Rich Cloud-based Web Applications with CloudBrowser 2.0

Xiaozhong Pan

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
Eli Tilevich
Ali R. Butt

April 28, 2015
Blacksburg, Virginia

Keywords: Web Frameworks, Javascript, Node.js, Distributed Systems

Rich Cloud-based Web Applications with CloudBrowser 2.0

Xiaozhong Pan

(ABSTRACT)

When developing web applications using traditional methods, developers need to partition the application logic between client side and server side, then implement these two parts separately (often using two different programming languages) and write the communication code to synchronize the application's state between the two parts. CloudBrowser is a server-centric web framework that eliminates this need for partitioning applications entirely. In CloudBrowser, the application code is executed in server side virtual browsers which preserve the application's presentation state. The client web browsers act like rendering devices, fetching and rendering the presentation state from the virtual browsers. The client-server communication and user interface rendering is implemented by the framework under the hood. CloudBrowser applications are developed in a way similar to regular web pages, using no more than HTML, CSS and JavaScript. Since the user interface state is preserved, the framework also provides a continuous experience for users who can disconnect from the application at any time and reconnect to pick up at where they left off.

The original implementation of CloudBrowser was single-threaded and supported deployment on only one process. We implemented CloudBrowser 2.0, a multi-process implementation of CloudBrowser. CloudBrowser 2.0 can be deployed on a cluster of servers as well as a single multi-core server. It distributes the virtual browsers to multiple processes and dispatches client requests to the associated virtual browsers. CloudBrowser 2.0 also refines the CloudBrowser application deployment model to make the framework a PaaS platform. The developers can develop and deploy different types of applications and the platform will automatically scale them to multiple servers.

Acknowledgments

I am very fortunate to have Dr. Godmar Back as my advisor. I would like to thank him for his patience and guidance. I am very grateful for the time and effort he put in our weekly one-to-one meetings. These meetings are at least equivalent to three graduate courses.

I would also like to thank Dr. Ali R. Butt and Dr. Eli Tilevich for serving on my committee and providing valuable feedbacks.

I would also like to thank my fellow graduate students in computer science department, it is a honor to spend two years with you wonderful guys.

Finally, I would like to thank National Science Foundation. This thesis is based upon work supported by the National Science Foundation under Grant No. CCF-0845830.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Most newly developed applications that provide a user interface to end users are web-based. Modern browsers provide powerful and expressive user interface elements, allowing for rich applications, and the use of a networked platform simplifies the distribution of these applications. As a result, researchers and practitioners alike have devoted a great deal of attention to how to architect frameworks on this platform, which is characterized by the use of the stateless HTTP protocol to transfer HTML-based user interface descriptions to the client along with JavaScript code, which in turn implements interactivity and communication with the application's backend tiers.

In many recently developed frameworks, much of the presentation logic of these applications executes within the client's browser. User input triggers events which result in information being sent to a server and subsequent user interface updates. Such updates are implemented as modifications to an in-memory tree-based representation of the UI (the so-called document object model, or DOM), which is then rendered by the browser's layout and rendering engine so the user can see it. However, the state of the DOM is ephemeral in this model:

when a user visits the same application later from the same or another device, or simply reloads the page, the state of the DOM must be recreated from scratch. In most existing applications, this reconstruction is done in an rudimentary and incomplete way, because application programmers typically store only application state, and little or no presentation state in a manner that persists across visits. As a result, many web applications do not truly feel like persistent, "in-cloud" applications to which a user can connect and disconnect at will. By contrast, users are accustomed to features such as Apple's Continuity [1] that allows them to switch between devices while preserving not only essential data, but enough of the applications' view to create the appearance of seamlessly picking up from where they left off.

CloudBrowser [26] is a server-centric web framework that keeps the state of the HTML document in memory on the server in a way that is persistent across visits. In this model, presentation state is kept in virtual browsers whose life cycle is decoupled from the user's connection state. When a user is connected, a client engine mirrors the state of the virtual browser in the actual browser which renders the user interface the user is looking at. Any events triggered by the user are sent to the virtual browser, dispatched there, and any updates are reflected in the client's mirror. This idea is reminiscent of "thin client" designs used in cloud-based virtual desktop offerings, but with the key difference that in this proposed design the presentation state that is kept in a virtual browser is restricted to what can be represented at the abstract DOM level; no flow layout or rendering is performed by the virtual browser on the server.

This model entails additional potential benefits: since only framework code runs in the client engine, the application code running on the server does not need to handle any client/server communication and can be written in an event-based style similar to that used by desktop user interface frameworks. Since the virtual browser has the same JavaScript execution capabilities as a standard browser, emerging model-view-controller (MVC) frameworks such as AngularJS [21] can be directly used, further simplifying application development. More side benefits include: a lighter weight client engine that can load faster, a resulting application

that is potentially more secure since no direct access to application data needs to be exposed, the ability to co-browse by broadcasting the virtual browser state to multiple clients.

1.2 Core Contributions

Prior to this work, the implementation of CloudBrowser was single threaded and supported deployment on only one process. It could not scale horizontally by adding more processors or more machines to increase the system's capacity. To enable CloudBrowser to host large scale web applications, we designed and implemented CloudBrowser 2.0 which can distribute virtual browsers across the CPU cores of a multiprocessor machine or across a cluster of machines.

