The CloudBrowser Web Application Framework

Brian McDaniel Godmar Back

Department of Computer Science
Virginia Tech
brianmcd@vt.edu gback@cs.vt.edu

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

Categories and Subject Descriptors D.2.2 [*Software Engineering*]: Design Tools and Techniques User Interfaces; H.5.3 [*Information Interfaces and Presentation*]: Group and Organization Interfaces Web-based Interaction

Keywords web application framework, AJAX, server-centric, remote display, PaaS, cloud applications

1. Introduction

More and more applications are moving from the desktop to the web. Web applications can be accessed from any web browser, regardless of underlying platform, allowing them to be deployed and updated instantly. Users have begun to expect rich and expressive user interfaces whose single-page design mirrors that of desktop applications. At the same time, users assume that the data on which these applications operate resides "in the cloud," which stores any changes immediately and persistently. Increasingly, users expect that the state of the user interface is retained across page navigation and sessions so they can pick up where they leave off.

The creation of such rich Internet applications within the context of the current web infrastructure is difficult for multiple reasons. Application developers and framework designers must decide how to split the application's user interface code and its business logic between client-side JavaScript code and server-side code, and how to structure the communication between the client and the server on top of the stateless HTTP protocol.

Traditional AJAX [18] applications, which are developed using a mix of client- and server-side programming, fully expose developers to the underlying infrastructure's distributed nature. Developers must write view logic to produce an initial rendering of the user interface, then implement client-side controller logic to track the user interface state using JavaScript, use some variation of AJAX to inform the server of relevant changes to the application data, and incorporate any server responses into the HTML document that is rendered for the user. If the user refreshes the page, or makes use of the browser's navigation buttons, the ephemeral client state must be restored from scratch, often using hints stored on the server, because there is no automatic way of preserving user interface state.

To address these problems, server-centric AJAX frameworks [11, 27] move all application logic to the server, hiding most or all client-side programming from the developer. Such frameworks maintain the view state for each visit in a server-side representation, such as a document in a framework-specific, higher-level markup language that provides elements that represent user interface components. These components encapsulate the initial rendering into HTML, manage the forwarding of client-side events to server-side controller logic, and handle the propagation of any resulting updates to the client-side document with which the user interacts.

Existing server-centric frameworks suffer from multiple shortcomings. First, since they instantiate components for each visit, they also do not automatically preserve user interface state across visits. Second, they still require the programmer to synchronize updates to the application's model

SPLASH'12, October 19–26, 2012, Tucson, Arizona, USA.

state across visits. Third, in practice they can make styling difficult, since the specifics of the rendering strategy used by these high-level components is encapsulated in their implementation. Fourth, the necessary computation of incremental client-side document updates after mutating the server-side view is tedious and error prone. Fifth, these server-centric frameworks often do not leverage the numerous JavaScript libraries that have been developed to facilitate the interaction with HTML documents, and thus cannot leverage the substantial skill sets developers have acquired.

This paper presents CloudBrowser, a web application framework that addresses these concerns. CloudBrowser maintains an application's user interface state server-side, as a document in a headless, virtual browser. The application logic interacts with the server-side representation in a manner similar to how a desktop application interacts with a graphical user interface (GUI) library, by creating and manipulating components and listening for events fired in response to user interactions. This design hides the distributed nature of the web from the application developer, because all client-server communication is encapsulated inside our framework. Clients connecting to application instances mirror the user interface state using a synchronization/update protocol we have developed. CloudBrowser automatically interposes on any changes to the server-side document, removing the need to manually compute updates. For efficiency, the actual layout and rendering of user interface elements is performed inside the actual browser by the client, rather than the virtual browser on the server.

CloudBrowser uses exclusively HTML, CSS, and JavaScript to express the user interface and its interaction with the application, allowing us to leverage existing libraries and developer skill sets, and avoiding any semantic overhead associated with a translation from high-level components to low-level components. Since CloudBrowser application instances persist across visits, this design naturally handles page navigation and refresh. It also provides a natural cobrowsing ability since it can support simultaneous display to multiple clients.

CloudBrowser is targeted at developing web applications in which most user interactions trigger persistent changes to the application data that is stored on the server, and which do not require read access to the computed layout from the controller logic. Where necessary, CloudBrowser can be extended using traditional client-side components represented by proxy objects on the server.

We have developed a prototype of our framework and implemented several example applications, including some that use sophisticated JavaScript libraries. We have found that our framework greatly simplifies the development of the web applications we target and that it imposes acceptable latency overhead and bandwidth costs. We are currently creating a Platform-as-a-Service (PaaS) infrastructure based on CloudBrowser.

2. Motivation

We motivate our approach using several example application scenarios. First, consider an example application such as the popular meeting scheduling service 'Doodle' (doodle.com). A user may initiate event scheduling by entering a set of possible meeting times, which are displayed to potential participants on a specially crafted webpage. Participants then enter their name, check boxes indicating their preferences when to meet, and hit a submit button, which navigates to a new page that displays their preferences along with the preferences of all other participants who have entered their preferences so far.

The user experience of Doodle, as well as many other currently available similar services, could in our opinion be significantly improved. For instance, a user does not see what other participants have entered until they submit their own, and subsequently only when they refresh or revisit the page, even when the potential meeting participants visit the page at around the same time. In addition, if the process of entering their name and checking appropriate times is interrupted, perhaps because the user clicked an ad and navigated to another page and returned to the page, the user will need to start over. Lastly, if the user overlooks the submit button before closing the page, their submission will not reach the server at all, which frequently happens to users accustomed to the single-click style used in their native OS (i.e., Mac OSX).

