stateless. AJAX [27] is a technique that allows developers to create interactive web applications using client-side JavaScript programming. Developers using AJAX are fully exposed to the distributed nature of the web, and must manually program the asynchronous

network communication between a client's browser and the application server. Application logic must be split between the client and the server, with each part usually written in a different programming language. Multiple representations of model data must be maintained: one representation in the client's browser, and one or more representations, depending on caching and redundancy, on the server computer(s). Developers must manually synchronize the client-side representation with the authoritative, server-side representation, which is usually kept in a database.

Server-centric frameworks [18, 38] move all application code to the server. These frameworks handle the browser/server communication on behalf of the developer, and therefore shield developers from the distributed nature of the web. Since the application code is running exclusively on the server, developers do not need to manually synchronize multiple representations of the same data. Unfortunately, however, developers using existing server-centric frameworks are forced to use server-side proxy objects to manipulate the user interface. These server-side proxy objects must be translated to HTML and CSS for display on the client, a process which is error-prone and out of the hands of the application developer. Since the application developer can only indirectly program the client-side view, errors in styling and layout become difficult to debug [15].

These existing framework implementations force developers to choose between having full control of the user interface or being fully exposed to the distributed nature of web applications. In addition to an unsatisfactory developer experience, existing web application frameworks, both AJAX and server-centric, make it difficult to remember client-side user interface state across requests. The client-side state inside a user's web browser, including the variables in the JavaScript engine and the browser-maintained user interface data structures, is transient. Whenever a user visits another page or clicks the refresh button, the client-side environment is re-initialized from scratch. As users become accustomed to having their data stored in the cloud, they expect that changes made to an application's user interface are persistent - a clear conflict with the design of web browsers. On subsequent visits to an application, users expect to be able to pick up where they left off. Delivering this

experience requires a large effort from the application developer, and it is often only done in a coarse-grained fashion, if at all, with no aid from the framework. To implement user interface persistence, a developer must manually store hints on the server about the state of each user interface element.

## 1.2 Core Contributions

The issues caused by existing web application frameworks are not intrinsic in web application development, but are instead the result of design decisions made by framework developers. We have created CloudBrowser to address the concerns listed in the previous section, providing a better experience for both developers and users. CloudBrowser maintains an application's complete user interface state server-side, as a document in a headless, virtual browser. CloudBrowser applications employ a thin-client architecture, whereby clients connecting to application instances are treated as remote displays for the server-side virtual browser. Virtual browsers may persist past the lifetime of a particular client's connection to it. In this way, user interface state is automatically preserved on the server, since users can disconnect and reconnect to their application instances.

This thin-client architecture also provides a natural co-browsing ability, since it can support simultaneous display to multiple clients. Co-browsing can be used to create real-time, collaborative applications in which the entire user interface is shared among all connected clients. As one client changes the interface, that change is immediately and automatically propagated to all other clients. This architecture also amortizes the memory allocation cost of a virtual browser among all connected clients.

CloudBrowser applications are written exclusively in HTML, CSS, and JavaScript, which are the same languages used for client-side development in web applications. The virtual browser provides an environment that mimics that of a desktop browser, allowing developers to use existing client-side libraries to develop CloudBrowser applications. This allows developers

and designers to leverage their existing skill sets. Furthermore, application interfaces can be developed and tested locally in a browser, without the need to configure a CloudBrowser server.

CloudBrowser does not perform any rendering or layout computation on the server. Performing such rendering would impose extra memory and processing overhead, and we have found that it is not necessary for the development of rich, interactive applications. This design decision means that CloudBrowser does not support application code that relies on access to pixel locations of user interface components. Such access is typically used to implement animations or effects. With the upcoming CSS3 standard, however, much of the functionality that previously required access to computed layout can be implemented using CSS, including animations and effects.

Similar to existing server-centric frameworks, model data and application code both reside exclusively in the server-side environment. This allows developers to place model data directly into the user interface, which is embodied in the virtual browser. Our underlying implementation platform, Node (also known as Node.js), provides thousands of libraries for accessing other application tiers, including databases and third-party APIs. Node is a JavaScript runtime, which means that CloudBrowser allows JavaScript code running in the context of a virtual browser to directly access other application tiers.

We have developed a prototype of the CloudBrowser framework, along with several example applications, including some that use sophisticated JavaScript libraries. Our experience has shown that CloudBrowser greatly simplifies the development of web applications and that the resulting applications more closely match user expectations. We have also completed an evaluation of our framework, measuring the bandwidth, latency, and memory costs. Additionally, we have performed tests to measure how well our virtual browser environment matches that of a desktop browser. We have found that the latency and memory overhead introduced by CloudBrowser is acceptable for many classes of applications.

# Chapter 2

## Background

We assume the reader is familiar with basic HTML, CSS, and JavaScript concepts. This section will give background information on topics central to CloudBrowser where more than a basic understanding of web technologies is required.

### 2.1 Web Development Concepts

This section explores general concepts in web development that form the foundation of modern web applications.

#### 2.1.1 The Document Object Model

The structure of a web page exists in two forms: the HTML text in a .html file, and the in-memory representation once the HTML is parsed by a web browser. The in-memory representation is constructed according to the Document Object Model [29] (DOM) specification. The DOM is a W3C [11] specification that defines objects and methods for representing and manipulating the in-memory representation of an HTML page. In a web browser, the DOM specifies the programming interface through which JavaScript running on a page can

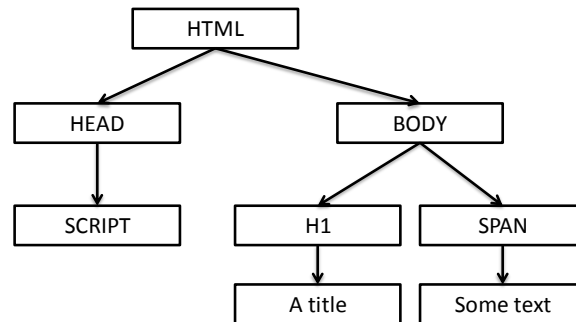


Figure 2.1: A DOM Tree

manipulate a web application’s user interface.

## The DOM Tree

Well-formed HTML documents form a natural hierarchy, and therefore the DOM is represented using a tree structure. The DOM tree is made available to JavaScript code running in the browser, giving developers a way to manipulate it, possibly in response to user events (such as clicks) or network events (such as AJAX requests), while the page is being displayed.

As an example, consider the following HTML code:

```
1 <html>
2   <head>
3     <script src="library.js"></script>
4   </head>
5   <body>
6     <h1>A title</h1>
7     <span>Some text</span>
8   </body>
9 </html>
```

When processed by a web browser, the DOM tree shown in Figure 2.1 is constructed. JavaScript on a page may manipulate the DOM tree, resulting in changes to the user interface rendered in the client’s browser.

## DOM Events

A key part of the DOM specification is the DOM Event [40] specification. Similar to other graphical user interface programming environments, the DOM defines an event system that can invoke callback functions in response to event triggers. For example, a developer may want to display to the user the number of clicks on a particular DOM element. The following code accomplishes this:

```
1 <html>
2   <head>
3     <script>
4       document.addEventListener('DOMContentLoaded', function () {
5         var numClicks = 0;
6         var clickTarget = document.getElementById('clickTarget');
7         var clickTally = document.getElementById('clickTally');
8         clickTarget.addEventListener('click', function () {
9           clickTally.innerHTML = ++numClicks;
10        });
11      });
12    </script>
13  </head>
14  <body>
15    <div id='clickTarget'>
16      I've been clicked <span id='clickTally'>0</span> times.
17    </div>
18  </body>
19 </html>
```

We register 2 event handlers in this example. The `document` object represents the HTML-Document element of the DOM tree, which is the container in which a DOM tree lives. Since HTML is parsed in the order in which it is presented in the raw HTML file, we must wait until all of the HTML has been parsed, and therefore the entire DOM tree constructed, before we manipulate the DOM using JavaScript. To do this, we wait for the `document` to



fire the `DOMContentLoaded` event, which means that the DOM tree structure has been fully constructed. Had we not waited for this event, we would not have been able to find the `clickTarget` or `clickTally` elements, since they would not have been created at the time the script element was run.

The second event listener we add listens for the click event on a `div` element. Whenever the user clicks this element, the registered callback function is invoked. In our handler, we rewrite the contents of the `clickTally` span to indicate the current number of clicks.

There are 3 distinct phases in the event dispatch process: the capturing phase, the at-target phase, and the bubbling phase. When a DOM event is generated and dispatched, it is not dispatched directly to the target element. Instead, the event is first dispatched on each element along the path from the root of the DOM tree to the target element, during what is called the capturing phase of the event. The next phase is the at-target phase, in which the event is dispatched on the actual event target (for example, the `clickTarget` div in the above example). Finally, during the bubbling phase, the event “bubbles” back up the DOM tree, being dispatched on ancestor nodes in the reverse order of the capturing phase.

When registering an event handler, programmers can specify whether the handler should be invoked in either the capturing phase or the bubbling and at target phases. By intercepting events during the capturing phase, it is possible to halt the event propagation and prevent the event from actually reaching its intended target. We exploit this capability to implement the CloudBrowser client library (Section 5.2.3).

### 2.1.2 Traditional AJAX Applications

When a user navigates to a new web page, the client-side browsing environment, such as variables in the JavaScript engine and the DOM tree, is wiped clean. If a user subsequently returns to the application, the application code must manually recreate the client-side state using hints stored on the server, in cookies, or in local storage on the browser.

Asynchronous JavaScript and XML (AJAX) [27] provides a mechanism for receiving data from a web server without reloading the entire page. JavaScript code on a page uses the built-in `XMLHttpRequest` object to make AJAX requests, and places the resulting data into the client-side DOM tree.

As an example, consider the following JavaScript code.

```
1 var xhr = new XMLHttpRequest();
2 xhr.onreadystatechange = function () {
3     if (xhr.readyState == 4) {
4         var target = document.getElementById('target');
5         target.innerHTML = xhr.responseText;
6     }
7 };
8 xhr.open('GET', 'info.html', true);
9 xhr.send();
```

This code makes an AJAX request to a web server, requesting `info.html`. As the request progresses, the `onreadystatechange` callback is invoked. When the response has been fully received, the `XMLHttpRequest`'s `readyState` property is set to 4, which means the callback function can access the returned response. In this instance, the server returns HTML, which is placed directly into the DOM. In practice, the `responseText` could be JSON, XML, or any other string.

While they provide a richer user experience, AJAX applications are more complex to develop compared to non-AJAX web applications. Each AJAX request requires a server-side entry point into the application. On the server, developers must differentiate between AJAX requests, whose responses might return data in the form of JavaScript Object Notation (JSON) [19] or XML [16], and regular page requests, which return a complete HTML page. While the underlying `XMLHttpRequest` object that powers AJAX can be used synchronously, in practice, it is almost always used asynchronously to improve performance. This means that AJAX responses may arrive at unpredictable times. AJAX applications also force

application logic to be distributed between the client and server. Model data must also be distributed, with one representation maintained on the client for display in its view, and the authoritative representation residing on the server. In short, AJAX completely exposes the web application developer to the distributed nature of web applications.

AJAX also exacerbates the issue of remembering client-side state, since a given view could have mutated drastically from its initial rendering. Instead of each view change requiring a complete page refresh, and therefore accessing a unique URL on the server, web applications can be built as long-running JavaScript programs executing in the browser, making AJAX calls for server-side data and mutating the view accordingly. This means that the state in a client's web browser cannot be deduced from the URL of the initial request, and additional bookkeeping must occur on the server to restore the user interface state.

Another issue with traditional AJAX applications is that of data duplication. Since the client-side view engine is decoupled from the server-side application logic, multiple representations of application data must be created and maintained. For example, a row in a database may be used to create an object in a server-side application instance, which may then be serialized as JSON and sent to the client where it is reconstructed as an object instance in the client's JavaScript engine. As the user interacts with the application, changes to the client-side object instance may occur, and those changes must then be propagated back to the server and saved to the database. Alternatively, the server-side data may change independently of the client, requiring an update to be sent to and handled by the client. Even under optimal network conditions, ensuring data consistency adds significant overhead to AJAX application development. This task is made even more difficult when disruptions in network connectivity occur.

### **2.1.3 Event-Based Programming**

Event-based programming is a programming style for managing concurrent, asynchronous tasks, which are typically I/O operations. Event-based programs are run by a central event

loop, which processes events as they occur. For example, an event may be the completion of an I/O operation or a timer. Programmers organize their code into discrete events, scheduling when the events should run on the event loop. Event-based systems are typically single-threaded from the application developer's perspective, so event code itself should never block the CPU while waiting for I/O operations. Instead, I/O is performed by the framework outside of the event loop, allowing developers to register an event to run upon its completion.

Due to their single-threaded design, event-based systems eliminate many of the race condition errors that arise in multi-threaded programs with shared memory access. Event-based systems are well suited for applications that spend much of their time waiting for I/O. Both web servers and user interfaces are good candidates for event-based programming. For server applications, event-based systems typically use less memory than equivalent multi-threaded applications, since event-based systems only need to maintain a single execution stack.

JavaScript itself is a single-threaded language, and was primarily designed for user interface programming. This makes it an ideal candidate for an event-based design. As shown in Section 2.1.1, developers programming in a browser register event listeners to be invoked in response to user events, such as clicks. Section 2.1.2 showed that AJAX requests work similarly. The browser maintains an event loop that invokes these event handlers when the events they listen for fire.

The main downside of event-based programming is that the required programming style can lead to code that is difficult to understand, maintain, and debug [20]. Cases have been made both for [33] and against [49] event-based programming. The root cause of the debugging difficulty is the problem of “stack ripping [13].” When a function that performs an asynchronous operation is used, it requires its callee to be split into two functions. The first function includes the code before and including the asynchronous call, and second function includes the code that operates on the data prepared by the asynchronous call. The second function must be passed as a parameter to the function with the asynchronous call, so that it can be invoked once the asynchronous operation has completed. This phenomenon is called

stack ripping because the variables that would, in a thread-based approach, be stored on a single activation record, must instead be stored on two activation records: one for the first function, and one for the second. One can think of the execution stack as having been ripped in half at the point of the asynchronous function call.