We designed a single-master, multiple-workers architecture: the master is responsible for application management and load balancing, the workers host virtual browsers and serve user requests. To facilitate the inter-process communication among the processes in our system, we developed a remote procedure call framework nodermi [34] that encapsulates message communication as method calls. Nodermi transparently creates stubs that represent objects in other processes and allows a process to invoke methods of other processes' objects by calling methods on the stubs. We also developed an application programming interface that fully isolates application code from each other and from underlying systems code.

The underlying architecture change is transparent to applications. The new implementation fully preserves the semantics of the existing programming model while provide higher scalability. CloudBrowser applications do not need to be aware of the distributed architecture on which they run. Most of the existing applications can automatically scale to multiple processors without modification. Some applications needed to be modified because of necessary changes to API methods' signatures.

We have implemented a number of sample applications and profiled them to better understand the intrinsic and extrinsic limitations of this design. We also built a benchmark tool

that simulates multiple users interacting with the applications and used it to evaluate the performance and scalability of CloudBrowser 2.0. In our experiments, CloudBrowser 2.0 scales linearly, it supports 2,800 concurrent users using a chat room application on a eight core machine with average latency under 100ms.

Chapter 2

Background

This chapter provides background information to understand the concepts underlying the design of CloudBrowser 2.0. We assume the reader has basic knowledge of web-related protocols and languages, including HTTP [15], HTML [22], DOM [46], JavaScript [12] and CSS [5].

2.1 Scalable Web Server Architectures

CloudBrowser 2.0 aims to provide a scalable platform for web applications. In this section we introduce some typical methods to design scalable architectures for web applications.

Web servers provide the infrastructure on which web applications are hosted. Scalability is an important issue for web servers as the number of users visiting a site can grow significantly even during short time spans [2]. Even without an increase in users, web applications may demand more resources as they become more sophisticated.

Figure 2.1 shows a diagram of a typical scalable web server architecture, which is able to harness resources from a cluster of servers. We divide the servers in such a system into two layers. The web layer processes HTTP requests from users and generates HTTP responses

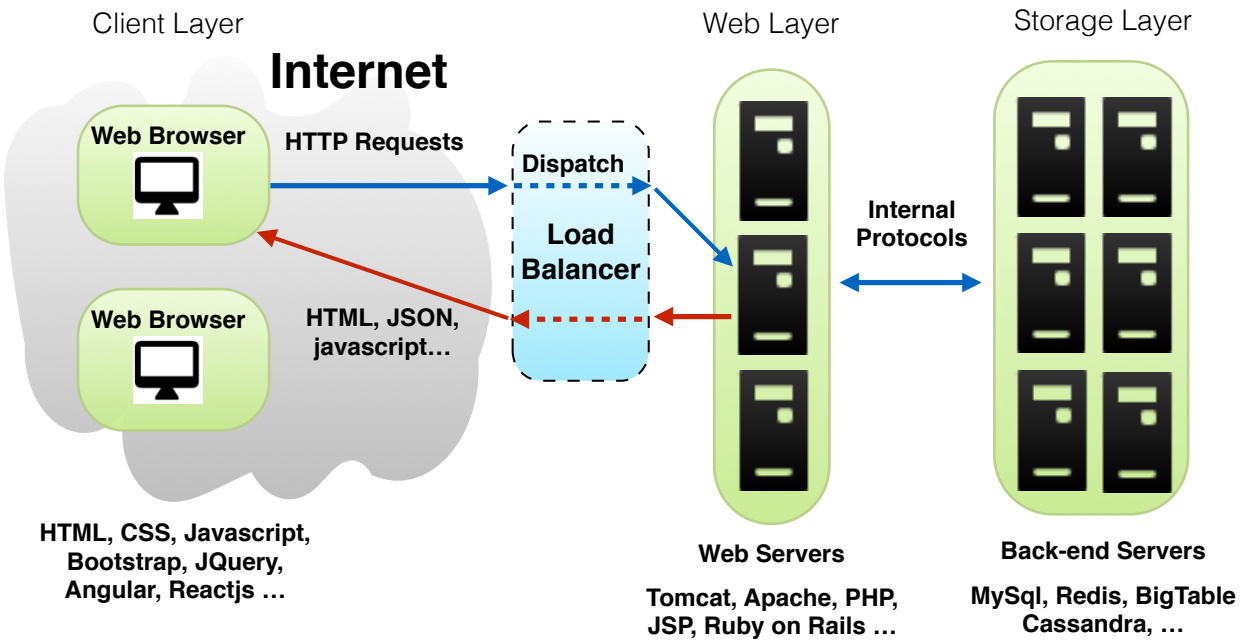

Figure 2.1: Scalable web server architecture

which will be rendered in the user's browser. The web layer offloads storage and computational tasks to a storage layer that consists of database servers or other types of storage servers. The benefit of such layered design is twofold: it makes the web servers lightweight so they are able to handle more concurrent connections; the whole system becomes modular, so that the system operator can upgrade parts of the system without interfering with other parts.

To balance requests evenly across the web servers, a load balancer component sits between the clients and the web layer, which dispatches user requests to web servers and returns the responses from the web servers to users. The use of a load balancer allows users to access the web server using a single URL because it hides the distributed architecture from the user. There are many ways to implement load balancing [7]. For example, one can use round-robin DNS [6] to distribute the load to a set of servers using distinct IP addresses. The DNS server will rotate the list of addresses returned, causing clients to connect to different machines. DNS- based load balancing has a number of drawbacks: first, it is difficult to add or remove

web servers since clients will cache DNS information. Second, it requires that all servers have public IP addresses so that they are directly reachable by clients.