Second, consider a social forum application such as Piazza (piazza.com), which provides an interactive Q & A forum for instructional settings in which students can post questions to other students and to instructors. From an instructor's perspective, the Piazza user interface presents a constantly changing dashboard - new questions are being posted, questions are withdrawn, marked as answered by other instructors or students, or archived (hidden from view) after being answered. If the Piazza application is used from multiple computers (say, the instructor's work PC, work laptop, home PC, and perhaps a mobile device), the dashboard views are not kept in sync: already answered questions appear as unanswered when the instructor revisits the class site after returning home from work, forcing a manual refresh of the page. When such a refresh happens, some user interface state is lost - for instance, a different course may be selected, a homepage displayed instead of a student posting, displayed toolbars may disappear and have to be reenabled, or selected elements in an accordion-style display are not remembered, requiring the instructor to find the point in the application at which to resume responding to student questions.

Similar weaknesses apply to applications such as Google's Email service GMail. Although it is able to remember coarse information about which emails the user has read, it generally does not remember fine-grained state such as which emails in a thread the user looks at are shown in collapsed view and which are not, often requiring manual search to find the relevant email in a given thread. If a

user was working on a draft, they will have to navigate to a "Draft" folder, find the draft, and continue, which sometimes results in duplicated edits or even duplicated emails.

Third, consider online tax preparation programs such as TaxACT (taxact.com). To prepare a tax return under the United States tax code, taxpayers have to enter income and expense information, and answer numerous questions to determine their tax burden. Many people visit the tax site multiple times as documents arrive from their employers or financial institutions and need to be entered, or if answering a question requires off-line inquiries. If the user revisits the site, they expect to be able to continue at exactly the question where they left off. In currently available implementations, users are instead led to a top-level navigation point from which they must find out where to continue, and often have to repeat answering questions they already answered. Similar considerations apply to configuration management applications that are web-based and which may involve many tabs, dialogs, input fields, checkboxes or radio buttons, which are often conditionally displayed or hidden based on a user's history of navigating through the application.

These motivating applications share the following common characteristics.

- The user interface of these applications is rich, resembling that of a desktop application. Users prefer a single-page application style that gives them free reign in how to use the application. The navigation space is typically large.
- There appears to be little potential for offloading any application logic, or keeping application state in the client's browser, which is used purely as a display device to display a view of a model's state that is kept server-side. In applications involving multiple users, a lack of immediate and tight synchronization between the displayed view of the user interface and the application's model state may result in a degraded user experience.
- Users expect, or would prefer, if most or even all interactions with the user interface resulted in persistent effects independent of the device used to access the application, and independent of how often they navigate to the application.
- The number of users that access an instance of the application simultaneously is small when compared to the number of users a video distribution site such as YouTube would need to accommodate. This reduces the need for replication of an application's state and makes them "embarrassingly" horizontally scalable in the sense that any increase in the amount of dedicated resources yields a proportional increase in the number of application instances (with disjoint state) that can be supported.
- The frequency with which a user triggers relevant user interface events is limited by human processing speed and low when compared to, for instance, video game applications. For instance, such applications generally do not require tracking of mouse (move) events.

Current application frameworks make it difficult to maintain and synchronize an application's view on the server. The dominant model-view-controller (MVC) paradigm views the client-side user interface as the view component that is controlled by changes in a server-side model that is usually devoid of user interface state. Programmers must manually decide which, if any, user interface state they deem important enough to include in their models. When a user returns to a page, an initial view is constructed in response to the browser's HTTP request, which often involves constructing HTML using server-side templating. Any changes resulting from past interactions with the user interface must be incorporated manually into the templating logic, based on saved model state. Even when such state is kept, it is often associated with sessions, which are typically implemented using cookies that are not synchronized across different devices a user may use to access an application. Lastly, once a page displays to the user, programmers must decide which portions of the view they wish to keep in sync, using methods such as long polling or server push which typically take extra effort.

3. Developing CloudBrowser Applications

This section presents four examples that illustrate the paradigm in which CloudBrowser applications can be built. We start with an example that demonstrates the server-side execution and compatibility with existing HTML/JS applications, then discuss how to use observers, templating, and how to access shared data, and finally present a short example of how to implement the model-view-controller paradigm with a database-backed model.

3.1 Simple Document Example

Figure 1 shows an HTML document that contains a complete CloudBrowser application. This application contains two HTML text `<input>` elements and a `<span>` element whose text content depends on the input the user enters into the text input fields. This example uses exclusively the DOM Level 2 API [8] to manipulate DOM elements, and retrieve and set their attributes. It runs in any DOM-Level 2 compliant browser when fetched via a URL; in CloudBrowser, this code is run *server-side* in a virtual browser. This example demonstrates a key benefit: existing browsers can be used to test the user interface logic before deployment, because there is no user interface logic outside the HTML document describing an application!

Today, most JavaScript developers use higher-level libraries such as jQuery [30] rather than the DOM API directly. Figure 2 shows the JavaScript code of the same application, expressed more compactly using the appropriate jQuery selectors and event bindings. CloudBrowser executes

```
<html>
  <body style="font-family: Arial">
    First name: <input id='fname' type='text' />
    Last name: <input id='lname' type='text' />
    Hello <span id='output'></span>!
    <script>
      var fname = document.getElementById('fname'),
          lname = document.getElementById('lname'),
          output = document.getElementById('output');

      function onChange () {
        output.innerHTML = fname.value + ' '
                         + lname.value;
      }

      fname.addEventListener('change', onChange);
      lname.addEventListener('change', onChange);
    </script>
  </body>
</html>
```

Figure 1. A simple CloudBrowser application.

this application as well, *without* requiring any changes to the jQuery (v1.7.1) library.

```
$('#fname,#lname').change(function () {
  $('#output').text($('#fname').val()
            + ' ' + $('#lname').val());
});
```

Figure 2. CloudBrowser applications can use libraries such as jQuery for DOM manipulation like the ones used in Figure 1.