As an example, consider the following JavaScript code:

```
1 function doWork () {
2     var value = 3;
3     var returnValue = doSomethingSynchronous(value);
4     return returnValue + 1;
5 }
```

If `doSomethingSynchronous` in the above function is made asynchronous, the stack must be ripped in half at the point of invoking the asynchronous function. This is achieved by splitting the application logic and program state into two functions instead of having only one. The following code shows the asynchronous version where the stack has been ripped:

```
1 function doWork (callback) {
2     var value = 3;
3     doSomethingAsynchronous(value, function (returnValue) {
4         callback(returnValue + 1);
5     });
6 }
```

Not only did the internals of `doWork` need to be split across two functions, but also its function signature needed to be changed. This highlights another problem with stack ripping: when a synchronous function call is replaced with an asynchronous function call, the stack ripping effect ripples up through all callees of the changed function. In this example, `doWork` must take a callback in order to return the proper value, and therefore any function calling `doWork` must rip its stack around the call to `doWork`.

One benefit of using JavaScript for asynchronous programming is its built-in support for closures and anonymous nested functions. While the stack ripping effect does force the

introduction of new functions, all of the logic can still be lexically contained inside of a single outer function. Many languages, such as C or C++, do not allow functions to be declared within other functions. This makes code harder to read when program logic must be split across multiple functions. Furthermore, languages without support for closures must manually bundle program state and pass it between functions.

When languages do not support closures or anonymous functions, asynchronous programming can be made easier with the use of libraries and preprocessor enhancements. Tame [33] is a runtime library and source-to-source translator that allows programmers to program asynchronous code in a synchronous style. Behind the scenes, it adds support for closures and multiple function entry-points, allowing computations to resume where they left off. Tame has been ported to JavaScript, and released as TameJS [10]. Even though JavaScript has some language features that make asynchronous programming easier, there is still a need for a library like TameJS. Asynchronous JavaScript programming tends to induce many levels of nested indentation, since each asynchronous operation must be continued inside of another function. TameJS allows developers to program in a synchronous style while retaining the asynchronous underpinnings. This leads to code that is much easier to refactor, and also easier to read.

While TameJS introduces a translation step, there are also a number of user-level libraries designed to make asynchronous control flow more manageable. One such library is Async.js [34]. This library provides a number of helper functions for asynchronously manipulating collections and for writing readable asynchronous code. As an example, consider the problem of invoking three asynchronous functions in parallel, and only processing the results once all functions have returned. The following JavaScript code accomplishes this:

```
1 function runInParallel (callback) {
2     var finished = 0;
3     var results = [];
4     function checkDone (err, result) {
5         if (err) {
```

```
6         return callback(err, null);
7     }
8     results.push(result);
9     if (++finished == 3) {
10         callback(results);
11     }
12 }
13 asyncFunction1(checkDone);
14 asyncFunction2(checkDone);
15 asyncFunction3(checkDone);
16 }
```

This example requires a significant amount of boilerplate code to collect the results and invoke the callback on completion. Furthermore, if the number of asynchronous calls changes, then the developer must remember to change the terminating condition inside of the `checkDone` function. Async.js allows a developer to rewrite the code as follows:

```
1 function runInParallel (callback) {
2     async.parallel([
3         asyncFunction1,
4         asyncFunction2,
5         asyncFunction3
6     ], callback);
7 }
```

As long as the asynchronous functions follow the usual convention of invoking the callback with the first argument designating a possible error condition, Async.js can automatically collect results and pass them to a final callback once all asynchronous tasks have completed. Async.js also supports a number of other control flows, such as serial execution or repeated execution until some condition is met.

## 2.2 Web Development Tools

This section investigates tools that are used for developing modern web applications. Many of these tools have also been used in the construction of CloudBrowser.

### 2.2.1 Node

CloudBrowser has been implemented on the Node platform (commonly referred to as Node.js). Node is a JavaScript run-time system built on top of Google's V8 [2] JavaScript engine. Node provides asynchronous, non-blocking, event-based I/O libraries for tasks such as file system manipulation, socket communication, and serving or requesting data over HTTP. The JavaScript language is conducive to asynchronous event-based programming due to its first-class functions, closures, and single threaded design.

As an example, consider the following code for reading a file from disk and printing the results.

```
1 var fs = require('fs');
2
3 fs.readFile('hello.txt', 'utf8', function (err, data) {
4     if (err) throw err;
5     console.log(data);
6 });
```

This code instructs Node to read a file from disk, and then supplies a callback to be invoked once the file's data is available. It is important to note that the process does not block while waiting for the file to be read. Since Node is single-threaded, blocking while waiting for I/O operations would block the entire event loop, drastically degrading performance. Instead, the Node runtime will invoke the supplied callback function once the data has been read from the file. Since all I/O is done asynchronously, developers must deal with the stack ripping issue, either manually or by employing helper libraries.



Node provides a module system for packaging and organizing JavaScript code. The module system is based on the CommonJS Modules specification [1]. As shown in the above example, modules can be included by passing the module name (for built-in modules) or the module path (for third party modules) to the `require` method. Each JavaScript file in a Node project is wrapped in a closure function before being invoked. The closure function is passed the following arguments: `exports`, `module`, `require`, `__filename`, and `__dirname`. To create a custom module, one uses the `exports` and `module` variables. The `exports` variable points to the JavaScript object that will be returned by a `require` call for this module. To export functionality from a module, one can either attach properties to the `exports` object, or reassign `module.exports` to a new JavaScript object.

Here is an example of creating a custom module that exposes a `hello` function.

```
1 // mymodule.js
2 exports.hello = function () {
3     console.log('hello');
4 };
```

Another file can now require `mymodule.js`, and use its `hello` method.

```
1 var mymodule = require('/path/to/mymodule');
2 mymodule.hello();
```

The module system can be used to organize code within a single project or to package code that can be shared across projects. To facilitate the second use case, Node comes bundled with `npm` [45], the Node Package Manager, which provides both a centralized module repository and a command line tool for installing and managing modules.

## 2.2.2 CoffeeScript

CoffeeScript [14] is a scripting language that compiles into JavaScript. CoffeeScript allows programmers to express JavaScript programs more succinctly. CoffeeScript provides built-in support for classes, unlike JavaScript. Programming in an object-oriented style in pure

JavaScript requires an in-depth knowledge of JavaScript's prototypal inheritance mechanism. CoffeeScript also provides automatic function-level lexical variable scoping, preventing a major source of hard-to-find bugs in JavaScript programs. In JavaScript, unless a variable declaration is prefixed with the `var` keyword, it is treated as a global variable. A common mistake is to forget the `var` keyword, unintentionally introducing new global variables. CoffeeScript removes the `var` keyword and scopes all variables to the function level.

Many of CoffeeScript's improvements simply cut down on the number of lines of code required to express a program. For example, instead of delineating code blocks with curly braces (`{` and `}`), indentation is used. Eliminating curly braces saves a line of code for each level of nesting, since closing braces are usually placed on their own line. CoffeeScript also supports comprehensions, which provide a concise syntax for iterating over the elements in an array or object.

As an example, consider the following JavaScript code:

```
1 var i;
2 for (i = 0; i < myArray.length; i++) {
3     var elem = myArray[i];
4     doSomething(elem);
5 }
```

Using comprehensions, we can express the same loop more concisely:

```
1 doSomething(elem) for elem in myArray
```

CoffeeScript is effectively shorthand JavaScript. Originally, CloudBrowser was implemented using pure JavaScript. Part way through the development, it was converted to CoffeeScript, which resulted in smaller source files, better code organization, and faster development.

As an example of the built-in class support of CoffeeScript, consider the following JavaScript program:

```
1 var fs = require('fs');
2 function Person (name) {
```

```

3   this.name = name;
4   this.info = null;
5 }
6 Person.prototype.readFromFile = function () {
7   var self = this;
8   fs.readFile(name + '.txt', 'utf8', function (err, data) {
9     if (err) throw err;
10    self.info = JSON.parse(data);
11  });
12 };

```

This program uses JavaScript's prototypal inheritance mechanism to implement a `Person` class. When a function is invoked with the `new` keyword, it is treated as a constructor for a new object. In this example, the `Person` constructor sets two properties on the new object. To dereference property lookups on an object, the object itself is checked first for the property, and if it is not found, then the object's prototype object is checked. If the prototype object does not have the property, then *its* prototype is checked, and so on. This list of prototypes is called the prototype chain.

When an object is constructed from a function, its prototype is set to the prototype of that function. In this example, the `Person` prototype has one property: the `readFromFile` function. All instances of `Person` will have access to `readFromFile`, since it is found in their prototype chain. To override this method, objects can add a `readFromFile` method on themselves, which will be found before the `readFromFile` on their prototype. By manipulating the prototype chain, one can implement inheritance.

CoffeeScript's built-in class support removes the need to manipulate object prototypes. The following code expresses the example program more succinctly using CoffeeScript:

```

1 fs = require('fs')
2 class Person
3   constructor : (@name) ->
4     @info = null

```

```
5  readFromFile : () ->
6    fs.readFile "#{name}.txt", 'utf8', (err, data) =>
7      throw err if err
8    @info = JSON.parse(data)
```

In addition to the previously cited benefits of CoffeeScript, this example demonstrates some extra features. The `@` symbol is shorthand for `this.`, which allows more succinct property access. When used in a constructor parameter list, those variables prefixed with `@` will automatically be set as properties of the object. The `=>` operator binds a function to the current `this` value, helping to prevent a common source of errors in JavaScript programming. If a function is not invoked with a particular receiver object using the `.` notation, then the `this` value is set to the global object. The `=>` syntax allows developers to avoid workarounds for this behavior, such as saving a reference to the receiver object in a closure, as seen in the pure JavaScript implementation with the `self` variable. Finally, one can see that parentheses are optional when invoking a method that takes arguments. This saves lines, since we do not need to take up an extra line closing the call to `fs.readFile`. As more and more callbacks are nested, which is typical in Node code, saving the trail of closing braces is a significant boost to code density.

### 2.2.3 Client-side Libraries

As JavaScript has gained acceptance among developers, there has been a large investment in the JavaScript library ecosystem. These libraries make it easier for developers to perform common front-end tasks, such as manipulating the DOM tree or making AJAX requests. CloudBrowser allows developers to leverage these existing client-side libraries (and their expertise with using them). While our goal is to allow any client-side library to run unmodified in CloudBrowser, we have found two libraries especially useful: jQuery and Knockout. Since these libraries are used in examples throughout this thesis, we will give an overview of their use here.

## jQuery

jQuery [42] provides a succinct, cross-browser interface for accessing and manipulating the DOM tree of a browser. The library is accessed through the `$` function, which is placed in the global scope. This object can be used for locating DOM elements using CSS selectors, attaching event handlers, and making AJAX requests, to name just a few features.

As an example, consider the task of adding a `click` event handler to every entry in the unordered list in the following HTML document:

```

1 <html>
2   <body>
3     <ul id='list'>
4       <li>A</li>
5       <li>B</li>
6       <li>C</li>
7     </ul>
8   </body>
9 </html>

```

The following code accomplishes our goal using the raw W3C DOM API:

```

1 var ul = document.getElementById('list');
2 var i;
3 for (i = 0; i < ul.childNodes.length; i++) {
4   var li = ul.childNodes.item(i);
5   li.addEventListener('click', function (event) {
6     var text = event.target.textContent;
7     console.log(text + ' was clicked!');
8   });
9 }

```

We can write this much simpler using jQuery:

```

1 $('#list li').click(function () {
2   var text = $(this).text();

```

```
3 console.log(text + ' was clicked!');  
4 });
```

Not only is the jQuery implementation shorter and easier to understand, it is also compatible with all major browsers. The raw DOM implementation uses `addEventListener` and `textContent`, which are not implemented in most versions of Internet Explorer.

## Knockout

The model-view-controller (MVC) [17] pattern is well suited for developing web applications. In MVC, application concerns are split into three main areas. The model represents the back-end business domain objects, which are typically persisted to a database. The controller is responsible for fetching the model objects from the back-end store, and passing them on to the view. The view is in turn responsible for displaying these model objects to the user. The view is usually implemented using a templating facility whereby model data is substituted in place of placeholder variables in the HTML code.

Knockout [43] is a client-side JavaScript library that enables developers to use data bindings. Data bindings are used to map, or *bind*, model data to view elements and their attributes, effectively playing the role of the controller. Knockout uses the observer and observable patterns [26] to automatically update the view whenever the model data changes. Likewise, Knockout can also bind model data to attributes of mutable fields in the view, such as `input` elements, such that whenever the user changes the field in the view, the model data will automatically be updated.

Being designed for the client portion of traditional AJAX applications, Knockout uses the term Model-View-View Model (MVVM) to express the assumption that the client-side UI (“View”) is synchronized with a collection of JavaScript observables (“View Model”), which is separately synchronized with the actual model that is kept server-side. This structure is a result of the distributed nature of web applications, since the view runs in a separate process

from the server. When running Knockout in CloudBrowser on the server, however, the view model and model can become one and the same, eliminating the need for programming any client-server interaction. This makes it very easy to write complex interactive applications without synchronizing distributed representations of the same data.

Knockout bindings are expressed using HTML5 `data-` attributes. Knockout detects attributes of the form `data-bind`, and uses the attribute value to specify which data is bound in which ways. Knockout requires a view model object that contains the data to be bound, and it automatically resolves variable references in `data-bind` statements as properties of the view model. To enable automatic change detection and propagation, the properties of the view model should be of type `observable` or `observableArray`, which are part of the Knockout library.

As an example, consider the following HTML and JavaScript code:

```
1 <html>
2   <head>
3     <script src="knockout.js"></script>
4   </head>
5   <body>
6     <select multiple="multiple" data-bind="options: people"></select>
7     <script>
8       var viewModel = {
9         people : ko.observableArray(['Joe', 'Sally', 'Fred'])
10      };
11      ko.applyBindings(viewModel);
12    </script>
13  </body>
14 </html>
```

This code binds the `observableArray` `people` to the options of the `select` element. Figure 2.2 shows the resulting view. If we add to or remove elements from `viewModel.people`, the view will automatically re-render itself to reflect the changed view model. Knockout supports

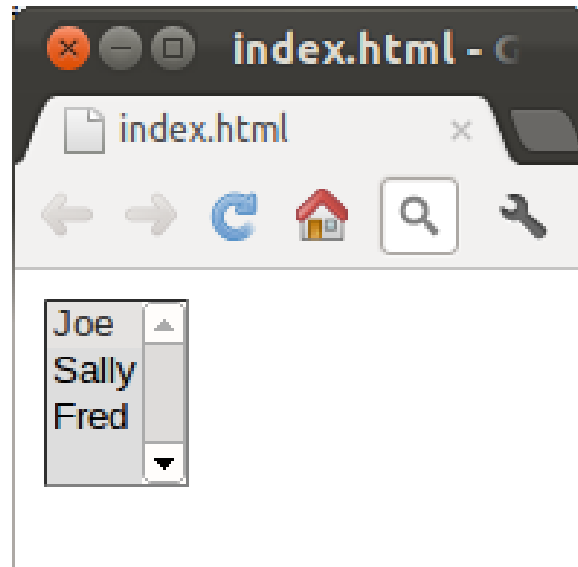


Figure 2.2: Knockout Simple Example Screen Shot

many other types of data-bindings, such as `text`, `value`, `style`, and more.