To overcome the limitations of DNS-based load balancing, system architects often add a layer of reverse proxies between the clients and the web servers. Reverse proxies (such as nginx [38]) forward user requests to one or multiple web servers and copy responses from the web servers back to the users.

It is possible to have multiple reverse proxies and use DNS-based load balancing among them. User requests are first distributed among the reverse proxies, then the reverse proxies distribute user requests across a larger number of web servers.

2.1.1 Session State Management

Web applications use session state to remember client-specific information across HTTP requests so they can provide stateful services on top of the stateless HTTP protocol. Web applications typically assign a unique session id for each user and use this session id to retrieve the user's session state for each request. For the first request, the server automatically creates a new session for the user. Applications store various bits of information in a user's session state, such as authentication information, user profile, shopping carts, etc.

Most web frameworks provide two modes to manage session state [35, 24]: a local mode where session state is stored in memory or in the file system and a distributed mode where session state is stored in a storage system, often a relational database.

If session state is stored locally, it can be accessed faster, but it is not accessible from other web server instances. To overcome this limitation, some web servers such as Tomcat support a session replication mode [19] that synchronizes locally stored session state across a cluster of web servers. Since a web server needs to broadcast any changes to session state to all other web servers, this mode is expensive when there are many web servers.

Large scale web applications usually adopt distributed session state management. Besides

traditional relational databases, the backend session storage could be implemented by high performance key value stores such as Redis [37] or distributed data stores such as Memcached [16], or Cassandra [25].

2.1.2 Load Balancing

The design choices regarding load balancing and their implementation can affect application development. We use the taxonomy from the survey of Cardellini et al. [7] to categorize load balancing approaches into client-blind or client-aware, based on whether the load balancer uses information from the client's request to perform the dispatch.

A client-blind load balancer does not use information from the client when deciding how to dispatch a request. For example, it could randomly select a web server from the web layer for each request, or use a round robin algorithm. In this case, multiple requests from one client could be relayed to different web servers. In this design, session state cannot be stored locally, rather it must be accessible from any server handling the client's request. For every request, the web server needs to fetch the session state from session storage and pushes the changes back to session storage if session state is modified.

A client-aware load balancer dispatches every request from a given client to the same web server. In this model, the web server can use a local mode session state implementation, which is faster, but client-aware load balancers can also benefit distributed session state implementations. Since a given client's requests are dispatched to the same server, the web server can cache session data to get better performance. In this way, a web server retrieves session state once for every user and communicates with the storage system only when session data changes. The downside of the client-aware load balancing approach is that the load balancer needs to identify each client's assigned server to make the correct routing decision. To that end, the load balancer needs to examine the HTTP request header to extract the session identifier and assign web servers based on a hash of this identifier.

Because different requests can impose different load on a web server, purely random or round-robin based approaches can cause uneven load distribution among web servers. Especially for modern web applications with long connections, the time a connection keeps open can vary from milliseconds to hours. A naive load balancer could keep distributing requests to an already busy server even when there are less loaded servers in the cluster. The load balancer can also be server-aware such that it takes servers' load information into consideration when making dispatching decisions. Because client-aware load balancing has the restriction of keeping requests from a given user to the same server, server- and client-aware load balancing is more complex than server-aware and client-blind load balancing.

2.2 Node.js

CloudBrowser is built on top of the Node.js framework. Node.js [43] is a standalone JavaScript runtime built on Google Chrome's V8 JavaScript engine [44] and it allows JavaScript code to be executed without a web browser. Node.js comes with a standard library that provides API for file system access, network I/O, binary data manipulation, and others. It enables developers to write server-side network applications in JavaScript. Node.js is appealing for building high performance scalable web applications because it adopts a non-blocking, event-driven I/O model which is capable of handling a large number of concurrent connections using a single thread.

Node.js has a thriving community with a variety of third party packages. We have relied on multiple packages in CloudBrowser to implement complex tasks such as server side DOM manipulation as well as for many utility or formatting functions. Another reason we chose Node.js is that we are able to write the CloudBrowser infrastructure in the same language as its applications, avoiding costs associated with crossing languages.

In this section we discuss Node.js's non-blocking I/O model and some important third party packages we used.

2.2.1 Non-blocking I/O

A JavaScript engine has only one thread that executes JavaScript code. In this model, if a function blocks on some operation such as I/O, then whole process blocks and cannot make progress. By contrast, in a multi-threaded languages such as Java only the current thread is blocked whereas other threads within the same process can still make progress. To avoid blocking and achieve concurrent execution of multiple tasks, JavaScript uses a non-blocking I/O model based on an event queue. In this model, I/O functions return immediately after issuing I/O requests. The runtime processes I/O operations in the background and keeps track of when they complete. Upon completion, an event is scheduled that, when executed, will fire a programmer-provided callback function. In this way, the program can concurrently performing multiple I/O tasks while executing JavaScript code.

The execution thread polls the event queue in the event loop, processing events in order. The handlers/callback functions associated with each event must run to completion. This lack of preemption means that JavaScript code is immune from the race conditions that arise in preemptive, multi-threaded environments, and makes it simpler to reason about the behavior of individual functions.

```
1  var fs = require('fs');
2  fs.readFile('/etc/passwd', function (err, data) {
3    if (err) throw err;
4    console.log(data);
5  });
6  fs.readFile('/etc/hosts', function (err, data) {
7    if (err) throw err;
8    console.log(data);
9  });
10 console.log("Reading file");
```

Listing 1: Reading a file and printing its content using Node.js.

Listing 1 shows an example of the non-blocking I/O concept in Node.js. Because the I/O API such as `readFile` does not block, the developer needs to pass a callback function to the

`readFile` method that is invoked asynchronously when the I/O operation has completed.