3.2 Meeting Times Example

As the next example, consider the collaborative meeting time application discussed in Section 2. We prototyped this application using CloudBrowser using a model-view-controller (MVC) approach [10]. Most MVC implementations use 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, the popular Knockout [31] client-side library gives a concise syntax for associating DOM elements with model data, provides for the automatic update of the UI when the model state changes through observers and observables [17], and provides templating facilities to generate DOM elements based on a model. 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. When running Knockout in CloudBrowser, the 'view model' and the actual 'model' be-

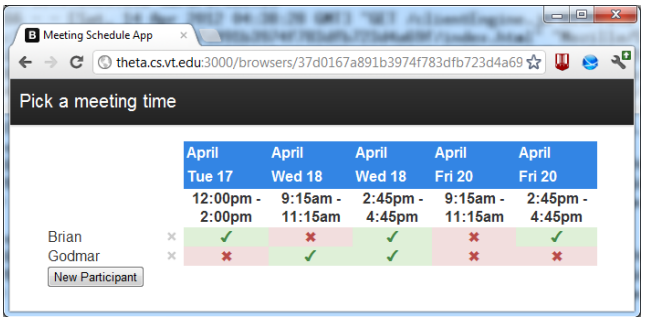

Figure 3. User interface of meeting time application.

```
<table>
  <thead>
    <tr><th width="25%"></th>
      <!-- ko foreach: times -->
      <th class="d-month container"
          data-bind="text: getMonth()"></th>
      <!-- /ko -->
    </tr>
    ...
  </thead>
  <tbody data-bind="foreach: participants">
    <tr class="participant-row">
      <td class="container">
        <a class="close" data-bind="
            visible: !editing(),
            click: $parent.removeParticipant">x</a>
        <span data-bind="
            visible: !editing(),
            click: function () {
               editing(true)
            },
            text: name"></span>
        <input data-bind="
            visible: editing,
            value: name,
            hasfocus: editing"></input>
      </td>
      <!-- ko foreach: available -->
      <td class="container" style="text-align: center"
          data-bind="
            text: $data.avail() ? '\u2714' : '\u2716',
            css: { 'alert-danger': !$data.avail(),
                   'alert-success': $data.avail()},
            click: function () {
                $data.avail(!$data.avail());
            }">
      </td>
      <!-- /ko -->
    </tr>
  </tbody>
</table>
<button data-bind="click: addParticipant">
    New Participant
</button>
```

Figure 4. Excerpts of the meeting time application.

```
function Participant(name, editing) {
  this.name = name;
  this.editing = ko.observable(editing);
  // ...
}

function Time(start, duration) {
  this.getMonth     = function ...
  // ...
}

var appModel = {
  times: ko.observableArray([
    // ...
  ]),
  participants: ko.observableArray(),
  addParticipant : function () {
    appModel.participants.push(
      new Participant('New Participant', true));
  },
  removeParticipant : function (participant) {
    appModel.participants.remove(participant);
  }
};
ko.applyBindings(appModel);
```

Figure 5. The view model for the meeting time application.

```
// if running in CloudBrowser, provide a
// JSON service to obtain current data
if (typeof require == "function") {
  var http  = require('http');
  http.createServer(function (req, res) {
    try {
      res.writeHead(200, {
                'Content-Type':
                'application/json'});
      res.end(ko.toJSON(appModel), 'utf-8');
    } catch (err) {
      res.writeHead(500);
      res.end('Server Error: ' + err);
    }
  }).listen(1337);
}
```

Figure 6. Exporting the view model as a JSON service.

come one and the same, eliminating the need for programming any client-server interaction.

Figure 4 shows relevant excerpts of the HTML describing this simple application, shown in Figure 3. An HTML table's headings are created from an array of possible meeting times, while its rows correspond to participants who have indicated their availability. Participants can toggle their availability, edit their names, or remove themselves from the schedule via mouse clicks. The set of participants and their availability is recorded in an underlying JavaScript object, which makes use of the observable pattern. For example,

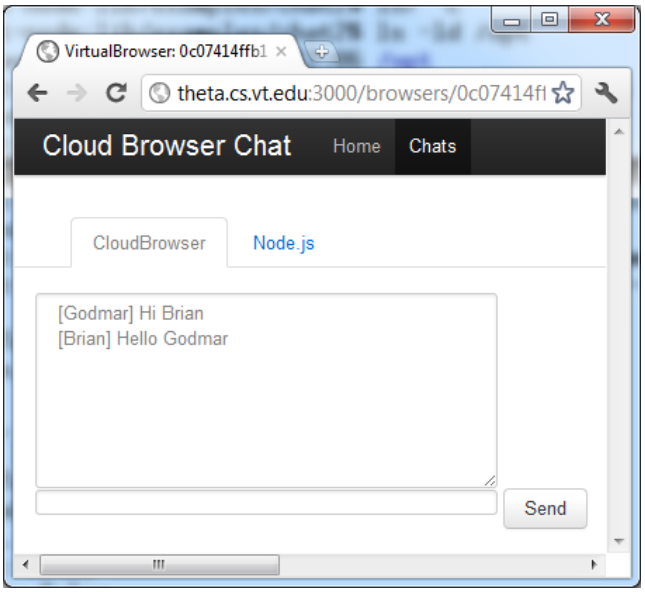

Figure 7. A screenshot of a simple chat room application.

clicking on a name trigger the corresponding 'click' data binding, which results in a transition to the editing state. In this state the `<input>` box is visible via the `visible` binding and its value is tied to the participant's name. The 'css' binding applies different styles (red vs. green) depending on the currently indicated availability.