## 2.2.4 Server-Centric Web Frameworks

Server-centric web frameworks attempt to alleviate some of the issues with traditional AJAX applications. These frameworks move application state to the server, allowing developers to manipulate the application state in one place and have the changes automatically reflected on the client. Typically, server-centric frameworks employ server-side components that embody the user interface state of the client. These components are programmed in a high-level language, such as Java. To construct the client-side representation, the components are translated to corresponding HTML and CSS. Server-centric frameworks typically employ a client-side JavaScript library, called the client engine, which is responsible for synchronizing the server-side components with their client-side rendering. The client engine uses a synchronization protocol that operates over AJAX requests or WebSockets.

In a pure server-centric framework, all events are intercepted on the client and sent to the server for processing. These events are translated into corresponding component events.



Application code running on the server can register event handlers on components, which are invoked in response to these client-side events. The event handler code can modify the state of the components, so the server-centric framework must be able to detect these changes. The changes must be sent to the client engine in the browser so it can synchronize the client-side rendering with the new state of the server-side component.

Server-centric frameworks hide all of the client/server communication from the developer. The developer is able to program the server-side components using a traditional, single-process desktop style, while the framework handles the translation and synchronization processes. Unfortunately, existing server-centric framework implementations introduce new problems for application developers.

Typical server-centric frameworks add their own markup elements and components, instead of being built on pure HTML and CSS. This requires a translation step from higher-level components to HTML and CSS. As an example of the difficulties with this approach, consider the use of the CSS `width` attribute in the ZK framework [15]. ZK adds an `hbox` element to aid with grid-based layouts. The following code attempts to create an `hbox` with two `div` elements, each using 50% of the containing `hbox`:

```
1 <hbox width="100%">
2   <div width="50%"> Left </div>
3   <div width="50%"> Right </div>
4 </hbox>
```

Unfortunately, the `hbox` element is translated into several intermediate elements, so the styling does not behave as expected.

The above ZK code translates to the following HTML:

```
1 <table width="100%">
2   <tr>
3     <td>
4       <div style="width:50%"> Left </div>
5     </td>
```

```
6     <td>
7         <div style="width:50%"> Right </div>
8     </td>
9 </tr>
10 </table>
```

Instead of each `div` taking up 50% of the `hbox`, each `div` only takes up 25%. This is because the `hbox` is translated into a table, and the `div` elements are children of new, invisible (to the developer) `td` elements. The CSS `width` property is relative to the target's containing element, so the `div` elements actually use 50% of their containing `td` elements, which in turn are already using 50% of the containing `tr`.

This is one example of issues that arise during the translation process. Another concern is that developers who have already mastered HTML, CSS, and JavaScript cannot effectively use their existing skill sets. Even expert web developers must learn a new markup language and component set to be able to create applications with ZK. Server-centric frameworks also suffer from the issue of remembering client-side interface state, since components are instantiated on each request. This means it still requires manual effort from the developer to reconstruct client-side state across page refresh and navigation.

# Chapter 3

## Developing CloudBrowser Applications

This chapter describes the CloudBrowser paradigm for developing applications, the framework architecture and underlying implementation platform, the life cycle of a CloudBrowser application, and the built-in support for higher-level components built on web standards.

### 3.1 The CloudBrowser Paradigm

The CloudBrowser paradigm is based on the following key ideas:

1. Persistent Server-side View State

An issue facing web developers is the separation of server-side application state and the client-side view state. CloudBrowser moves the view state to the server, giving the server-side application direct, in-process access to it. This results in a desktop-like programming environment that persists across requests. Server-side view state is accomplished by maintaining a server-side document in a headless, virtual browser, and then mirroring that document to the client. Client-side generated events, such as

mouse clicks, are captured and sent to the server-side document for processing. The application developer can therefore treat the server-side document as the actual view, allowing them direct, non-distributed access to the application's view state.

## 2. Direct Access to Back-end Resources

Along with direct access to view state, CloudBrowser also provides direct access to back-end resources, such as databases, the file system, or other application tiers. The data from these resources can be placed directly into the virtual browser's DOM, causing a corresponding change in all connected clients' views. This means that event handlers registered on elements in the view can easily access other application tiers.

## 3. Communication Details Hidden from Developer

Since client/server communication arises from the need to communicate between two separate processes, a model that mimics a single-process development environment should not expose these details to the application developer. As the developer manipulates the server-side view state, these changes are automatically mirrored to all connected clients. Since a client is effectively a "dumb" thin-client, the developer does not have any need to explicitly control it.

## 4. Ability to Leverage Existing Client-Side JavaScript Libraries

There are a large number of useful JavaScript libraries for manipulating client-side application and view state. jQuery [42] and Knockout.js [43] are two such libraries that drastically increase programmer productivity. CloudBrowser developers leverage these libraries to manipulate the application and view state, and thus may capitalize on their existing knowledge and skill sets.

Model data can be stored in three places in a CloudBrowser application, depending on application requirements. If the entire user interface and application state for a particular application instance can be shared among all users, the model data can be kept directly in the virtual browser as variables in the virtual browser's JavaScript engine. Users can

access the application by co-browsing a single virtual browser, and any user's changes will be automatically replicated to all other users. If only part of the model data is to be shared among users, this data can still be stored as JavaScript variables, but in the CloudBrowser server's Node context, independent of any particular virtual browser. Application instances can use references to this shared data. The third method for storing model data is to use a traditional database or other application tier. This approach is useful if the amount of data is too large to be kept inside of a single Node process.

Using JavaScript variables for model data allows a developer to maintain only a single copy of that data. Since application code has direct access to both the authoritative model data and the user interface, the actual model data can be placed directly into the user interface, eliminating the need to manually synchronize front- and back-end data representations. This also allows developers to leverage JavaScript libraries designed for tying model data to DOM nodes, such as Knockout.js, which we have found to drastically reduce the amount of code required to express complex user interfaces.

## 3.2 Framework Architecture

For our prototype implementation, we chose the open source Node [22] JavaScript execution environment detailed in Section 2.2.1, for the following reasons. First, Node is based on Google's V8 JavaScript engine, which represents the current state of the art with respect to execution performance. Second, Node's developer community provides many packages we use, such as the JSDOM DOM implementation [30] or the Sequelize [24] object-relational mapping library, and many others which provide an environment that facilitates access to other application tiers. Third, Node provides a single-threaded environment whose semantics matches exactly the familiar execution semantics found in today's browser. Fourth, Node's event-based design provides for fast and efficient I/O, although it requires the programmer to rearrange (i.e., stack-rip [13]) their application to handle all I/O completion

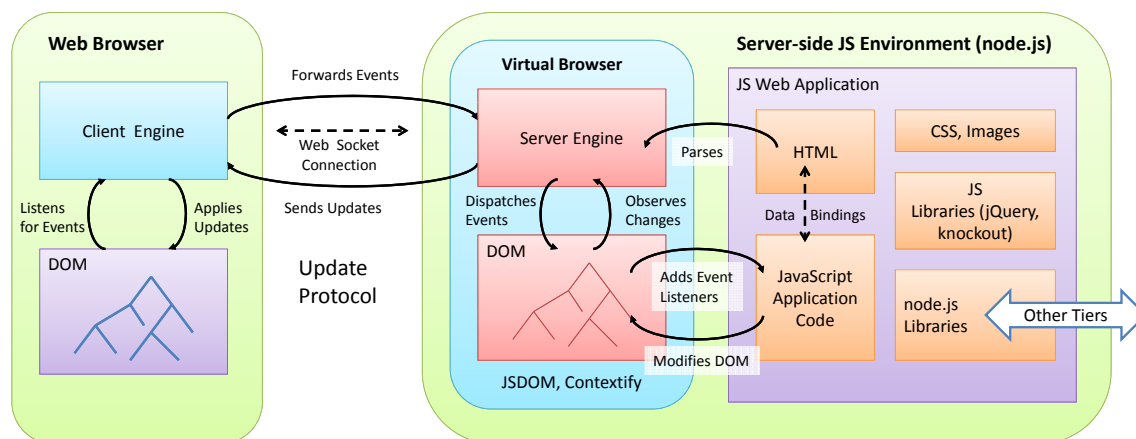


Figure 3.1: CloudBrowser System Architecture

in asynchronously invoked callbacks. Fifth, writing the server-side of our framework in JavaScript means that we can share framework code between the Client Engine (which runs in a browser) and the Server Engine (which runs in Node on the server). There are many instances in our implementation where code is shared verbatim, ensuring consistency and reducing the amount of code that needs to be written.

Figure 3.1 shows an overview of CloudBrowser’s design and components. The left half of the figure shows the Client Engine, which is responsible for intercepting client-side events and sending them to the corresponding virtual browser on the server. The virtual browser, shown in the right half of the figure, is loaded with the requested application’s code, which can access other application tiers using either built-in or third-party Node libraries. At the same time, the application code can use client-side libraries to facilitate the manipulation of the DOM tree residing in the virtual browser. As the DOM tree changes, the Server Engine detects those changes, and sends corresponding update instructions to the Client Engine, which in turn uses them to mirror the virtual browser’s DOM tree onto the client browser. The implementation of this architecture is discussed in Chapter 5.

## 3.3 Application Life Cycle

A traditional web application does not keep persistent, in-memory data structures on the server for each client. Typically, client-specific data is stored in a database, indexed by a session id. The session id is stored in a cookie, which is sent to the server on each request. The server uses the session id to reconstruct the necessary client state to handle the current request, then “forgets” the information for the next request. CloudBrowser applications work differently. On the first request from a client, the server creates a virtual browser, which is a persistent, in-memory data structure representing the view for that client. The requested application is loaded into the client’s server-side virtual browser, and the client’s actual browser is kept in sync with the virtual browser by the CloudBrowser framework. The virtual browser lifetime is decoupled from the request lifetime, and even from the lifetime of the client.

### 3.3.1 Initializing an Application Instance

If a client sends an initial HTTP request to a configured application entry point (such as `http://domain.com/example`), the CloudBrowser server starts a new virtual browser instance, creates an entry point for it (such as `http://domain.com/browsers/2e90b4b3/-index.html`), and redirects the user’s browser to that entry point. This browser URL is valid for the lifetime of the virtual browser instance. It may be shared among multiple users wishing to interact with the same application. The virtual browser loads the JavaScript, HTML, and CSS for the requested application. Much of the data that would traditionally be stored in a database as session data can be stored directly in the virtual browser’s JavaScript environment.

When a virtual browser accepts a new client, it sends a small amount of HTML along with JavaScript code to bootstrap the client. We call the JavaScript library sent to the client the Client Engine. After being initialized, the Client Engine requests the current state of the

server-side virtual browser’s DOM, which is retrieved and sent by the virtual browser’s Server Engine. Unlike in a traditional AJAX application, refreshing or navigating to the browser URL does not discard and re-initialize the virtual browser’s document. Since no state is kept in the client’s browser, clients can disconnect and reconnect while the application instance’s complete state, both user interface and application, is preserved in the context of the virtual browser document.

### 3.3.2 Managing Virtual Browsers

Virtual browsers pose a resource management problem. Since the virtual browser lifetime is decoupled from that of a client’s connection to it, it is impossible for the CloudBrowser server to know when to garbage collect a virtual browser without additional information from the user or an administrator. Our current implementation provides an administrative module to list, inspect and terminate instances. When used in conjunction with user authentication, it allows the implementation of such policies as limiting each authenticated user to at-most-one browser instance for a configured application so that the inadvertent creation of multiple, separate instances is prevented. Alternatively, users may maintain and select from multiple instances. Conversely, it is also possible to implement single-instance applications in which there is at most one virtual browser instance shared by all users, or groups of users.

## 3.4 Components

Unlike remote display systems such as Opera Mini [6] or SkyFire [8], CloudBrowser does not layout or render HTML documents server side. Browsers which do not have a graphical output and which do not compute layout or rendering information are called “headless” browsers. We deemed it too expensive to compute the layout inside the Server Engine, and we also discarded the idea to send computed layout information from the client to the server for the simple reason that it is difficult to predict which properties the code might



access. Moreover, when supporting multiple, simultaneous visitors to the same CloudBrowser instance, we cannot assume that they use identical screen sizes.

As a result, JavaScript code that accesses layout properties that are computed post layout, such as `offsetWidth`, will not work. Some JavaScript libraries use such information to position elements based on the actual size of content or the size of the viewport. Often, though not always, the desired effect of such computations can be expressed using style rules, particularly when targeting browsers that support the newer CSS3 standard. We note that CloudBrowser does support the manipulation of CSS styles via JavaScript, e.g., setting `elem.style.display = 'block'` or `$.addClass` works as expected.

We have observed a trend in the web design community moving away from directly accessing layout properties and instead relying on CSS properties wherever possible. For instance, the Bootstrap CSS library used by Twitter.com and other major websites minimizes the use of JavaScript in favor of pure CSS. Many cases where JavaScript accesses and manipulates position information can instead be expressed via CSS's fixed and absolute positioning. JavaScript-based animations can be replaced with CSS3 transitions; in fact, modern JavaScript libraries default to their use when it is available.

For applications whose user interface cannot be expressed using CSS style rules, we provide a mechanism to embed client-side components in a CloudBrowser application. This mechanism is based on the observation that well-designed user interface libraries, such as the Yahoo! User Interface Library, Version 3 (YUI-3) [12], are typically designed to coexist with other JavaScript code in the same page, and that they provide their functionality encapsulated in components that can be instantiated as JavaScript objects. The implementation of components is discussed in Section 5.3.

# Chapter 4

## Example Applications

To illustrate how CloudBrowser applications are different than traditional AJAX or client-centric web applications, we will show several example CloudBrowser applications.