Although event handlers run to completion, the order in which events are fired is unpredictable. As a result, it can be hard to reason about a program's execution order. Asynchronous callbacks may be executed in a order that is different from the order in which they were registered. For example, in Listing 1 the program called `readFile` twice but it is possible that the callback for the second `readFile` call is executed first. It is also impossible to tell from the function signature alone if a callback will be called asynchronously via the event queue mechanism or if it could be called synchronously. In Listing 1 line 10 is executed before line 3 if `readFile`'s callback fires asynchronously. If, in the future `readFile` were changed such that it caches the contents of files, it may call the callback function synchronously if it finds cached data in a request, so line 3 will be executed before line 10. Thus the programmer must take extra caution to not depend on a certain execution order.

This asynchronous execution model requires the programmer to preserve the application context in the callback functions' closure. This process is prone to subtle mistakes. For example, suppose we want to read a list of files and print out each file's name and content. Listing 2 lines 4–12 shows an incorrect solution that iterates over an array of file names and calls `readFile` with a callback that prints the current file name and file content. If the callback fires synchronously, filename will refer to the currently read file. If, however, the callback fires asynchronously - as it likely will in the current implementation of `readFile`, filename will refer to the name of the last file in the list. Line 14–23 shows a correct implementation that dynamically creates a function that references the current file name in its closure. Though correct, the resulting code is more difficult to read.

The prevalence of asynchronous callbacks in JavaScript is not intuitive for programmers that are used to languages like Java or C that provide blocking I/O primitives. When writing applications with complex logic, programmers often end up with deeply nested callbacks which make their code hard to understand, something which has been referred to by the

```
1  var fs = require('fs');
2  var files = ['/etc/passwd', '/etc/hosts'];
3  // incorrect implementation
4  for (var i = 0; i < files.length; i++) {
5      var fileName = files[i];
6      fs.readFile(fileName, function(err, data){
7          if (err) throw err;
8          // this will always print "content of /etc/hosts is ...",
9          // fileName is assigned with the last element of files
10         console.log("content of " + fileName + " is "+ data);
11     });
12 }
13 // correct implementation
14 var createReadFileCallback = function(fileName){
15     return function(err, data){
16         if (err) throw err;
17         console.log("content of " + fileName + " is "+ data);
18     }
19 };
20 for (var i = 0; i < files.length; i++) {
21     var fileName = files[i];
22     fs.readFile(fileName, createReadFileCallback(fileName));
23 }
```

Listing 2: Reading two files in a loop using Node.js.

developer community as *callback hell*.

For instance, to read two files in order, the program in Listing 2 could be changed such that the I/O for the second file is initiated in the callback that is fired when the first file has been read, which adds a level of nesting. However, this approach will not allow the I/O of those files to overlap. To handle this situation and more complex scenarios, developers frequently use libraries like async [27].

2.2.2 Node.js Packages Used in CloudBrowser

Async

Async is a library that helps developers implement complex control flow patterns of asynchronous functions without nested callbacks. It supports various recurring patterns, including execution in series, in parallel, or in an order that meets predefined dependencies, and others. The developers can focus on the implementation of individual steps and *async* will execute them in the right order, while simultaneously documenting the desired semantics.

For example, to compute the combined size of multiple files, it is necessary to read these files in parallel, add each file's size to a running total, and then output the total sum after every file is read. Listing 3 shows two implementations, lines 5–13 implement a barrier-style counter in vanilla JavaScript, whereas lines 14–21 use async's `each` function, which provides parallel iteration over an array. Although this is a simple use case, the use of *async* makes the intended control flow pattern more clear and includes proper exception handling.

Node-http-proxy

In CloudBrowser 2.0, we needed to implement a reverse proxy that allows us to specify destination servers for incoming HTTP requests. Furthermore, we required the ability to add and remove forwarding rules programmatically. Existing standalone reverse proxy software such as nginx rely on pre-defined URL matching rules and support only fixed dispatch policies such as round-robin when selecting destination servers. We found this software too difficult to adapt to achieve the required behavior.

Node-http-proxy [31] is a Node.js package that helps developers to implement customizable reverse proxy software. It allows the developer to write customized logic to implement specific forwarding policies for incoming HTTP requests and provides the necessary code to handle the network communication for proxying these requests.

```
var fs = require('fs');
var async = require('async');
var files = ['/etc/passwd', '/etc/hosts', '/etc/profile'];
var finished = 0, totalSize = 0, totalSize2 = 0;
for (var i = 0; i < files.length; i++) {
    fs.readFile(files[i], function(err, data){
        totalSize += data.length;
        finished++;
        if (finished == files.length) {
            console.log("total file size "+ totalSize);
        };
    });
};
async.each(files, function(fileName, done){
    fs.readFile(fileName, function(err, data){
        totalSize2 += data.length;
        done();
    });
}, function(err){
    console.log("total file size "+ totalSize2);
});
```

Listing 3: Print the total size of multiple files using async.

It also supports the WebSocket [14] protocol. WebSocket is a TCP-based protocol which provides a bidirectional, full-duplex communication mechanism between web browsers and servers. Using WebSockets, servers can send information to clients without requiring a previous request by the client, unlike in traditional HTTP. It also avoids the overhead of repeatedly establishing connections. We use WebSockets to implement DOM synchronization between clients and server. In node-http-proxy, the programmer needs to provide the destination server only for the hand shake request that establishes the connection, then all subsequent WebSocket messages will be transparently proxied to the destination server. This feature greatly simplified our implementation.