Figure 5 displays excerpts of the associated JavaScript code. Knockout's `applyBindings` code ensures that any changes to the observable members of the `appModel` object are reflected in changes of the DOM. When run in CloudBrowser, multiple users can use the URL at which the application is available. Any changes made by any user are displayed in real-time to all connected users. Moreover, if a user closes their browser and later reconnects, even from a different browser using a different session, they will be able to rejoin this meeting application.

Like the example shown in the previous section, we were able to program and test the entire logic offline using a web browser within the confines of a JavaScript sandbox. Figure 6 shows how to add interaction with the outside world. Using node.js's `http` package, a simple HTTP server can be created that provides a JSON [12] service that exports the current results.

3.3 Chatroom Example

We prototyped a simple chatroom application, shown in Figure 7, to highlight two additional features of CloudBrowser. Like the meeting time application, the chatroom keeps track of model state (in this case, chat messages) in a JavaScript object and updates observables when it changes. Unlike in the previous example, multiple virtual browsers are instantiated since each user may be at a different navigation point. For instance, users may have joined different chat-

```
<div id="chat-tabs" data-bind='foreach: myChats'>
  <div class='tab-pane'
       data-bind="visible:
                    $root.activeRoom() === $data">
    <textarea rows='20' data-bind='value: messages'>
    </textarea>
  </div>
</div>
<input type='text' size='160'
       data-bind='value: currentMessage'>
</input
<button data-bind='click: postMessage'>Send</button>
```

Figure 8. An excerpt of chat room application view responsible for displaying chat messages. The 'foreach' binding duplicates its contained DOM nodes for each chat room that the user has joined. The 'visible' data binding ensures that only the currently selected chat room is shown. The 'messages' binding displays the chat messages for a room.

rooms. CloudBrowser allows the sharing of state across virtual browser instances, thus allowing observers in all instances to be notified when a new message appears in a chat room. An excerpt of the view expressed using Knockout bindings is shown in Figure 8. For instance, the "data-bind='value: messages'" attribute inside the `<textarea>` element ties the textarea's content to the array of chat messages.

The second key feature demonstrated by this example is the possibility for reusing higher-level UI libraries designed for client-side use. For instance, the Bootstrap/JS library [1] is used to implement a set of tabs (one per chatroom), which are declared using HTML `<ul>` and `<li>` elements. When the document is loaded, additional DOM elements and styles to represent actual tabs will be produced. The number of tabs, and their labels, are thus tied directly to the number of chatrooms that are accessible. The ability to run such widgets is crucial for creating a rich user experience.

3.4 Phonebook Example

Whereas the preceding chatroom 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 [15] Object Relational Mapping package to map JavaScript objects to a MySQL [2] database.

Figure 9 shows the HTML template that lists phone book entries and renders them in a table using Knockout's `text` data bindings. The phone book entry objects whose properties (e.g., 'fname', 'lname') are referred to in these bindings are constructed directly from the database. Figure 10 shows a screenshot of the resulting application.

The view model for the application is shown in Figure 11. Here, some glue is necessary to construct a view model suitable for use in Knockout, which wraps the sequelized phone

```
<table>
  <thead>
    <tr>
      <th>First Name</th>
      <th>Last Name</th>
      <th>Phone Number</th>
    </tr>
  </thead>
  <tbody data-bind="foreach: entries">
    <tr data-bind="click: $parent.rowClick">
      <td data-bind="text: fname" /></td>
      <td data-bind="text: lname" /></td>
      <td data-bind="text: phoneNumber" /></td>
    </tr>
  </tbody>
</table>
```

Figure 9. The 'foreach' data binding is used to map database entries to table rows.

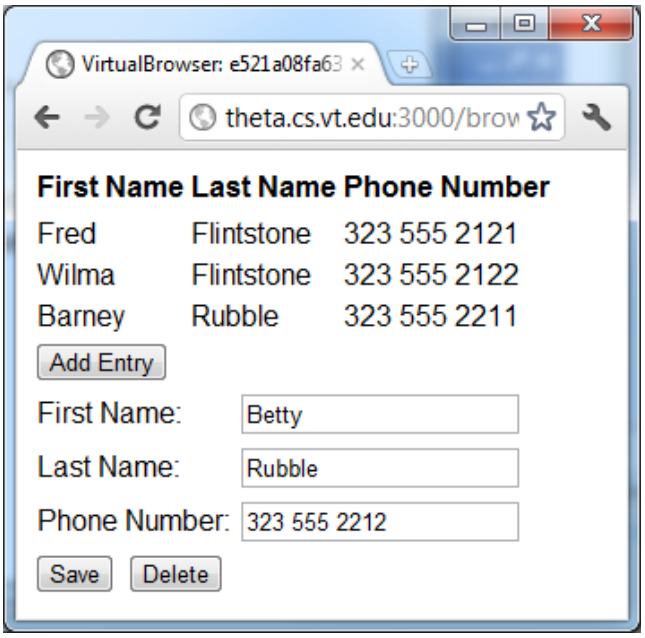

Figure 10. A simple database-backed phone book application.