### 4.1 Simple Document Example

The following code shows a simple, complete CloudBrowser application:

```
1 <html>
2   <body style="font-family: Arial">
3     First name: <input id="fname" type="text" /><br />
4     Last name: <input id="lname" type="text" /><br />
5     <br />
6     Hello <span id="output"></span>!<br />
7     <br />
8     <script>
9       var fname = document.getElementById("fname"),
10         lname = document.getElementById("lname"),
11         output = document.getElementById("output");
12
13     function onChange () {
```



Figure 4.1: Simple Document Example Screen Shot

```

14     output.innerHTML = fname.value + ' ' + lname.value;
15   }
16
17   fname.addEventListener("change", onChange);
18   lname.addEventListener("change", onChange);
19   </script>
20 </body>
21 </html>

```

The resulting interface is shown in Figure 4.1. This example uses HTML and CSS for layout, and JavaScript for behavior. When a user changes the contents of the input box, the `output` span is automatically updated. It's important to note that this application runs on the server. As the user changes the text from their web browser, the CloudBrowser client engine forwards the change event to the server, which in turn updates the server-side document. Changes to the server-side document are detected and sent back to the client. A client could switch to a different computer and connect back to this virtual browser instance, and the user interface would be exactly as they left it.

Today, most JavaScript developers use higher-level libraries such as jQuery [42], rather than using the DOM API directly. The following code implements the same simple CloudBrowser application more succinctly by using the appropriate jQuery selectors and event bindings.

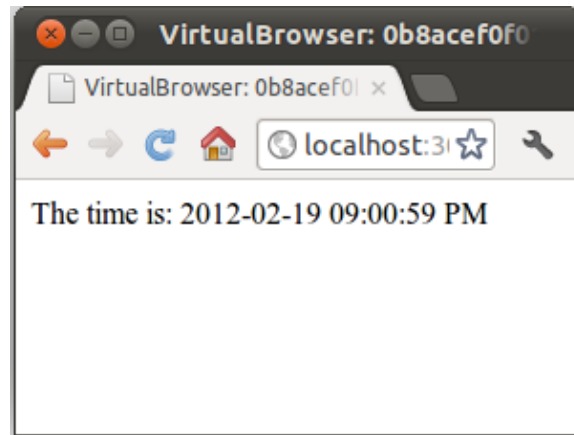


Figure 4.2: Date Application Screen Shot

```

1 <html>
2   <head>
3     <script src="jquery-1.6.1.js"></script>
4   </head>
5   <body style="font-family: Arial">
6     First name: <input id="fname" type="text" /><br />
7     Last name: <input id="lname" type="text" /><br />
8     <br />
9     Hello <span id="output"></span>!
10    <script>
11      $("#fname,#lname").change(function () {
12        $("#output").text($("#fname").val() + ' ' + $("#lname").val());
13      });
14    </script>
15  </body>
16 </html>

```

This demonstrates how CloudBrowser allows developers to leverage existing third-party libraries without modifications.

The code for this example has the same behavior when running locally, by opening the `.html` file in a browser, and when running on a remote server as a CloudBrowser application.

An advantage of using CloudBrowser is that it allows access to back-end resources via the Node module system, which is something that is not possible in a locally running JavaScript application. Node includes a `child_process` module, which allows a Node program to spawn and manage other arbitrary programs. We can use this library directly in our event handler code.

The following CloudBrowser application spawns the UNIX `date` program every second, placing its output directly into the server-side document. As the server-side document changes, the updates are reflected to all connected clients.

```
1 <html>
2   <body>
3     The time is: <span id="thetime"></span>
4     <script>
5       var exec = require("child_process").exec;
6       setInterval(function () {
7         exec("date '+%F %r'", function (e, out) {
8           document.getElementById("thetime").innerHTML = out;
9         });
10      }, 1000);
11    </script>
12  </body>
13 </html>
```

## 4.2 Model-View-Controller Chat Room Example

The previous examples combined application logic, application data, and presentation all in the same file. This approach leads to unmaintainable software as applications grow in size. While CloudBrowser does not enforce any specific program style, we will show how one can develop a CloudBrowser application using the model-view-controller (MVC) [17] pattern.

Most MVC based web application frameworks use a templating facility to express the view

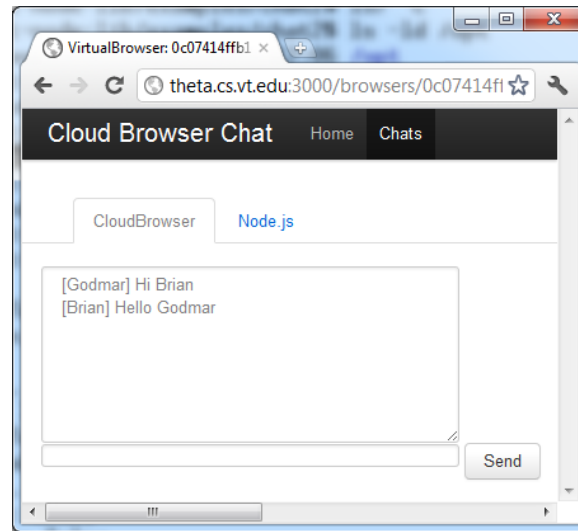


Figure 4.3: Chat Room Application Screen Shot

part of the application, and some kind of expression language to bind views to controller logic that draws from and updates an underlying model. CloudBrowser facilitates the MVC paradigm by using existing JavaScript libraries originally designed for client-side use. For example, we can use Knockout.js [43], discussed in Section 2.2.3, which provides an automatic way of synchronizing model data with its UI representation.

In traditional applications, an incoming HTTP request causes data to be fetched and passed to a templating system. The templating system substitutes the data values into a static HTML file, which is then returned to the client. If the server-side data changes, there is no automatic way of updating the client-side rendering - clients must request updates using AJAX, or additional server-side logic must be added to push data back using WebSockets or long polling techniques. With server-side data bindings, we can bind a server-side model object directly to an HTML element, and as the object mutates, the corresponding HTML element is automatically updated. Conversely, if the HTML element changes, such as the value in a text box, the model object is automatically updated.

As an example, consider a simple chat room application in which users can create new chat rooms and join existing ones. Figure 4.3 shows a screenshot of this application, while

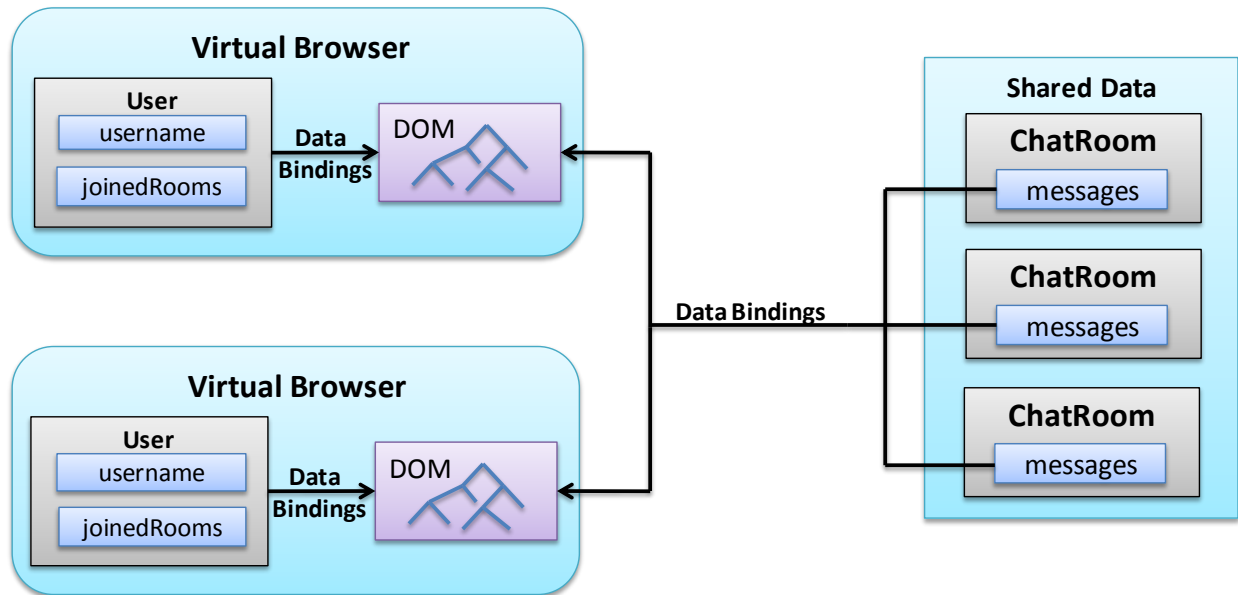


Figure 4.4: Chat Room Application Architecture Diagram

Figure 4.4 shows the application architecture.

As clients connect, virtual browsers are created for them, and each virtual browser allocates its own private data and maintains a reference to data that is shared among many virtual browsers. For the chat application, the chat room model objects are shared among all virtual browsers that have joined that room. This architecture uses the second approach to storing model data mentioned in Section 3.1: storing shared model data as JavaScript variables outside of any particular virtual browser, and then sharing references to that data among virtual browsers. Our application also utilizes local state inside each virtual browser to keep track of the current user.

The user interface for a chat room is described using Knockout data bindings using the following code:

```

1 <div id="chat-tabs" data-bind="foreach: joinedRooms">
2   <div class="tab-pane" data-bind="visible: ($root.activeRoom() === $data)
   ">
3     <textarea rows="20" data-bind="value: messages">
4     </textarea>

```

```

5   </div>
6 </div>
7 <input type="text" size="160" data-bind="value: currentMessage" />
8 <button data-bind="click: postMessage">Send</button>

```

The data bindings are expressed using expressions that occur in `data-bind` attributes. For instance, `data-bind='value: messages'` inside the `<textarea>` element ties the `textarea`'s content to the array of chat messages. A unique aspect of CloudBrowser is that both global application data and local virtual browser data can be bound to a given virtual browser's DOM elements. For example, in the chat view, `joinedRooms`, an `observableArray` of rooms that current user has joined, is local to the virtual browser, while the messages for a particular room are stored in a globally shared `observableArray`.

Knockout requires each page to create a view model object to which data bindings refer. The following code shows an excerpt from the view model for the chat page in our application:

```

1 var ko = require("knockout");
2 var user = CloudBrowser.local.user;
3 var viewModel = {
4   joinedRooms    : user.joinedRooms,
5   activeRoom     : user.activeRoom,
6   currentMessage : ko.observable(''),
7   ...
8 };
9 ko.applyBindings(viewModel);

```

The view model, combined with the Knockout library, acts as the controller implementation. All shown properties of the `viewModel` variable are observables; thus, as the number of available chat rooms changes, new tabs are automatically added (due to the `foreach: joinedRooms` data binding). Within each chat room, as the messages in a chat room change, the `value` property of all `textareas` observing this array changes. Developers can also add page-local state into a view model, and then bind that state to DOM elements. An example of this is the `currentMessage` property of the view model, which is used to keep track of



the contents of the chat input box while the user crafts their message.

In this example, the model state is kept directly in JavaScript objects, which can be shared among virtual browsers, and which can be shared among and bound to multiple views. For instance, consider the code for the ChatRoom class:

```
1 function ChatRoom (name) {
2   this.messages = ko.observableArray();
3 }
4
5 ChatRoom.prototype = {
6   postMessage : function (username, message) {
7     var str = "[" + username + " ] " + message;
8     this.messages.push(str);
9   }
10 };
```

A ChatRoom instance maintains its set of messages as an `observableArray`, and is shared among all virtual browsers whose users have joined that room. When a message is appended to the ChatRoom's array of messages, it will thus automatically be reflected in the views of all virtual browsers that include this chat room.

This example demonstrates the power of being able to leverage existing client-side libraries, such as Knockout.js. It also shows the benefits of being able to store model data as JavaScript variables, and being able to access those variables directly from virtual browsers.

### 4.3 Database-backed Phone Book Application

Whereas the preceding chat room example used in-memory JavaScript variables to represent its model state, a more typical scenario is the use of a persistent store such as a relational database. We provide a simple example of a phone book application whose entries are backed by a database. We use the Sequelize [24] Object Relational Mapping package to map

JavaScript objects to a MySQL [3] database.

The following code shows the HTML template that lists phone book entries and renders them in a table using Knockout's `text` data bindings:

```

1 <table>
2   <thead>
3     <tr>
4       <th>First Name</th>
5       <th>Last Name</th>
6       <th>Phone Number</th>
7     </tr>
8   </thead>
9   <tbody data-bind="foreach: entries">
10    <tr data-bind="click: $parent.rowClick">
11      <td data-bind="text: fname" /></td>
12      <td data-bind="text: lname" /></td>
13      <td data-bind="text: phoneNumber" /></td>
14    </tr>
15  </tbody>
16 </table>

```

The phone book entry objects whose properties (e.g., `fname`, `lname`) are referred to in these bindings are constructed directly from the database. If the user clicks on a row, additional input fields to edit this entry are displayed.

The following code displays the editing box:

```

1 <div data-bind="with: currentEntry">
2   <br /><br />
3   First Name: <input data-bind="value: fname" />
4   <br />
5   Last Name: <input data-bind="value: lname" />
6   <br />
7   Phone Number: <input data-bind="value: phoneNumber" />
8   <br />

```

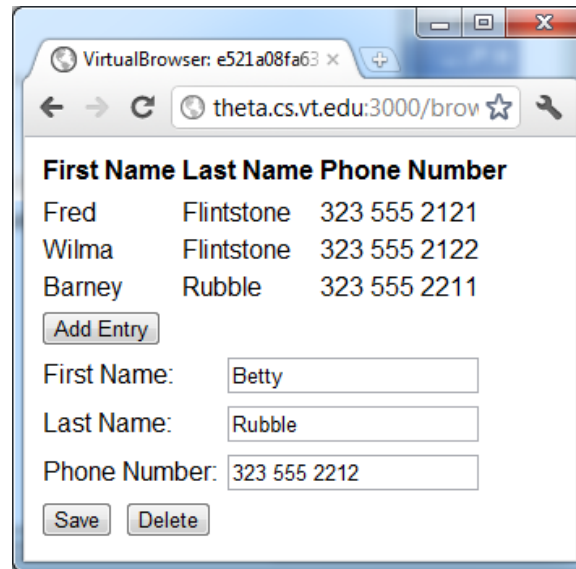


Figure 4.5: Phone Book Application Screen Shot

```

9   <br />
10  <button data-bind="click: $parent.saveToDB">Save</button>
11  <button data-bind="click: $parent.removeFromDB">Delete</button>
12 </div>

```

Figure 4.5 shows a screenshot of the resulting application.