Jsdom

Jsdom [23] is a Node.js package which provides a JavaScript implementation of the DOM API. Jsdom works like a web browser except it does not render the DOM tree: it reads HTML documents to build the initial DOM tree and executes JavaScript files specified in script tags. It is possible to create multiple jsdom instances in the same process, each of these jsdom instance will have its own DOM tree and isolated namespace for its JavaScript code.

2.3 CloudBrowser

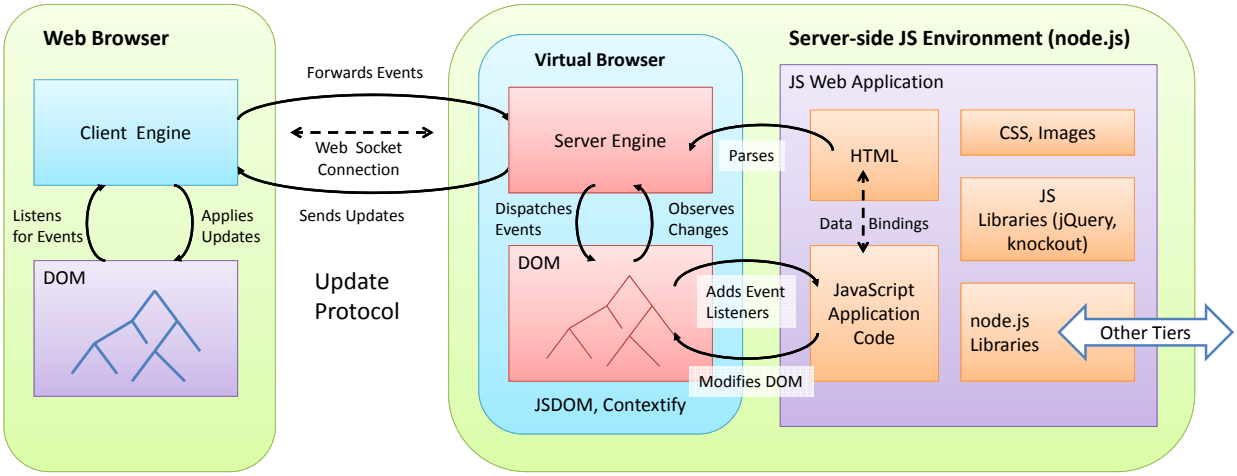

Figure 2.2: Single Process CloudBrowser Architecture Overview

This section reviews the implementation of the original, single-process version CloudBrowser, which formed the basis for this work [26]. Figure 2.2 shows the relationship between the client engine running in the user's browser and the virtual browser (implemented using jsdom) running server side. When the user visits an application, the client engine code is downloaded, which then restores the current view of the application by copying the current state of the server document. Subsequently, user input is captured, forwarded to the server engine inside the virtual browser, which then dispatches events to the document. All application logic

runs in the global scope associated with the virtual browser's window object. Since the server environment faithfully mimics a real browser, libraries such as AngularJS [21] can be used unchanged to implement the user interface. Client and server communicate through a lightweight RPC protocol that is layered on top of a bidirectional WebSocket communication channel. Stylesheets, images, etc. are provided to the client through a resource proxy. The client's browser is responsible for rendering the document on the screen using the provided styles.

2.3.1 Deployment Model

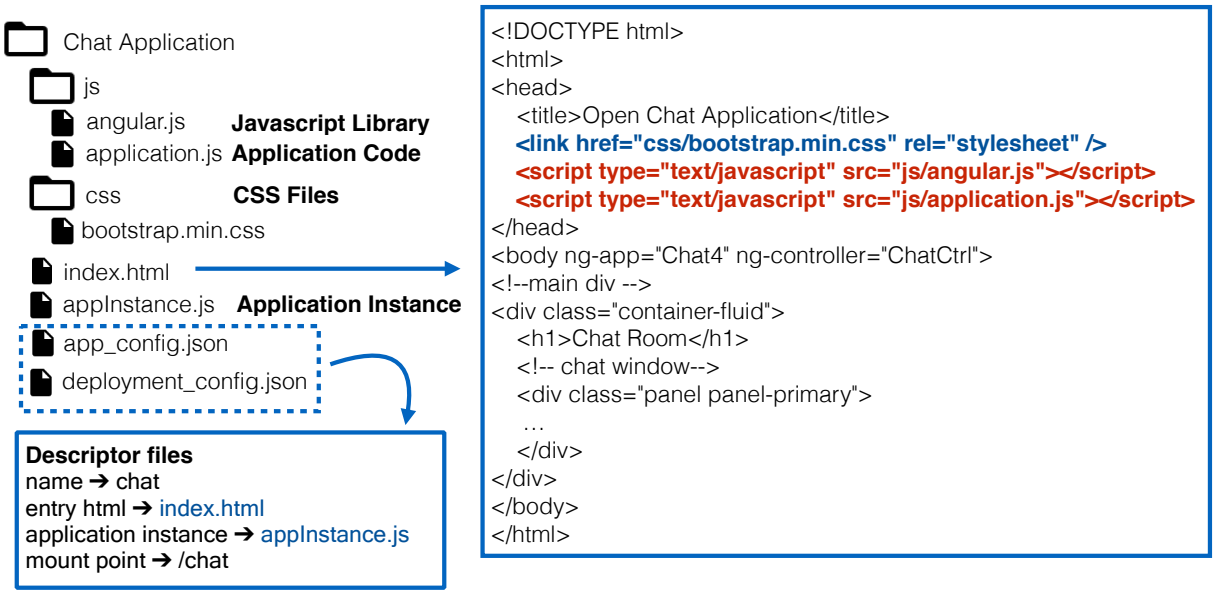

Figure 2.3: Folder structure of a CloudBrowser application

CloudBrowser uses an application server model. Application code is packaged in bundles in a well-defined format. Developers create application bundles, which can be deployed from CloudBrowser's application directory or uploaded using CloudBrowser's administrative interface. Multiple applications can be deployed simultaneously.