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. Design and Implementation

We chose the open source Node.js [14] JavaScript execution environment for our prototype, for multiple reasons. First, Node.js is based on Google's V8 JavaScript en-

```
var vm = {
  entries      : ko.observableArray(entries),
  currentEntry : ko.observable(),
  rowClick     : function () {
    vm.currentEntry(this);
  },
  save : function () {
    this.save();    // persist to db
    vm.entries.remove(this);
    vm.entries.push(this);
    vm.currentEntry(null);
  },
  remove : function () {
    this.destroy(); // remove from db
    vm.entries.remove(this);
    vm.currentEntry(null);
  },
  create : function () {
    this.currentEntry(phoneBook.createEntry());
  }
};
```

Figure 11. View model used in phone book application.

gine, which represents the current state of the art with respect to execution performance. Second, Node.js's developer community provides many packages we use, such as the JSDOM JavaScript library [20] or the Sequelize [15] object-relational mapping, and many others which provide an environment that facilitates access to other application tiers. Third, Node.js provides a single-threaded environment whose semantics matches exactly the familiar execution semantics found in today's browser. Fourth, Node.js's event-based design provides for fast and efficient I/O, although it requires the programmer to rearrange (i.e., stack-rip [9]) their application to handle all I/O completion in asynchronously invoked callbacks.

Figure 12 shows an overview of CloudBrowser's design and components. The left half of the figure shows the client engine, which uses an update protocol to communicate with a virtual browser instance executing inside a server-side JavaScript virtual machine, shown on the right half of the figure. This section discusses the application life cycle, the client and server engine implementations, and the update protocol used by CloudBrowser applications.

4.1 Application Life Cycle

If a client sends an HTTP request to a configured application entry point (say `/example/index.html`), the CloudBrowser server starts a new virtual browser instance, creates an entry point for it (such as `/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.

When a virtual browser accepts a new client, it sends a small amount of HTML along with JavaScript code to bootstrap the client engine. The client engine then requests the current state of the 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. Clients can disconnect and reconnect while the application instance's state is preserved in the context of the virtual browser document.

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 1. Client Engine RPC Methods

Virtual browsers pose a resource management problem. Our current implementation provides an administrative module to list, inspect and terminate instances. When used in connection 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.

If virtual browser instances are not terminated manually, CloudBrowser supports an idle timeout after which a browser is terminated if no clients interact with it, allowing for automatic garbage collection. If necessary, a callback allows an application to save any transient state before terminating.

4.2 Client Engine

The client engine is responsible for receiving and rendering the initial state of the virtual browser's document, as well as for handling events on the client and receiving and rendering any updates to the document. We use a remote procedure call (RPC) abstraction to structure this communication. This RPC layer is implemented on top of a bidirectional message-based transport provided by the Socket.io [6] JavaScript library, which in turn encapsulates a number of underlying transport mechanism (web sockets for browsers that support them, or other AJAX-based long polling mechanisms such as Vault [34]).

At initialization, the client engine creates a connection to the server engine and establishes an RPC endpoint. This endpoint provides the methods shown in Table 1. The server

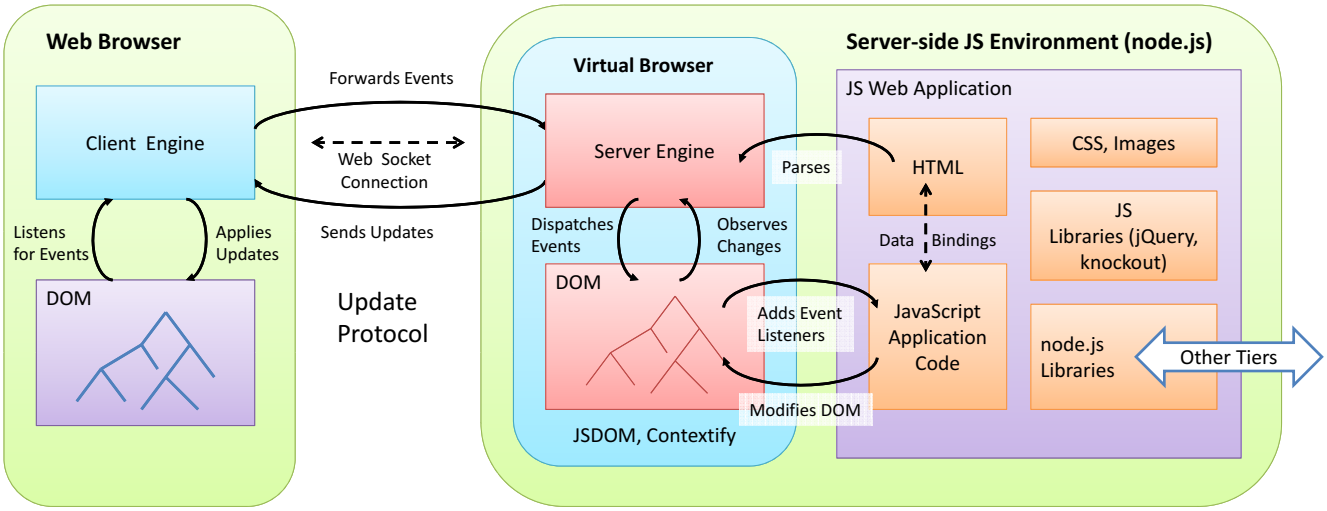

Figure 12. CloudBrowser Architecture Overview

processEvent(event)
setAttribute(target, attribute, value)

Table 2. Server Engine RPC Methods

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 3. DOM Node Record Format

engine then performs a pre-order traversal of the DOM tree, which results in a serialized representation of the DOM's nodes that is sent to the client engine in JSON format, using the 'PageLoaded' RPC method shown in Table 1. Each DOM node is assigned an id that can subsequently be used to refer to it. The client engine reconstructs the DOM by interpreting the received representation. Table 3 shows the record format used in this JSON representation.