The view model for this application is represented using the following code:

```

1  var vm = {
2    entries      : ko.observableArray(entries),
3    currentEntry : ko.observable(),
4    rowClick     : function () {
5      vm.currentEntry(this);
6    },
7    saveToDB : function () {
8      this.save();    // persist to db
9      vm.entries.remove(this);
10     vm.entries.push(this);
11     vm.currentEntry(null);
12   },

```

```

13  removeFromDB : function () {
14      this.destroy(); // remove from db
15      vm.entries.remove(this);
16      vm.currentEntry(null);
17  },
18  create : function () {
19      this.currentEntry(phoneBook.createEntry());
20  }
21 };

```

The view model for the application maps between Sequelize objects and Knockout observables. Here, some glue code is necessary to wrap the sequelized phone book entries that are mapped to the database. The callback functions which are invoked for the Save and Delete actions directly affect the persistent storage using methods provided by the reconstituted objects. This example demonstrates how CloudBrowser applications can avoid separate client- and server-side representations of their data, creating the appearance of a non-distributed environment in which application objects can be mapped to database records.

## 4.4 Co-browsing Based Scheduling Application

As a final example, consider a collaborative meeting scheduling application. In such an application, all participants manipulate a shared view of a set of data (meeting times). Since all data, including the interface, is shared among all participants, we can use the first method of storing model data discussed in Section 3.1: storing all data as JavaScript objects in a virtual browser and co-browsing to it. This architecture is different from the chat application, in which only parts of the model data were shared, but not the entire user interface.

The follow HTML excerpt describes most of the application interface shown in Figure 4.6:

```

1 <table>

```

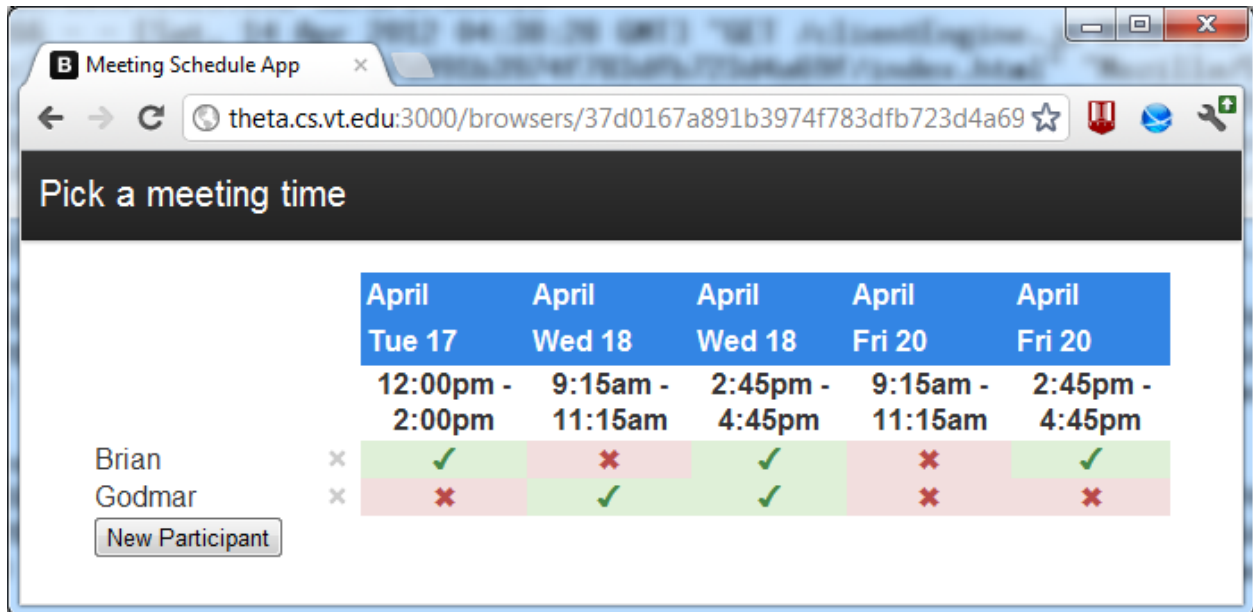


Figure 4.6: Meeting Time Application Screen Shot

```

2 <thead>
3   <tr><th width="25%"></th>
4     <!-- ko foreach: times -->
5     <th class="d-month container" data-bind="text: getMonth()"></th>
6     <!-- /ko -->
7   </tr>
8   <tr><th></th>
9     <!-- ko foreach: times -->
10    <th class="d-month container" data-bind="text: getDay()"></th>
11    <!-- /ko -->
12  </tr>
13  <tr><th></th>
14    <!-- ko foreach: times -->
15    <th class="container" data-bind="text: getTimeRange()"></th>
16    <!-- /ko -->
17  </tr>
18 </thead>
19 <tbody data-bind="foreach: participants">
20   <tr class="participant-row">

```



participant's name. The `css` binding applies different styles (red vs. green) depending on the currently indicated availability.

The following JavaScript code implements the scheduling behavior and controller logic for the scheduling application:

```

1 function Participant(name, editing) {
2   this.name = name;
3   this.editing = ko.observable(editing);
4   this.available = ko.observableArray();
5   for (var i = 0; i < appModel.times().length; i++)
6     this.available.push({avail: ko.observable(false)});
7 }
8
9 function Time(start, duration) {
10  this.getMonth      = function ...
11  this.getDay        = function ...
12  this.getTimeRange = function ...
13 }
14
15 var appModel = {
16   times: ko.observableArray([
17     new Time(1334678400000, 7200000),
18     // etc...
19   ]),
20   participants: ko.observableArray(),
21   addParticipant : function () {
22     appModel
23       .participants
24       .push(new Participant('New Participant', true));
25   },
26   removeParticipant : function (participant) {
27     appModel.participants.remove(participant);
28   }
29 };

```

```
30 ko.applyBindings(appModel);
```

Knockout's `applyBindings` code ensures that any changes to the observable members of the `appModel` object are reflected in changes of the DOM. Since the application is intended to be accessed via co-browsing, users will see the interactions of fellow co-browsers displayed in real time. The application code itself does not need to synchronize access to shared data, nor does it need to add any special handling for the simultaneous display to multiple users. Co-browsing adds real-time interaction for free.

Each virtual browser in CloudBrowser has its own unique URL. To facilitate co-browsing, users simply share the URL for their particular browser. This URL is valid for the lifetime of the virtual browser. Moreover, if a user closes their browser and reconnects later, even from a different device using a different session, they will be able to rejoin this meeting application instance by visiting its URL.

It is important to note that using co-browsing imposes restrictions on the application, since the entire user interface is shared among connected clients. If, for example, we wanted to add user accounts to our application, we would not be able to leverage co-browsing, since each user would need their own login form and user control panel. In that case, we would use an architecture more similar to the chat room application in Section 4.2.

Since we did not use any Node modules, we were able to program and test the entire logic of this application offline using a web browser within the confines of a JavaScript sandbox. As an extension, we can take advantage of Node's built-in `http` package to provide a JSON [19] service to export the current results, which is something that could not be done in a regular browser. The following code accomplishes this:

```
1 // if running in CloudBrowser, provide a
2 // JSON service to obtain current data
3 if (typeof require == "function") {
4   var http = require('http');
5   http.createServer(function (req, res) {
6     try {
```



```
7     res.writeHead(200, {"Content-Type": "application/json"});
8     res.end(ko.toJSON(appModel), "utf-8");
9   } catch (err) {
10     res.writeHead(500);
11     res.end("Server Error: " + err);
12   }
13 }).listen(1337);
14 }
```

We first detect if we are running in a Node environment by probing for the `require` method. If it's available, we know that we can use the `http` package to start an HTTP server, which acts as the JSON API server. The server itself simply responds to all requests with a JSON version of the model data.

# Chapter 5

## Implementation

This chapter first provides an overview of the different software components that comprise the underlying CloudBrowser framework, then looks at how these components are used to implement key features of CloudBrowser.

### 5.1 System Components

This section provides a high-level overview of each of the major components in the CloudBrowser system.

#### 5.1.1 HTTP Server

The HTTP server handles two major types of requests: initial requests and Socket.io connections. We have built the HTTP server on top of the popular Express [28] Node module, which provides support for sessions, URL based routing, form data parsing, and more.

On an initial request, the HTTP server allocates a new virtual browser for the client and loads it with the requested application. As detailed in Section 3.3, the HTTP server returns

to the client the necessary resources for loading and initializing the Client Engine. The Client Engine connects back to the HTTP server via Socket.io, which is the second type of request the server must handle.

The Socket.io library attaches to an Express server, adding its own routes for setting up two-way connections with the browser. The server creates a `Socket` object for each connection, which can be used to receive messages from the client or push messages back to it. When a client connects via Socket.io, the HTTP server looks up the requested virtual browser in a data structure called the Browser Manager, detailed in the next section, and passes the Socket.io socket to it.

### 5.1.2 Browser Manager

The Browser Manager is responsible for creating, organizing, and deleting virtual browsers. Each application hosted by a given HTTP server receives its own Browser Manager. Our system supports two types of browser creation strategies: in-process or multi-process. These strategies are represented as subclasses of a common Browser Manager base class, confining the details to a single point in the program and making it simple to add new strategies in the future.

The in-process Browser Manager creates new virtual browsers inside the same process as the HTTP server, and, therefore, all virtual browsers reside in the same process. This has the advantage of address space sharing, which facilitates the shared-data data binding techniques shown in the chat room example. The downside is that, since Node is event-driven, too much computation in an event can pause the entire HTTP server. Such long running events include HTTP parsing or loading JavaScript application code into a virtual browser. The multi-process Browser Manager creates each virtual browser inside of its own isolated process, which is discussed in detail in Section 5.4. This allows virtual browsers to perform computation without blocking the HTTP server.

### 5.1.3 Shared Components

As mentioned in Section 3.2, a key advantage of building both the client and server using JavaScript is the ability to share code between them. Before diving into the specifics of the Client and Server Engines, we will first examine these shared components.

CloudBrowser needs to synchronize the server-side virtual browser's DOM with the client-side browser's DOM. To do this, both sides need to agree on how to identify particular nodes in their respective trees, each of which should have a corresponding node in the other side's tree. To do this, we introduce a data structure we call the `TaggedNodeMap`. As nodes are created in a virtual browser, they are added to that browser's `TaggedNodeMap`. The `TaggedNodeMap` maps unique identifiers to DOM nodes. When the server needs to communicate updates to the client, or serialize the DOM entirely, it can identify particular nodes to modify or create by using the `TaggedNodeMap` identifier in place of the DOM node. The identifier is also stored as an `expando` property on the node object itself, allowing translation both ways. The Client Engine also maintains an instance of `TaggedNodeMap`, and the code is shared between the client and server verbatim. By sharing this code, we ensure that both sides are always using the same identifier rules. This also gives us a single place where we can change the identification scheme for both client and server. As another example of code sharing, we support compression for our update protocol instructions, and the compression classes are shared verbatim between the client and server.

### 5.1.4 Client Engine

As discussed in Section 3.3, the Client Engine is responsible for managing the state of the client browser and synchronizing it with its server-side counterpart. The Client Engine and server engine communicate using a Remote Procedure Call (RPC) channel which is implemented on top of the Socket.io [9] JavaScript library, which in turn encapsulates a number of underlying transport mechanisms (WebSockets for browsers that support them,

PageLoaded(records)
DOMNodeInsertedIntoDocument(records)
DOMNodeRemovedFromDocument(parent, target)
DOMAttrModified(target, name, value)
DOMPropertyModified(target, property, value)
DOMCharacterDataModified(target, value)
DOMStyleChanged(target, attribute, value)
AddEventListener(target, type)
PauseRendering()
ResumeRendering()

Table 5.1: Client Engine RPC Methods

or other AJAX-based long polling mechanisms such as Vault [46]).

During the bootstrap phase, the Client Engine initializes its different facilities, including the event monitor (Section 5.2.3), `TaggedNodeMap` (Section 5.1.3), and RPC channel. The RPC channel provides the methods shown in Table 5.1. In subsequent sections, we will examine how these methods are used by the server to manipulate the client-side DOM.

### 5.1.5 Server Engine

The Server Engine is part of the implementation of each virtual browser instance. It comprises an HTML parser and a complete implementation of the DOM Level 2 and DOM Event APIs, based on the JSDOM library [30].

The Server Engine has many responsibilities, including initializing and maintaining the virtual browser environment, bootstrapping newly connected clients, notifying connected clients of virtual browser DOM changes, and handling DOM events triggered by clients. The Server Engine is composed of multiple components that interact with each other through an event-based interface, which works naturally in the Node environment.

To detect changes to the virtual browser DOM, the Server Engine uses aspect-oriented techniques [32] (e.g., advice associated with the DOM manipulation methods) to interpose on

<code>processEvent(event)</code>
<code>setAttribute(target, attribute, value)</code>

Table 5.2: Server Engine RPC Methods

any changes to the server document. Consequently, unlike in server-centric frameworks such as ZK [18], neither the server document implementation nor the application code incur any additional implementation burden to ensure that server document changes are propagated to the client.

The interposed advice code invokes RPC methods on the client in order to manipulate the client’s DOM to synchronize it with the server-side virtual browser DOM. The Server Engine also exposes RPC endpoints to the client, enabling the client to forward events and to update DOM nodes in the virtual browser. The server’s RPC endpoints are shown in Table 5.2.

Lastly, the Server Engine includes a resource proxy and translator for style sheets and other resources used by an application. We rewrite references to those resources so that they refer to the rewritten resource. We maintain application-defined CSS style classes, element ids, and element structure, allowing us to send the style sheet mostly unchanged to the client, except where there are references to other resources (e.g. `@import`, or `url()` in `background-image`).

## 5.2 Key Implementation Aspects

### 5.2.1 Bootstrapping the Client

After a client has been redirected to a virtual browser URL (as discussed in Section 3.3), they must be synchronized with the current state of the virtual browser. This starts with the downloading of the Client Engine and base HTML page. When the Client Engine is done initializing, it creates a Socket.io connection back to the HTTP server, passing along the virtual browser ID to which it wants to connect. The HTTP server’s Socket.io handler

Property	Type	Description
type	String	The type of node: “text” or “element” or “comment”
id	String	The node’s unique identifier.
ownerDocument	String	The id of the document to which this node belongs
parent	String	The id of this node’s parent node.
name	String	For element nodes, the type of element (e.g. “div”)
value	String	For comment and text nodes, the content of the node.
attributes	Object	The node’s attributes, in with object properties as attribute names and property values as attribute values.