Figure 2.3 shows an example of an CloudBrowser application bundle contained in a directory.

The bundle includes descriptor files, an entry point HTML file, JavaScript files and resource files such as CSS files, images, etc. The descriptor files specify the application's name, owner, mount point, and other application configuration information. The mount point is the URL path to the application, for example, if a CloudBrowser is deployed at `example.com` and an application's mount point is *chat*, the user could access the application via `http://example.com/chat`. The JavaScript files include libraries like AngularJS or JQuery, the application code and an optional *application instance definition* file (detailed later). Like in regular browser, the entry point HTML can have script tags specifying the JavaScript files the developer wants to be executed.

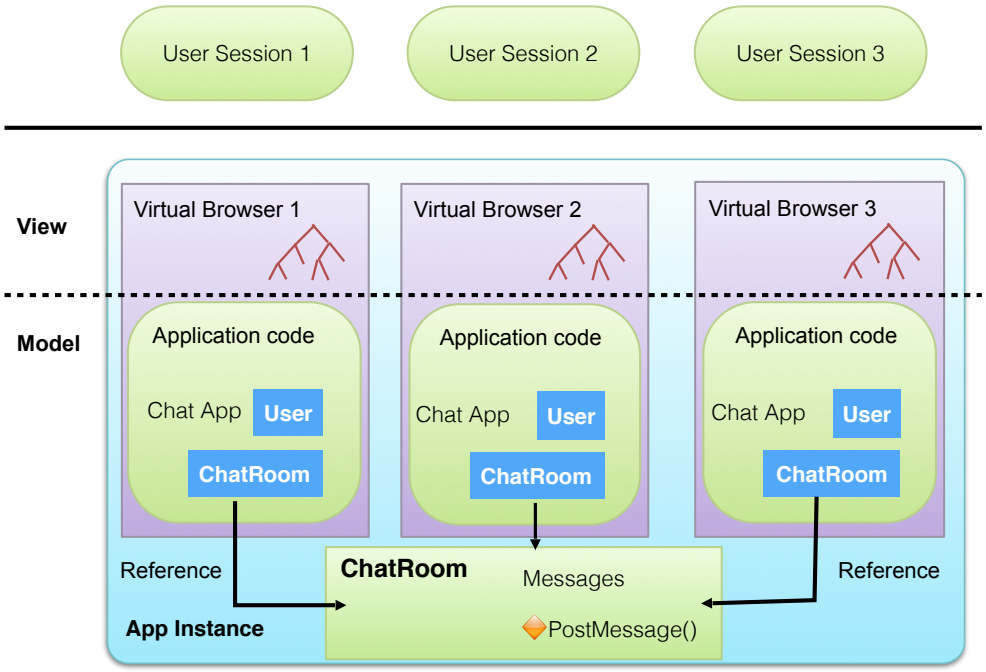

Figure 2.4: Sharing data among multiple virtual browser via application instance. This figure shows how multiple virtual browsers can directly and seamlessly share relevant application data, in this case chat messages, which then become part of the model that drives the presentation MVC framework.

Application execution in CloudBrowser is organized via *application instances* and *virtual browsers*. We motivate these abstractions using the following use case scenario. Consider

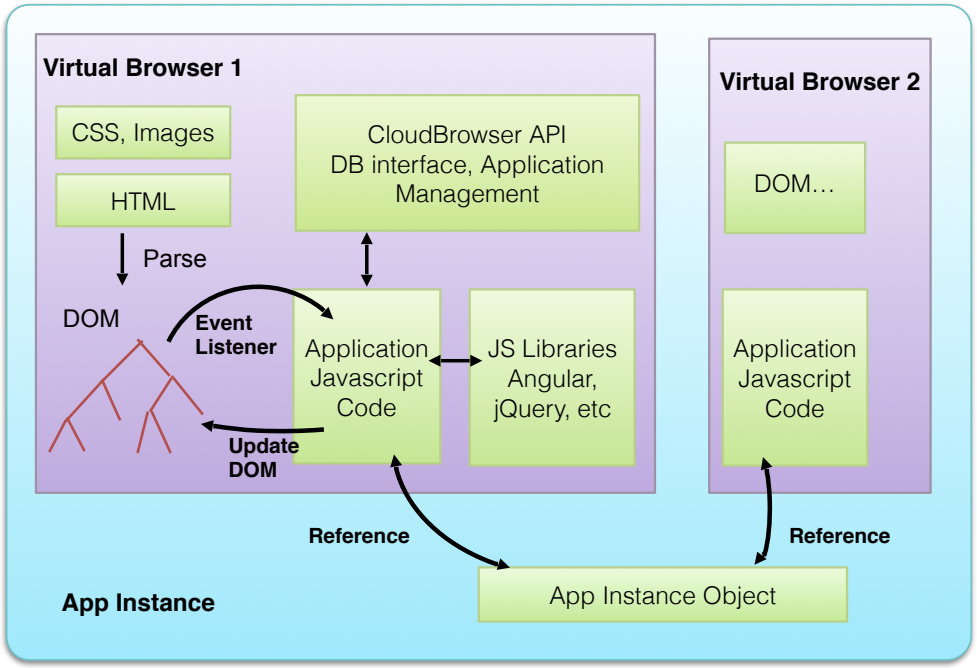

Figure 2.5: Application Instance

a Chat application developed using AngularJS, as depicted in Figure 2.4. After a CloudBrowser administrator installs this application, authorized users will have the ability to create chatrooms and join existing chatrooms. Each user who has joined is provided with its own view. These views are represented as virtual browsers in CloudBrowser.