The client engine must listen for and forward any client events to the server engine. 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. 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 the event 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 means that, for instance, clicks on links represented by `<a>` elements do not result in the user navigating away from the page. Instead, default actions are processed by the server engine and any resulting updates are propagated to the client.

Instead of having the client engine blindly listening 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 application 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.

4.3 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 [20].

The server engine processes forwarded events sent by the client engine(s) with which it maintains connections. Table 2 shows the RPC methods exposed by the server engine's en-

try 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 a `<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. The server engine uses aspect-oriented techniques [22] (e.g., advices associated with the DOM manipulation methods) to interpose on any changes to the server document. Consequently, unlike in server-centric frameworks such as ZK [11], 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 the client engine's `DOM*` methods shown in Table 1. 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 node and all its descendants.

An event listener will typically cause multiple updates to the server document. If these were sent to the client 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 3, we 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 web sockets currently do not support any.

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, 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. To achieve this goal, we implemented host methods such as 'setTimeout', 'setInterval', 'XMLHttpRequest' etc. using Node.js's and V8's facilities. We created a helper library, *Contextify*, which allows us to bind a JavaScript object whose properties contain the implementations of these methods to a V8 context such that these properties are visible in the global scope to JavaScript code executing in this context. This library ensures invariants upon which JavaScript libraries rely, such as the invariant that global variables appear as properties of the object to which `window` is bound, as well as corner cases such as the expression `window === this`, which must yield true within the global scope.

4.4 Styling and Layout

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 hierarchy, 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).

Unlike remote display systems such as Opera Mini [3] or SkyFire [5], CloudBrowser does not layout or render HTML components server side. We deemed it too expensive to re-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

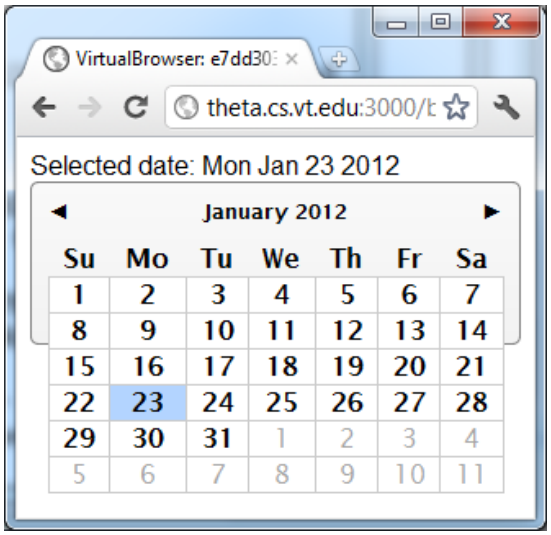

Figure 13. Encapsulating the YUI-3 DatePicker component in CloudBrowser

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.

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

4.5 Client-side Components

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) [7], 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. We provide client-side and server-side glue code for these 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

```
<script>
window.addEventListener('load', function () {
  var vm = { selectedDate : ko.observable() };
  ko.applyBindings(vm);

  var picker = cloudbrowser.createComponent(
      'calendar',
      document.getElementById('datePicker'),
      { // options
          height : '100px', width : '300px',
          showPrevMonth : true,
          showNextMonth : true
      });

  picker.addEventListener('Calendar:dateClick',
    function (e) {
      var date = new Date(e.info.date)
      vm.selectedDate(date.toDateString());
    });
});
</script>

<body style="font-family: Arial">
  <div>Selected date:
    <span data-bind='text: selectedDate'></span>
  </div>
  <div id='datePicker'></div>
</body>
```

Figure 14. Encapsulating the YUI-3 DatePicker component in CloudBrowser

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, YUI-3 Slider and DatePicker (calendar) components. Figures 13 and 14 show how such components can be integrated in a CloudBrowser application. The 'CloudBrowser.createComponent' API encapsulates access to the server-side glue for each supported components. We were able to encapsulate these components with rela-

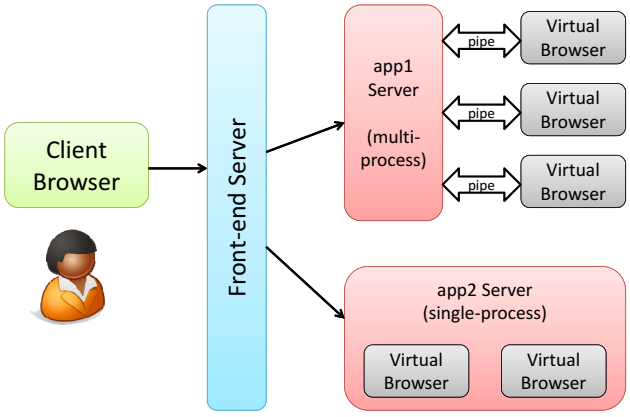

Figure 15. Load balancing across multiple CloudBrowser instances.

tively 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.

4.6 Multiprocess Implementation

Single-threaded, event-based servers such as Node.js have the potential for high performance [28, 29, 37], 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 15.

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 colocation is required only if an application shares JavaScript state across instances.

A front-end server hands off requests to a server that manages instances for each application. Our current implementation uses client-side redirection (via a 301 HTTP response), but other mechanisms such as passing of sockets 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, we are currently exploring how to extend the Socket.io library to support the handing off of an established web socket connection directly to the corresponding process.

5. 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.

5.1 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 Node.js 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 stylesheets and JavaScript code, which must be parsed and compiled by the V8 engine.

Using our x86_64 Node.js 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 3.3, which in addition includes the Bootstrap CSS stylesheets, consumes about 2.6MB per browser instance. We see 2 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 [13]. 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.