Table 5.3: DOM Node Record Format

uses the Browser Manager to find the corresponding virtual browser, and then passes the Socket.io socket object to it. At this point, the virtual browser is solely responsible for managing the socket connection.

Upon receiving a reference to a Socket.io socket, the virtual browser serializes the current DOM tree by performing a pre-order traversal of the DOM and generating a text representation of each DOM node. Thanks to the single-threaded nature of Node, it is guaranteed that the DOM tree will not change during the pre-order traversal. The record format for each node is shown in Table 5.3. The serialized DOM is sent to the client using the `PageLoaded` RPC method, where the Client Engine reconstructs the DOM in the client’s browser.

There is a subtle issue to consider when bootstrapping the client: it is possible for a client to create a Socket.io connection before the corresponding virtual browser is ready. In this case, incoming Socket.io sockets are stored in a queue. When the virtual browser’s `DOMContentLoaded` event fires, the DOM is serialized once and sent to all pending Socket.io sockets.

## 5.2.2 Synchronization Protocol

The synchronization protocol is responsible for keeping the Client Engine and Server Engine in sync. The synchronization protocol is built on top of the client-side RPC methods shown in Table 5.1, which grant the Server Engine control over the client’s browser.

As detailed in Section 5.1.5, we use aspect-oriented programming techniques to detect DOM changes in the virtual browser. The interposed advice code invokes the Client Engine's `DOM*` RPC methods, which allow the manipulation of client-side nodes. One of the key client-side methods, `DOMNodeInsertedIntoDocument`, uses the serialization functionality from the bootstrapping process. When bootstrapping, a serialized version of the entire DOM is sent to the client. When updating, the server sends a serialized version of only the newly added nodes, using the same record format in Table 5.3. To reduce the number of calls to the client, we do not send a DOM node until after it has been attached to the server document. Frequently, user interface libraries build complex structures of unattached DOM nodes before inserting them into the document. Sending these nodes to the client when they are inserted allows us to batch such updates by sending a serialized array of records representing a subtree of nodes.

A single event listener in the virtual browser will typically cause multiple updates to the server document. If these were sent to the client one-by-one and immediately applied to the client's document, flicker would result since the browser will re-render the document after delivering each RPC request. To avoid these unnecessary rendering cycles and prevent the associated flicker, the server surrounds all DOM updates occurring during an event handler with `PauseRendering` and `ResumeRendering` calls. `PauseRendering` instructs the Client Engine to buffer any DOM updates it receives until it receives the `ResumeRendering` call, at which point all DOM changes are applied to the client document. Batching the execution of multiple RPC requests by the Client Engine, rather than the Server Engine, allows us to reduce latency by overlapping the transmission of requests with the computation of additional requests.

To reduce the bandwidth consumption associated with client/server RPC calls, we exploit two compaction methods that are applied transparently to reduce the size of each RPC request message. First, instead of using method names, we use a numbered encoding for each RPC method. Since we do not use an IDL compiler, the encoding table is built dynamically by the Server Engine, and the method codes are broadcast to all connected clients as



necessary.

Second, instead of sending full DOM records as JSON objects based on the format shown in Table 5.3, we offer a compression option to use a positional encoding that avoids repeating the property names and omits those properties that are not used in a particular call. These encoding operations are implemented transparently in a separate layer whose code is shared between client and server and which is downloaded as part of the bootstrap library. Additional compression techniques such as GZip compression could be applied by the underlying transport, although WebSockets currently do not support any.

### 5.2.3 Client Event Processing

In traditional AJAX applications, JavaScript handlers are attached to DOM nodes and triggered when certain DOM events occur, such as a mouse click on an element. These event handlers typically modify the DOM in some way. In CloudBrowser, the event source (client browser) is decoupled from the event target (a node in the virtual browser's DOM). Client generated events are sent across the RPC channel for dispatch into the virtual browser DOM using the `processEvent` RPC method exposed by the virtual browser.

The Client Engine has two responsibilities with respect to event processing: detecting and forwarding client events to the Server Engine, and preventing default actions from triggering on the client. The Client Engine does not keep track of which specific elements have event listeners associated with them in the virtual browser application. Instead, it exploits capturing event handlers as defined in the DOM Event specification and discussed in Section 2.1.1. Any event that results from a user interaction is first dispatched to capturing event listeners associated with the event target's ancestors, starting with the document root. The Client Engine intercepts events here, encodes the event in a message, and forwards it to the Server Engine. The event's `stopPropagation` and `preventDefault` methods are then invoked to prevent the browser from further processing the event. Preventing the default actions via `preventDefault` means that, for instance, clicks on links represented by `<a>` elements do

not result in the user navigating away from the page, and therefore breaking out of the virtual browser. Instead, default actions are processed by the server engine and any resulting updates propagated to the client.

Instead of having the Client Engine blindly listen for all possible event types, the Server Engine infers which events are actually being listened for in the server-side document by intercepting calls to the `addEventListener` API. When a server-side event listener is added, the `AddEventListener` RPC is invoked at the client, instructing it to add a capturing event listener for that event type. This optimization avoids excessive client-server traffic for high-frequency event types, such as mouse movement events, unless the applications makes use of them. Certain types of events are always listened for on the client, such as mouse clicks, to avoid the inadvertent execution of client-side default actions. The Server Engine processes forwarded events sent by the Client Engine(s) with which it maintains connections. Table 5.2 shows the RPC methods exposed by the Server Engine's entry point. Besides the main `processEvent` method, which clients invoke when forwarding events, the server provides a `setAttribute` method that allows a client to set certain element attributes that might be accessed from within event handlers. For instance, when a `change` event fires on an `<input>` field, the change event listener expects to be able to access the current `value` attribute of the `<input>` element.

The `processEvent` method dispatches the received event to the server document according to the DOM Event specification, invoking registered event listeners and/or performing the default actions for certain events (e.g., navigation when a `click` event is dispatched on a `<a>` anchor). These event listeners are typically part of the application's controller code, and may directly or indirectly result in changes to the server side document.

## 5.2.4 JavaScript Execution

In addition to dispatching client events and forwarding DOM modifications to the client, the Server Engine must provide a faithful implementation of the host environment in which the

application code's can run. As discussed in Section 3.1, our goal is to provide an environment that is nearly indistinguishable from the environment familiar to web developers writing ordinary JavaScript code that operates on a client document. This is also essential for supporting existing JavaScript libraries.

JavaScript libraries rely on accurate behavior of the `window` object, which is an instance of the `DOMWindow` class [23]. The `window` object serves as the global object for the JavaScript environment. Any variables declared without the `var` keyword are implicitly added to the global object. Likewise, one can reference global object properties using the unqualified property name, provided that property has not been shadowed due to a non-global variable with the same name in an enclosing scope. Accesses to these global properties must be reflected on the actual window object, both for reading and writing. Furthermore, certain JavaScript invariants must be upheld, such as the expression `window === this` yielding true within the global scope.

Initially, JSDOM, the open-source DOM implementation we use for our Server Engine, used built-in Node methods from the `vm` [5] module to supply a JavaScript execution context. Unfortunately, many of the aforementioned semantics were not correct with this implementation, including a lack of global object property forwarding and no support for asynchronous callbacks (such as timers and `XMLHttpRequest` callbacks).

To provide a semantically valid execution context, we created a helper library called Contextify. Contextify is written in C++ using the Node addon API and the V8 embedding API. By using C++, we gain access to V8 context manipulation methods not made available to JavaScript code executing in Node. Contextify properly forwards global variable accesses, supports asynchronous callbacks, and upholds the global identity invariants such as `window === this`. Contextify has officially been merged into the JSDOM project, replacing the existing faulty JavaScript execution environment implementation and providing compatibility for a much wider range of JavaScript libraries and existing web applications.

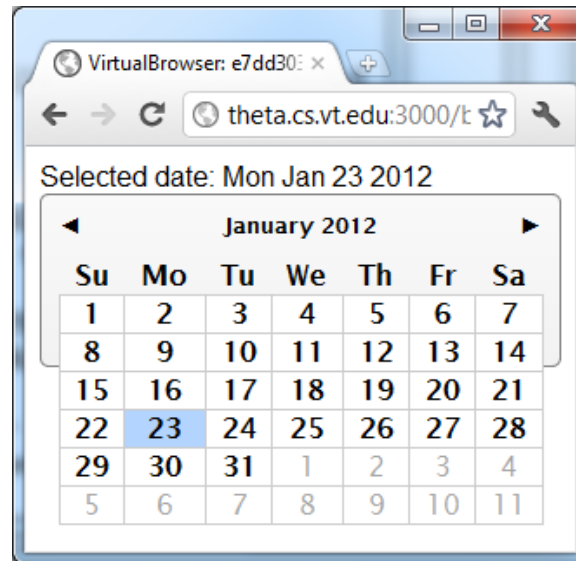


Figure 5.1: YUI3 Date Picker Component Screen Shot

### 5.3 Component-based Widgets

For interfaces that must rely on client-side computed positioning, we provide client-side and server-side glue code for components that allows them to be included in server documents. The client-side code includes the original component library's JavaScript code, instantiates client-side components, inserts them into the client document, registers event handlers, and provides way to set a component's properties. For instance, for a component such as a slider, the client glue code may set/get the slider's value, and register event handlers to listen for value changes. Like for ordinary events, the client-side glue forwards those events to the Server Engine.

The server-side glue code allows the application to instantiate components, which returns a server proxy object that the application code can use to interact with the component. For instance, the application can attach event listeners to this proxy object, which are fired when the client-side event listener fires and forwards the corresponding event to the Server Engine. We provide a cache to support direct access to properties. For instance, if application code accesses the `value` property of a proxied slider component, it will obtain the last known

snapshot of the component's value property. All component-related events send a copy of a component's properties to the Server Engine, allowing it to update this cache before executing event handlers.

As a proof of concept, we have implemented the glue code for two components: the YUI-3 Slider and DatePicker (calendar). Figure 5.1 shows a screenshot of the YUI-3 DatePicker component integrated into a CloudBrowser application. The following code excerpt shows how this example is implemented:

```

1 <script>
2 window.addEventListener('load', function () {
3   var vm = { selectedDate : ko.observable() };
4   ko.applyBindings(vm);
5
6   var picker = cloudbrowser.createComponent(
7     'calendar',
8     document.getElementById('datepicker'),
9     { // options
10      height : '100px', width : '300px',
11      showPrevMonth : true,
12      showNextMonth : true
13    });
14
15   picker.addEventListener('Calendar:dateClick',
16     function (e) {
17     var date = new Date(e.info.date)
18     vm.selectedDate(date.toDateString());
19   });
20 });
21 </script>
22
23 <body style="font-family: Arial">
24   <div>Selected date:
25     <span data-bind='text: selectedDate'></span>

```

```
26 </div>  
27 <div id='datePicker'></div>  
28 </body>
```

The `CloudBrowser.createComponent` API encapsulates access to the server-side glue for each supported component. We were able to encapsulate these components with relatively little effort; because of YUI-3's inheritance-based design, much of the code is reusable to support the inclusion of other components.

Providing support for components required changes to our DOM event capturing model in the Client Engine, as some events are now handled locally and must be passed through so the local browser can process them. In addition, the `PageLoaded` call was extended to instruct the client engine to load and instantiate the needed component libraries and instantiate the client-side glue.

## 5.4 Multiprocess Architecture

Single-threaded, event-based servers such as Node have the potential for high performance [51, 39, 41], but they cannot take advantage of multiple CPUs or cores. Moreover, long-running event handlers that involve such tasks as the parsing of large HTML documents delay the processing of subsequent events, increasing request latency. To overcome this problem, we defined a load balancing architecture that allows `CloudBrowser` instances to be distributed across multiple processes on a single machine or multiple machines, as shown in Figure 5.2.

Our architecture allows for flexible mappings of virtual browser instances to OS-level processes - each virtual browser may have its own dedicated process, or multiple virtual browser instances may share one process. Such co-location is required only if an application shares JavaScript state across instances.

A front-end server hands off requests to an application server that manages virtual browser

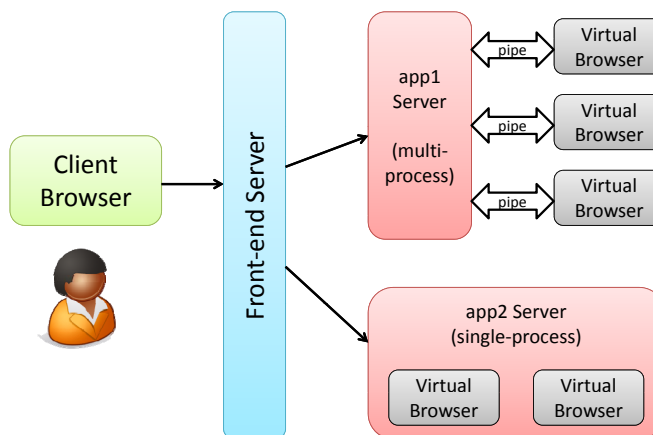


Figure 5.2: Load Balancing CloudBrowser Applications

instances for each application. Our current implementation uses client-side redirection (via a 301 HTTP response) to redirect users to the corresponding server for their requested application, but other mechanisms, such as passing the client's socket from the front-end server directly to the application server via `sendmsg(3)`, could be used. For multi-process arrangements, the application server forwards requests and responses via an inter-process communication mechanism (i.e., Unix pipes). Since this arrangement leaves the application server involved in each request/response, a more optimal approach would be to extend the Socket.io library to support the handing off of an established WebSocket connection directly to the corresponding process.

A multi-process architecture allows the HTTP server to continue handling requests while the virtual browsers are performing computations. Fault tolerance is also increased, since an error in one virtual browser process will not propagate to other browsers or the HTTP server. Unfortunately, isolating each virtual browser to its own process prevents the direct sharing of data between virtual browsers. For situations like browsing existing web pages, or applications requiring no shared data access, the multi-process strategy provides a more robust application.

# Chapter 6

## Evaluation

We have evaluated our prototype in terms of memory usage, event-processing latency, bandwidth consumption, and the completeness of our server-side DOM implementation.

### 6.1 Memory Usage

We measured two aspects of memory usage: the cost of allocating a virtual browser and the additional cost of adding a client to an existing virtual browser when co-browsing. We created a benchmark application that spawns an instance of the CloudBrowser server and connects a configurable number of clients to it. The clients can connect to existing virtual browsers or force the creation of new ones. In between each connection, we force a full garbage collection cycle on the server and record memory usage as reported by the Node `process.memoryUsage()` API, which reports the size of the live heap maintained by the V8 virtual machine.