Virtual browsers in the same chatroom must share related session state such as chatroom's title, chatting history, etc. Simultaneously, it should be possible for multiple users to create separate, unrelated chatrooms. CloudBrowser provides App Instance to achieve this behavior. Application instance state can be shared by multiple virtual browsers. The developer describe the state underlying an App Instance in the *application instance definition* file in the application bundle. For the chat application, this state may include the messages and participants of one or more chatrooms shared by these participants, see Figure 2.5.

To create a chatroom, a user would create an App Instance and share its URL with chat participants. As the participants join the chat site, a virtual browser is created on demand

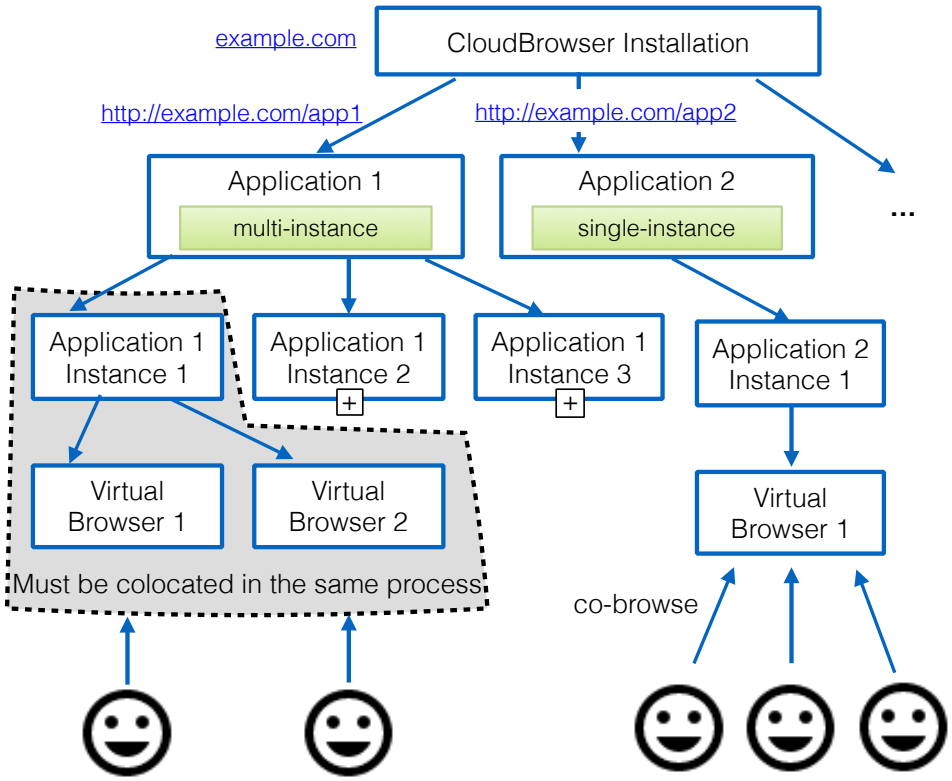

Figure 2.6: Application deployment model. Hierarchy of applications, application instances, and virtual browsers. Note that a single virtual browser may be broadcast to multiple clients (cobrowsing).

for each participant, which is connected to the application instance (the users can bookmark their virtual browser's URL to later return). The virtual browser can directly access the App Instance that contains the shared chatroom-related state, which simplifies the synchronization between this state and the virtual browser's DOM.

Thus in our model, an application can have multiple App Instances, each App Instance can have multiple virtual browsers. It is also possible for multiple clients to simultaneously access a virtual browser, although in this case all users will see exactly the same UI (See Figure 2.6).

2.3.2 Authentication

CloudBrowser provides support for common authentication-related features to applications, such as registering user accounts, external authentication through outside providers, and maintains authentication state transparently and outside the actual application code.

The application developer can enable authentication support for an application by setting the authentication option in the application's descriptor. The system will then redirect the user to a login page if the user has not been authenticated. Applications can access the user's account information via the CloudBrowser API once the user has logged in.

The login page itself is implemented by a virtual browser representing the system's login application. The login page provides two authentication options: a local authentication option where the user's credentials are stored in a local database, for CloudBrowser supports local account creation and lookup. Second, an OAuth [20] option which authenticate users via Google's OAuth authentication services. In this option, the user clicks a *Login with Google Account* button which redirects to a Google account authorization page asking the user to authorize CloudBrowser to access the user's Google account information. After the user confirms Google will redirect the user to CloudBrowser's authentication callback URL to finalize the authentication.

After the user logs in using either mode, the system adds an entry of the user's email address and the application's mount point to the user's session. Then the login page virtual browser is closed and the user is redirected to the URL he originally requested. For all subsequent requests of this session, the system will detect the user has logged in and the requests are handling directly by the application.

The user is automatically logged out after the session expires. The application can also display a link to the application's logout URL in the user interface to allow the user to manually logout the application.