5.2 Latency

To measure latency, we ran a single-process CloudBrowser server on a server machine with 2 AMD Opteron 2380 2.5GHz quad-core processors and 16GB of RAM. Our simulated clients run within multiple processes (100 clients per

```
<html>
  <head></head>
  <body>
    <div id='target'></div>
    <script>
      var count = 0;
      var div = document.getElementById('target');
      div.addEventListener('click', function () {
        div.innerHTML = ++count;
      });
    </script>
  </body>
</html>
```

Figure 16. The example application used in the latency tests.

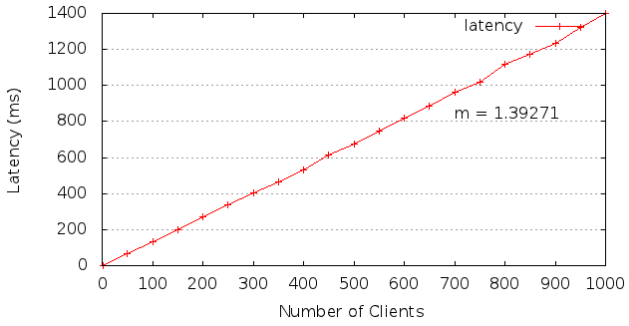

Figure 17. The average latency for all connected clients when running the application in Figure 16 with increasing numbers of clients. In this experiment, clients sent new requests immediately upon receiving a response to their previous request.

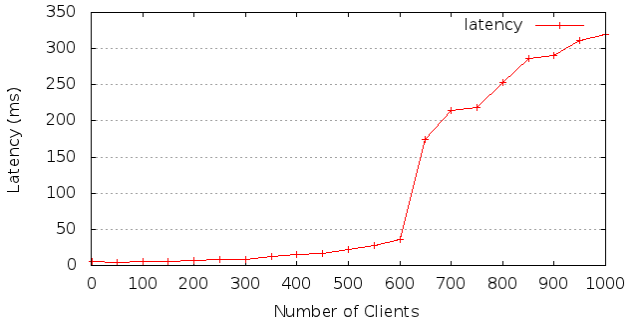

Figure 18. The average latency for all connected clients when running the application in Figure 16 with increasing numbers of clients. In this version of the experiment, clients paused for between 1 and 5 seconds using a uniformly random distribution before sending subsequent events.

process) on a separate machine with an Intel Q9650 quad-core 3.00GHz processor with 8GB of RAM. The 2 machines were connected via a Gigabit LAN.

The simulated clients each connect to the CloudBrowser server and request a new virtual browser instance. The CloudBrowser application used is shown in Figure 16. 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 17. 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 18. 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 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" [21].

A well-known result from usability engineering research [25] holds that response times of less than 100ms feel instantaneous to the user, and that response times between 100ms and 1 second, while noticable, 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 4.6, could further increase the number of supported clients without adding significant latency, especially when connections are handed off to separate processes.

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

Table 4. 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 5. Comparing CloudBrowser snapshots with the equivalent HTML that would be sent to a regular browser.

5.3 Bandwidth Consumption

Although not as important as latency, bandwidth consumption is an important performance indicator, particularly in 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 web socket transport mode for the client/server communication and used Google Chrome (v16) as our client browser.

Table 4 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.

Table 5 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 4.3, web sockets do not currently support GZip compression, although such an optimization is being considered [38]. GZip compression at the web socket 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. jQuery Test Suite Performance

5.4 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. 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.

6. Related Work

ZK [11] 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). ZUML components are translated into HTML and CSS when a page is rendered. A client-

side library handles synchronization between the client's view of and interaction with components, and their server-side representation. Our extensive experience deploying applications with ZK [16, 35] inspired the work on CloudBrowser. Compared to CloudBrowser, ZK does not maintain a representation of the server document across HTTP requests. Every visit to a ZK page creates a new ZK desktop, at which point the developer must use session-state information 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. 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.

ItsNat [32] 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. Unlike CloudBrowser, it also does not maintain the server document state across visits, and cannot make use of existing JavaScript libraries.

The Google Web Toolkit [19] allows the implementation of AJAX applications in Java that are compiled to JavaScript (or other targets). Like CloudBrowser, it provides an environment similar to that provided by desktop libraries, but focuses on the client-side only; communication with the server is outside its scope.

Fiz [26, 27] is a server-centric component-based AJAX framework that uses page properties to maintain component state on the server. Unlike in CloudBrowser, 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.

FlapJax [24] reduces the complexity of client-side JavaScript programming by introducing "event streams" and "behaviors" abstractions. Behaviors provide a data binding mechanism similar to Knockout.js's, and event streams provide a way to react to asynchronous events. The FlapJax primitives are intended to be used to simplify the client-side portion of an AJAX application, but the communication is still handled by the programmer. Since CloudBrowser applications can use existing client-side libraries, FlapJax could be used in conjunction with CloudBrowser to mitigate the error proneness of asynchronous programming; in fact, we were able to run a subset of the published Flapjax examples without changes.

Ripley [36] 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. Similar to CloudBrowser, client-side events are sent to the browser where they are dispatched into a server-side DOM. Unlike CloudBrowser, the events are also dispatched into the client-side DOM. 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.

Crawljax [23] 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 [4]. 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.

Opera Mini [3] 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.

Our system shares ideas with traditional thin-client and remote display systems, going back to "dumb terminals" based on the X Window System [33]. Compared to these systems, CloudBrowser is unique in that it uses a markup document and differential update to it to describe the structure and evolution of the user interface that is rendered to the user.