The memory requirements of a virtual browser depend on the size of the HTML document describing the application, as well as the amount of CSS and JavaScript code, which must be parsed and compiled by the V8 engine.



Using our x86\_64 Node implementation, we found the base memory consumption for an empty browser (with just 3 DOM nodes for `<html>`, `<head>`, and `<body>`) to be 164 KB. Including the jQuery 1.7.2 library and the Knockout.js 2.0.0 libraries increases this consumption by 1.05 MB and 0.33 MB, respectively. The chat application discussed in Section 4.2, which in addition includes the Bootstrap CSS stylesheets, consumes about 2.6 MB per browser instance. We see two opportunities for optimizations that have the potential to reduce this memory usage drastically. First, the V8 engine could recognize if the same JavaScript code is included in multiple browsers and transparently share the resulting intermediate representations and machine code, a technique commonly exploited in multitasking Java virtual machines such as MVM [21]. Second, the CSS implementation could similarly recognize when style sheets are included multiple times and share the immutable portions of their representation.

Adding additional clients to an existing virtual browser adds only minimal overhead of about 16KB per connection, independent of the memory consumed by the virtual browser.

## 6.2 Latency

To measure latency, we ran a single-process CloudBrowser server on a server machine with two AMD Opteron 2380 2.5GHz quad-core processors and 16GB of RAM. Our simulated clients run within multiple processes (100 clients per process) on a separate machine with an Intel Q9650 quad-core 3.00GHz processor with 8GB of RAM. The two machines were connected via a gigabit LAN.

The simulated clients each connect to the CloudBrowser server and request a new virtual browser instance. The following CloudBrowser application was run in each virtual browser:

```
1 <html >
2   <head></head >
3   <body >
4     <div id='target'></div >
```

```
5     <script>
6         var count = 0;
7         var div = document.getElementById('target');
8         div.addEventListener('click', function () {
9             div.innerHTML = ++count;
10        });
11    </script>
12 </body>
13 </html>
```

Listing 6.1: The benchmark application.

The clients send a click event object corresponding to a click on the `div` element, which triggers an event handler that modifies the DOM by setting the `innerHTML` property of an element, which triggers the necessary RPC requests to the client (in this case, a `DOMNodeRemovedFromDocument` followed by a `DOMNodeInsertedIntoDocument` call). The client measures the elapsed time between sending the event and receiving the `resumeRendering` RPC call, signifying the end of the DOM updates for that event. Once a client receives a response, it sends another event. This simulates a user that interacts with a page, waits to see the results of their action, and then interacts with the page again. To model a more realistic use case, we added a uniformly random delay between 1 and 5 seconds before the client submits the next request, modeling the frequency with which a human might interact with the application.

We measured the average latency for each client while increasing the number of connected clients (up to 1000, with a step size of 50). Each step was run with fresh CloudBrowser server and client processes. When clients are sending requests back-to-back, latency increases linearly by about 1.4ms per client for this particular interaction, which is shown in Figure 6.1. The different arrival process that results from a 1-5 second delay that models active human users results in the latency characteristics shown in Figure 6.2. These results show that for this benchmark, a single CPU can support up to 600 clients interacting with an equal number of virtual browser instances before the average delay exceeds 37 ms. These

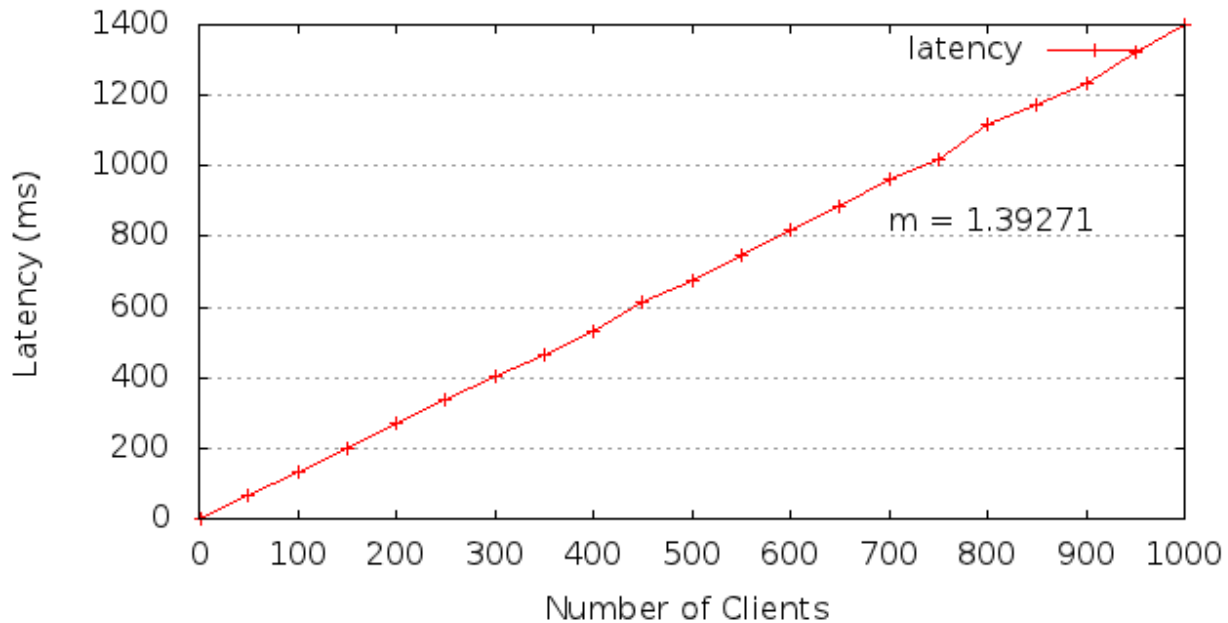


Figure 6.1: Average Latency without Simulated WAN Delay

results were obtained over a LAN; a WAN deployment would incur added latency equal to the connection’s TCP round-trip time, which primarily depends on the propagation delay introduced by geographical distance and the queuing delay due to network congestion. As a point of comparison, Keynote’s Internet Health report considers latencies of less than 90ms between major US backbone providers “healthy” [31].

A well-known result from usability engineering research [36] holds that response times of less than 100ms feel instantaneous to the user, and that response times between 100ms and 1 second, while noticeable, allow uninterrupted workflows. Our results show that CloudBrowser is able to support an acceptable number of users economically for applications that benefit from the interaction style that motivates our approach. We expect that the use of multiple processes, as discussed in Section 5.4, could further increase the number of supported clients without adding significant latency, especially when connections are handed off to separate processes.

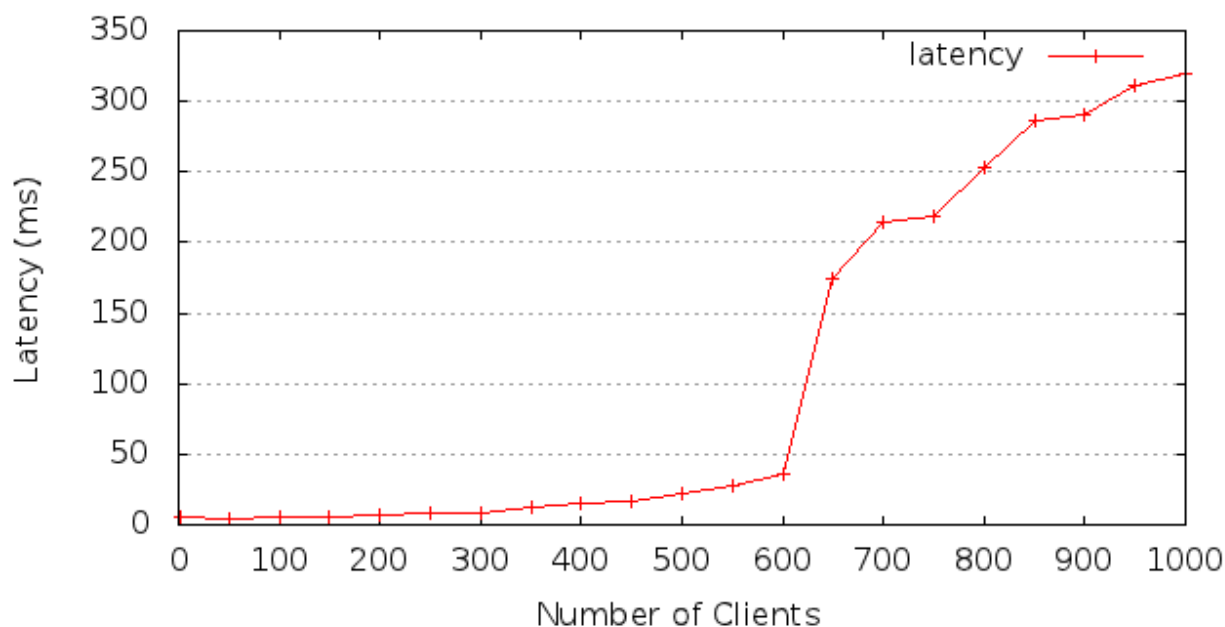


Figure 6.2: Average Latency with Simulated Client Mental Processing Delay

### 6.3 Bandwidth Consumption

Although not as important as latency, bandwidth consumption is an important performance indicator, particularly in metered cloud environments. CloudBrowser consumes bandwidth during the bootstrapping process to download an application’s initial DOM snapshot and for client and server engine RPCs. This section provides estimates for each of these components. We instrumented our server to count the number of bytes sent and received at the TCP level. We selected the WebSocket transport mode for the client/server communication and used Google Chrome (v16) as our client browser.

Table 6.1 shows the sizes of the bootstrap files sent to the client. The JavaScript code (our Client Engine, and the Socket.io Client and jQuery libraries we use) is minified and GZip-compressed. Their bandwidth consumption is small when compared to the amount of JavaScript code that is transferred to clients in contemporary AJAX applications, and they can be cached across different applications.

Client Engine	7.44KB
Socket.io Client	8.18KB
jQuery	29.09KB
Base HTML	771 bytes
<b>Total</b>	45.46KB

Table 6.1: Sizes of Static Bootstrap Files

Site	Snapshot Size	Raw HTML
twitter.github.com/bootstrap	276.03 KB	82.06 KB
news.ycombinator.com	54.60 KB	22.92 KB
ebay.com	123.87 KB	70.86 KB
reddit.com	169.57 KB	84.03 KB

Table 6.2: Serialized DOM Snapshot Sizes vs. HTML Text Size

Table 6.2 compares the initial DOM snapshot sent to a CloudBrowser client compared to the equivalent HTML that would be sent to a regular browser when loading the same page for selected URLs. The record-based serialized representation of the DOM snapshot eliminates the need for a parser, but introduces an increase in size ranging from 1.7x to 3.4x when considering uncompressed sizes. As mentioned in Section 5.1.5, WebSockets do not currently support GZip compression, although such an optimization is being considered [52]. GZip compression at the WebSocket layer would transparently reduce the required bandwidth.

The bandwidth consumed during server engine RPC calls is small, typically around 300 bytes for serialized events. A `setAttribute` call, if required, adds an additional 70-80 bytes. The bandwidth consumed for client engine calls depends on the number of DOM elements changed; we would expect a similar ratio when compared to the size of an equivalent HTML representation as for the initial DOM snapshot.

Test Suite	Pass	Total	% Passed
Core	1306	1309	99.77
Callbacks	418	418	100
Deferred	155	155	100
Support	28	38	73.68
Data	290	290	100
Queue	32	32	100
Attributes	453	473	95.77
Events	476	482	98.75
Selector (Sizzle)	310	314	98.72
Traversing	297	298	99.66
Manipulation	530	547	96.90
CSS	58	93	62.37
AJAX	329	349	94.26
Effects	367	452	81.19
Dimensions	61	83	74.49
Exports	1	1	100
Offset	N/A		
Selector (jQuery)	N/A		

Table 6.3: jQuery Test Suite Performance

## 6.4 DOM Conformance

To measure the completeness of our virtual browser implementation, we have used the jQuery test suite (version 1.7.1), which includes 5828 tests that exercise all aspects of the jQuery JavaScript library. We run the tests in their unmodified QUnit test harness inside a virtual browser, which we visit to observe the results.

The results of running the jQuery test suite are shown in Table 6.3. The results show that our server document implementation is mature enough to pass a majority of the jQuery tests. This result, which reflects the implementation effort invested so far by us and the developer community supporting JSDOM, indicates that a complete server-side implementation of DOM specification is feasible with additional engineering effort. We use jQuery heavily for our administrative interface, which is written itself as a CloudBrowser application.

We also compared the time it took to run the jQuery test suite inside a virtual browser

to the time it takes when run in the Google Chrome browser. We observed a slowdown of roughly 15x, indicating a tremendous potential for optimizations in the server document implementation.

# Chapter 7

## Related Work

This section explores projects that are related to CloudBrowser. Existing server-centric frameworks are discussed first, with a focus on the design decisions made by the framework developers that differentiate their frameworks from CloudBrowser. Next, applications of server-side browsing environments is discussed.

### 7.1 ZK

ZK [18] is a Java-based server-centric web framework that is in wide use. ZK applications are constructed using components, which are represented using the ZK User Interface Markup Language (ZUML) or created programmatically using Java. Developers construct these components on the server, and the components act as the server-side representation of the client-side view. Since the components on the server are not represented as HTML and CSS, each component must be translated to HTML and CSS so that it can be displayed on the client. ZK's choice of using a higher-level user interface description language offers the benefit of allowing ZK programs to target non-HTML based platforms. For example, ZUML pages can instead be rendered to SVG or XML. In practice, this disparity between



the server- and client-side view representations can cause many issues for developers. First, the translation process can create difficulty during debugging, as discussed in 2.2.4. The translation process means that developers are only indirectly influencing the view, instead of programming it directly. This is in contrast to CloudBrowser, where the server-side view representation matches exactly the client-side representation. Second, the component translation process itself is error-prone, and can introduce bugs that are outside of the developer's control. CloudBrowser sidesteps this issue since it does not require a component translation process. Third, detecting changes to components, which in turn require updates to be sent to the client, must be handled manually by the component developer. CloudBrowser automatically detects changes to the server-side representation using aspect-oriented programming, as discussed in Section 5.1.5, so the client and server representations are always synchronized without explicit effort from the framework developer.

Similarly to CloudBrowser, ZK uses a client-side library to handle synchronization between the client's view of and interaction with components, and their server-side representation. Our extensive experience deploying applications with ZK [47, 25] inspired the work on CloudBrowser. Compared to CloudBrowser, ZK does not maintain a representation of the server document across HTTP requests. Server-side components are instantiated anew on each request. To reconstruct client-side user interface state, a developer must manually store hints in server-side session-state, and then on each request, use these hints to bring the UI back into the desired initial state, which may be far from the state it was in when the user last visited it. ZK also does not support multiple users connecting to a single application instance (co-browsing), which CloudBrowser supports naturally.

Unlike CloudBrowser, ZK aims to support layout attributes, but we have found that the complexity of its client engine leads to numerous layout and compatibility bugs developers must work around, particularly when the server-side document and the client-side document are not identical. ZK also allows developers to break from the server-centric model and specify sections of code which run locally on the client. This is accomplished using a special XML namespace. CloudBrowser allows developers to break from the server-side document

model, but only in controlled, compartmented components, as outlined in Section 3.4. Even in this case, the components have a corresponding server-side representation, which is treated as a black box by developers.

## 7.2 ItsNat

ItsNat [44] is a Java-based AJAX component framework similar to ZK, although it uses HTML instead of ZUML to express server documents, along with the Java W3C implementation. ItsNat differs from CloudBrowser in two areas. First, it does not maintain the server-side document across page visits. Similarly to ZK, each visit receives a fresh allocation of server-side resources, and client-side state must be manually restored from server-side session state. Second, ItsNat programs are written in a mix of HTML, CSS, and Java. Application code is implemented as Java event listeners, which are registered on DOM elements in the server-side document. ItsNat applications cannot take advantage of existing client-side JavaScript libraries in the server-side DOM. ItsNat does provide a mechanism for using client-side libraries only on the client, which breaks away from the server-centric paradigm. The closest parallel in CloudBrowser is the component implementation, but CloudBrowser components have a corresponding representation in the server-side document. Since ItsNat programs are written primarily in Java, web developers must learn new APIs and programming languages to write ItsNat applications, as opposed to CloudBrowser, which allows developers to leverage their existing client-side development skills.

Similar to CloudBrowser, ItsNat does include co-browsing support. ItsNat's implementation includes options to further customize the co-browsing experience. ItsNat supports a read-only mode, where multiple users may browse the same document, but events are only processed from a single user. Furthermore, ItsNat supports allows users to grant or deny access to their browsing session, while co-browsing is enabled by default in CloudBrowser.

## 7.3 Fiz

Fiz [37, 38] is a server-centric component-based AJAX framework that uses page properties to maintain component state on the server. The server-side implementations of Fiz components can be balanced across multiple machines, allowing for horizontal scaling. Fiz does not present the abstraction of a server-side document to application developers, but it provides a way to build component-based web applications, and simplifies the development of additional components within its framework. Fiz shares many similarities with ZK, being a Java, component-based AJAX framework. Fiz introduces the ideas of page properties and reminders, which are mechanisms for storing user state for AJAX components on the server and client, respectively. This stored state must be manually recalled, and so Fiz does not automatically remember user interface state like CloudBrowser does.

## 7.4 Ripley

Ripley [48] uses server-side browser emulation and event processing to ensure the integrity of client-side computation in AJAX applications. Ripley is integrated with Volta, a distributing compiler that partitions .NET applications between client and server. Ripley uses a server-side, headless browser that performs no layout computation, similar to CloudBrowser. Client-side events are sent to the server-side browser where they are dispatched into a server-side DOM. Unlike CloudBrowser, the events are not intercepted at the client - they are simply mirrored on the server. The JavaScript code still runs locally on the client. With Ripley, the resulting server-side DOM changes are used only to verify that the client has not sent malicious code or data to the server, and the client- and server-side DOMs are compared after event processing. For the example application studied, Ripley used around 1.3 MB of memory for each server-side DOM, which is similar to CloudBrowser's memory usage.

## 7.5 Crawljax

Crawljax [35] is web crawler that supports AJAX applications, which are commonly ignored by search engines due to their reliance on client-side computation. Crawljax explores AJAX applications using a programmatically controlled browser via the Selenium testing framework [7]. Unlike CloudBrowser, Selenium uses off-the-shelf browsers (IE, Chrome, Firefox) that render into a framebuffer display device that is not made visible to the user. This is a heavyweight approach, which would impose large resource requirements on the server.

## 7.6 Opera Mini

Opera Mini [6] is a mobile browser that optimizes the client experience by offloading browser rendering to a server. Compression algorithms are run on the rendered output from the server, reducing bandwidth requirements. While Opera Mini does use a server-side DOM representation, its goals are inherently different from CloudBrowser. Thus far, we have not tested CloudBrowser on mobile devices, but the popularity of Opera Mini shows that it may be possible to leverage server-side computation to provide a better mobile experience than traditional AJAX applications.

# Chapter 8

## Conclusion

### 8.1 Future Work

While our prototype implementation provides a platform suitable for developing real world applications, there are still opportunities for performance improvement. Rendering pages with large amounts of HTML can result in a noticeable delay on the client. Ad hoc experimentation has led us to believe that the majority of the delay occurs in the HTML parser. There are efforts underway to improve the performance of our chosen HTML parser and DOM implementation. Since we have built CloudBrowser on popular open-source libraries, our framework automatically benefits from performance improvements in the underlying platform. Performance could also be increased by further optimizing the synchronization protocol, such as by switching to a more compact binary protocol using Node Buffers [4] on the server and TypedArrays [50] on the client. As discussed in Section 5.4, we are also investigating ways to optimize the multiprocess architecture.

We also plan on further development of an API for facilitating the building and distribution of CloudBrowser applications. This also implies the identification of development best practices using this new paradigm. For example, we found that data bindings for shared server-side

data increased our productivity when developing CloudBrowser applications.

## 8.2 Summary

As more and more applications move from the desktop to the web, developers must create web applications that mimic the behavior of desktop applications. Users expect applications to behave like applications - not web pages. Delivering this user experience is difficult due to the distributed nature of the web. Client state lives in the client's browser, and is forgotten on page refresh or navigation. Remembering user interface state requires manually storing hints on the server, and reconstructing the state for each user.

CloudBrowser is a server-centric web application framework for the development of rich Internet applications that embodies a new paradigm for server-centric web applications: keeping both application and presentation state on the server in a virtual browser environment. As a result, web application development is greatly simplified because both the stateless and the distributed nature of the web is hidden from the developer. User interface state is automatically preserved in the server-side virtual browser, so users can leave and return to their application later, finding it in the same state in which they left it. Web developers can use existing client-side libraries and skill sets since the server-side environment mimics that of a browser. This use of client-side libraries also allows the CloudBrowser framework to automatically benefit from the continuous innovation occurring in the client-side JavaScript ecosystem.

We have developed a prototype environment and several applications, which indicates that a server-centric approach is feasible and desirable for web applications in which users expect that most interactions with the user interface result in updates that are immediately stored “in the cloud,” even across page visits. Furthermore, we have identified best practices when developing with this new paradigm, such as using data bindings to bind presentation state to shared data, and provided a development API to aid programmers in taking advantage

of this new way of creating web applications.

# Bibliography

- [1] CommonJS Modules Specification 1.0. <http://wiki.commonjs.org/wiki/Modules/1.0>.
- [2] Google V8. <https://developers.google.com/v8>.
- [3] MySQL. <http://mysql.com>.
- [4] Node.js Buffer Module. <http://nodejs.org/docs/v0.6.15/api/buffer.html>.
- [5] Node.js vm module. <http://nodejs.org/docs/v0.6.15/api/vm.html>.
- [6] Opera Mini Mobile Browser. <http://www.opera.com/mobile>.
- [7] Selenium Web Browser Automation. <http://seleniumhq.org>.
- [8] SkyFire Mobile Browser. <http://www.skyfire.com/en/for-consumers/android/android>.
- [9] Socket.io. <http://socket.io>.
- [10] TameJS. <http://tamejs.org>.
- [11] The World Wide Web Consortium. <http://www.w3.org>.
- [12] Yahoo! User Interface Library. <http://yuilib.com>.



- [13] Atul Adya, Jon Howell, Marvin Theimer, William J. Bolosky, and John R. Douceur. Cooperative Task Management Without Manual Stack Management. In *Proceedings of the USENIX Annual Technical Conference*, pages 289–302, Monterey, CA, USA, June 2002. USENIX Association.
- [14] Jeremy Ashkenas. Coffeescript. <http://coffeescript.org>.
- [15] Godmar Back. On widths and percentages. [http://en.wikibooks.org/wiki/ZK/How-Tos/Concepts\\_and\\_Tricks#On\\_Width\\_and\\_Percentages](http://en.wikibooks.org/wiki/ZK/How-Tos/Concepts_and_Tricks#On_Width_and_Percentages).
- [16] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible Markup Language (XML) 1.0 (Fifth Edition). *W3C Recommendation*, 2008. <http://www.w3.org/TR/REC-xml/>.
- [17] Steve Burbeck. Application Programming in Smalltalk-80: How to use Model-View-Controller (MVC). Technical report, University of Illinois in Urbana-Champaign (UIUC).
- [18] Henri Chen and Robbie Cheng. *ZK: Ajax without the Javascript Framework*. Apress, Berkely, CA, USA, 2007.
- [19] Douglas Crockford. JSON. <http://www.json.org>.
- [20] Ryan Cunningham and Eddie Kohler. Making events less slippery with eel. In *Proceedings of the 10th conference on Hot Topics in Operating Systems, HOTOS'05*, Santa FE, NM, USA, June 2005. USENIX Association.
- [21] Grzegorz Czajkowski and Laurent Daynés. Multitasking without compromise: a virtual machine evolution. In *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, OOPSLA '01*, pages 125–138, Tampa Bay, FL, USA, October 2001. ACM.
- [22] Ryan Dahl. Node.js. <http://nodejs.org>.

- [23] Ian Davis and Maciej Stachowiak. Window Object 1.0. *W3C Working Draft*, 2006. <http://www.w3.org/TR/Window/>.
- [24] Sascha Depold. Sequelizeize. <http://sequelizejs.com>.
- [25] Stephen H. Edwards and Godmar Back. Bringing creative web 2.0 programming into CS1: conference workshop. *J. Comput. Sci. Coll.*, 26(3):54–55, January 2011.
- [26] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, USA, 1994.
- [27] Jesse Garrett. AJAX: A new approach to web applications. <http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications>, 2005.
- [28] TJ Holowaychuk, Ciaran Jessup, Aaron Heckmann, and Guillermo Rauch. Express - node web framework. <http://expressjs.com>.
- [29] Arnaud Le Hors, Philippe Le Hégarret, Lauren Wood, Gavin Nicol, Jonathan Robie, Mike Champion, and Steve Byrne. Document Object Model (DOM) Level 2 Core Specification. *W3C Recommendation*, 2000. <http://www.w3.org/TR/DOM-Level-2-Core/>.
- [30] Elijah Insua. JSDOM. <http://jsdom.org>.
- [31] Keynote Systems, Inc. Internet health report. <http://www.internetpulse.net/>.
- [32] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming, ECOOP'97*, pages 220–242, Jyväskylä, Finland, June 1997.
- [33] Maxwell Krohn, Eddie Kohler, and M. Frans Kaashoek. Events can make sense. In *Proceedings of the USENIX Annual Technical Conference, ATC'07*, pages 87–100, Santa Clara, CA, USA, June 2007. USENIX Association.

- [34] Caolan McMahon. Async.js. <https://github.com/caolan/async>.
- [35] Ali Mesbah, Arie van Deursen, and Stefan Lenselink. Crawling Ajax-Based Web Applications through Dynamic Analysis of User Interface State Changes. *ACM Trans. Web*, 6(1), March 2012.
- [36] Jakob Nielsen. *Usability Engineering*. Morgan Kaufmann, 1st edition, September 1993.
- [37] John Ousterhout. Fiz: A component framework for web applications. Technical report, Dep. of CS, Stanford University, 2009.
- [38] John Ousterhout and Eric Stratmann. Managing state for Ajax-driven web components. In *Proceedings of the 1st USENIX Conference on Web Application Development*, WebApps'10, pages 73–85, Boston, MA, USA, June 2010. USENIX Association.
- [39] David Pariag, Tim Brecht, Ashif Harji, Peter Buhr, Amol Shukla, and David R. Cheriton. Comparing the performance of web server architectures. *SIGOPS Oper. Syst. Rev.*, 41(3):231–243, March 2007.
- [40] Tom Pixley. Document Object Model (DOM) Level 2 Events Specification. *W3C Recommendation*, 2000. <http://www.w3.org/TR/DOM-Level-2-Events>.
- [41] Will Reese. Nginx: the high-performance web server and reverse proxy. *Linux J.*, 2008(173), September 2008.
- [42] John Resig. jQuery. <http://jquery.com>.
- [43] Steve Sanderson. Knockout. <http://knockoutjs.com>.
- [44] Jose Maria Arranz Santamaria. ItsNat: Natural AJAX. component based Java web application framework. <http://itsnat.sourceforge.net>.
- [45] Isaac Schlueter. npm - Node Package Manager. <http://npmjs.org>.

- [46] Eric Stratmann, John Ousterhout, and Sameer Madan. Integrating long polling with an MVC framework. In *Proceedings of the 2nd USENIX Conference on Web Application Development*, WebApps'11. USENIX Association, June 2011.
- [47] Eli Tilevich and Godmar Back. "Program, enhance thyself!": demand-driven pattern-oriented program enhancement. In *Proceedings of the 7th International Conference on Aspect-Oriented Software Development*, AOSD '08, pages 13–24, Brussels, Belgium, 2008. ACM.
- [48] K. Vikram, Abhishek Prateek, and Benjamin Livshits. Ripley: automatically securing web 2.0 applications through replicated execution. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 173–186, Chicago, Illinois, USA, November 2009. ACM.
- [49] Rob von Behren, Jeremy Condit, and Eric Brewer. Why events are a bad idea (for high-concurrency servers). In *Proceedings of the 9th conference on Hot Topics in Operating Systems*, HOTOS'03, Lihue, HI, USA, 2003. USENIX Association.
- [50] Vladimir Vukicevic and Kenneth Russell. Typed Array 1.0 Specification. <http://www.khronos.org/registry/typedarray/specs/1.0>.
- [51] Matt Welsh, David Culler, and Eric Brewer. SEDA: an architecture for well-conditioned, scalable internet services. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 230–243, Banff, Alberta, Canada, October 2001. ACM.
- [52] Takeshi Yoshino. WebSocket Per-frame DEFLATE Extension. Technical report, IETF Secretariat, Fremont, CA, USA, August 2011.