7. Conclusion

This paper presented CloudBrowser, a server-centric web application framework for the development of rich Internet applications that keep both their application and their presentation state on the server. 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. By providing existing client-side programming environments such as HTML documents and JavaScript virtual machines on the server, existing libraries and skill sets can be reused, and translation overhead is minimized. We have developed a prototype environment and applications, which indicate 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.

8. Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. CCF-0845830.

The source code for CloudBrowser is available at `https://github.com/brianmcd/cloudbrowser`. An application showcase is being created at `http://cloudbrowser.cs.vt.edu/`.

References

[1] Twitter Bootstrap JavaScript Demo. `http://twitter.github.com/bootstrap/javascript.html`.

[2] MySQL. `http://mysql.com`.

[3] Opera Mini Mobile Browser. `http://www.opera.com/mobile`.

[4] Selenium web browser automation. `http://seleniumhq.org`.

[5] SkyFire Mobile Browser. `http://www.skyfire.com/en/for-consumers/android/android`.

[6] Socket.io. `http://socket.io`.

[7] Yahoo! User Interface Library. `http://yuilibrary.com`.

[8] Document Object Model (DOM) Level 2 Events Specification. Technical report, Nov. 2000. URL `http://www.w3.org/TR/DOM-Level-2-Events/`.

[9] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur. Cooperative Task Management Without Manual Stack Management. In *Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference*, pages 289–302, Berkeley, CA, USA, 2002. USENIX Association.

[10] S. Burbeck. Application Programming in Smalltalk-80: How to use Model-View-Controller (MVC). Technical report, University of Illinois in Urbana-Champaign (UIUC).

[11] H. Chen and R. Cheng. *ZK: Ajax without the Javascript Framework*. Apress, Berkely, CA, USA, 2007.

[12] D. Crockford. Json. `http://www.json.org`.

[13] G. Czajkowski and L. Daynés. Multitasking without compromise: a virtual machine evolution. *SIGPLAN Not.*, 36(11): 125–138, Oct. 2001. doi: 10.1145/504311.504292.

[14] R. Dahl. Node.js. `http://nodejs.org`.

[15] S. Depold. Sequelize. `http://sequelizejs.com`.

[16] S. H. Edwards and G. Back. Bringing creative web 2.0 programming into CS1: conference workshop. *J. Comput. Sci. Coll.*, 26(3):54–55, Jan. 2011.

[17] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

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

[19] Google, Inc. Google web toolkit (gwt). `http://code.google.com/webtoolkit/`.

[20] E. Insua. JSDOM. `http://jsdom.org`.

[21] Keynote Systems, Inc. Internet health report. `http://www.internetpulse.net/`.

[22] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP'97 Object-Oriented Programming*, volume 1241, pages 220–242. Springer Berlin / Heidelberg, Berlin/Heidelberg, 1997. doi: 10.1007/BFb0053381.

[23] A. Mesbah, A. van Deursen, and S. Lenselink. Crawling Ajax-Based Web Applications through Dynamic Analysis of User Interface State Changes. *ACM Trans. Web*, 6(1), Mar. 2012. doi: 10.1145/2109205.2109208.

[24] L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi. Flapjax: a programming language for Ajax applications. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, pages 1–20. ACM, 2009. doi: 10.1145/1640089.1640091.

[25] J. Nielsen. *Usability Engineering*. Morgan Kaufmann, 1st edition, Sept. 1993. ISBN 9780125184069.

[26] J. Ousterhout. Fiz: A component framework for web applications. Technical report, Dep. of CS, Stanford University, 2009.

[27] J. Ousterhout and E. Stratmann. Managing state for Ajax-driven web components. In *Proceedings of the 2010 USENIX conference on Web application development*, WebApps'10, page 7, Berkeley, CA, USA, 2010. USENIX Association.

[28] D. Pariag, T. Brecht, A. Harji, P. Buhr, A. Shukla, and D. R. Cheriton. Comparing the performance of web server architectures. *SIGOPS Oper. Syst. Rev.*, 41(3):231–243, Mar. 2007. doi: 10.1145/1272998.1273021.

[29] W. Reese. Nginx: the high-performance web server and reverse proxy. *Linux J.*, 2008(173), Sept. 2008.

[30] J. Resig. jQuery. `http://jquery.com`.

[31] S. Sanderson. Knockout. `http://knockoutjs.com`.

[32] J. M. A. Santamaria. ItsNat: Natural AJAX. component based Java web application framework. `http://itsnat.sourceforge.net`.

[33] R. W. Scheifler and J. Gettys. The X window system. *ACM Trans. Graph.*, 5(2):79–109, Apr. 1986. doi: 10.1145/22949.24053.

[34] E. Stratmann, J. Ousterhout, and S. Madan. Integrating long polling with an MVC framework. In *Proceedings of the 2nd USENIX conference on Web application development*, WebApps'11, page 10. USENIX Association, 2011.

[35] E. Tilevich and G. 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, New York, NY, USA, 2008. ACM. doi: 10.1145/1353482.1353485.

[36] K. Vikram, A. Prateek, and B. 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, New York, NY, USA, 2009. ACM. doi: 10.1145/1653662.1653685.

[37] M. Welsh, D. Culler, and E. Brewer. SEDA: an architecture for well-conditioned, scalable internet services. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, volume 35 of *SOSP '01*, pages 230–243, New York, NY, USA, Dec. 2001. ACM. doi: 10.1145/502034.502057.

[38] T. Yoshino. WebSocket Per-frame DEFLATE Extension. Technical Report draft-tyoshino-hybi-websocket-perframe-deflate-04.txt, IETF Secretariat, Fremont, CA, USA, Aug. 2011. URL `http://www.rfc-editor.org/internet-drafts/draft-tyoshino-hybi-websocket-perframe-deflate-04.txt`.