2.3.3 Application Instantiation Strategies

Different types of applications need different strategies with regard to how to create App Instances and virtual browsers. These strategies differ with respect to how many instances there can be per application, and how many virtual browsers a user may create for each application instance.

CloudBrowser provides four application instantiation strategies that specify different ways to initiate App Instances and virtual browsers. The developer can set the application to adopt one of the application instantiation strategies in the application's descriptor.

multiInstance This is the most flexible option, which allows users to have multiple, separate virtual browsers connected to an application instance. For instance, a user could participate using separate nicknames in a chatroom. In those cases, the user has the largest flexibility, but will be exposed to having to manage the virtual browsers created for them. For instance, when visiting a chatroom application instance, they will need to manage whether to join an existing virtual browser or create a new one - similar to the choice a user may have when deciding whether to navigate to a new site in an existing browser tab or open a new one. In this model, there are also no restrictions on the number of instances created for an application.

singleBrowserPerUser In this mode, users may not create more than one virtual browser per application instance. When users access the application instance's URL, they will either be forwarded to their virtual browser or a virtual browser will be instantiated for them. This has the advantage that users will not be exposed to the management of virtual browsers - the application will appear to them as a standard web application whose URL they can bookmark. An example might be a chatroom application that only allows users to participate under a single nickname in each instance. This instantiation mode requires user authentication to establish a user's identity.

singleInstancePerUser Whereas multiInstance and singleBrowserPerUser support an ar-

bitrary number of application instance per application, and thus require the explicit creation and management of application instances, singleInstancePerUser applications support only a single application instance per user, which is automatically created on demand. singleInstancePerUser applications also limit users to a single virtual browser for their application instances, in the same way singleBrowserPerUser instances do. This mode is ideal for modeling single-page web applications in which only one view is needed for each user. This instantiation mode requires authentication to establish a user's identity.

singleAppInstance The application supports only one instance and a single virtual browser that is shared by all users. All connected clients will share a single server-side document in this singleton mode - this can be used for applications that display data, such as a weather application. These applications will not typically react to user input and users do not need to be authenticated.

Instantiation Strategy	Number of App Instances	Number of Browsers Per App Instance	User Manages App Instances	User Manages Virtual Browser
multiInstance	any	any	Yes	Yes
singleBrowserPerUser	any	1 per user	Yes	No
singleInstancePerUser	1 per user	1 total	No	No
singleAppInstance	1 total	1 total	No	No

Table 2.1: Application Instantiation Strategy

Table 2.1 summarizes how many App Instances and virtual browsers each initiation strategy allows applications to create.

For *singleUserInstance* and *multiInstance* applications where the user needs to manually create App Instances, the system provides landing pages where the user could view and manage his App Instances and virtual browsers. For instance, in a chat application, if the user requests the application's URL, he is redirected to the application's landing page where all the chatrooms (i.e. App Instances) he is participating are listed, from there the user can

create new App Instances, delete existing App Instances, share App Instances with other users, etc.

Chapter 3

Nodermi: A Remote Procedure Call Framework for Node.js

In the single process version of CloudBrowser, everything lives in the same address space, which allows application code to invoke system components' functionality. As CloudBrowser 2.0 divides CloudBrowser into multiple processes, some of those local method invocations need to be replaced by inter-process communication, because as the framework spreads its state and responsibility to multiple processes, an operation initiated in one process could require access to state located in another process. For example, in the single process version, closing a virtual browser is implemented by invoking the `close` method of the virtual browser object. If the virtual browser is in another process, closing it requires that a request be sent to that process. We could have introduced new objects and methods to send specific messages, but this approach is undesirable for the following reasons. First, there are many call sites that would need to be changed. Second, we would also have to design message formats for many different methods and write handler code to parse and process each of these messages. Third, since our codebase is continuously evolving, any changes to existing methods would require modifications to the message communication layer.

To avoid having to manually create and maintain messaging code for each method, we devel-

oped nodermi [34], an object-oriented remote procedure call (RPC) framework for Node.js.

Before we discuss the implementation of nodermi, we first introduce some terms for nodermi and RPC systems in general. We use the term *local object* to refer to objects that are allocated by the calling process. We use the term *remote object* to represent objects that are located in processes other than the calling process. We use *stub* or *remote reference* to refer to special objects created by nodermi that represents *remote object*s in the calling process (we can think of *stub*s as proxies to remote objects). We say an object is *remotely referenced* from a process if there are *stub*s representing it. Each *stub* represents one *remote object* and it has methods that mirror the *remote object*'s methods. From the programmer's perspective, *stub*s appear like ordinary local objects. When calling a *stub*'s method, the underlying RPC framework code sends messages to the *remote object*'s process to invoke the corresponding method there. For a given stub, the object it represents is called its *source object*, the process that creates the *source object* is referred to as the *host* of the stub. The *host* process takes on the role of a *server*, whereas the process that holds a *stub* to the object acts as a *client*. In JavaScript, since functions are first class citizens, it is possible to pass a function as an argument or return a function as a result. Nodermi treats functions and objects alike so that functions can also be referred to remotely.

3.1 Semantics

Synchronous Methods Unlike RPC frameworks in which a calling process blocks during the completion of the remote method [4], nodermi is fully asynchronous since JavaScript functions must not block. In nodermi a remote method call returns immediately after initiating the communication, similar to other IO-related asynchronous methods. Thus the method's return value is not available when the stub method returns on the client. To obtain the result of a remote method invocation, the remote method must be implemented asynchronously, passing its result(s) to the caller via a callback function. If a caller requires the knowledge

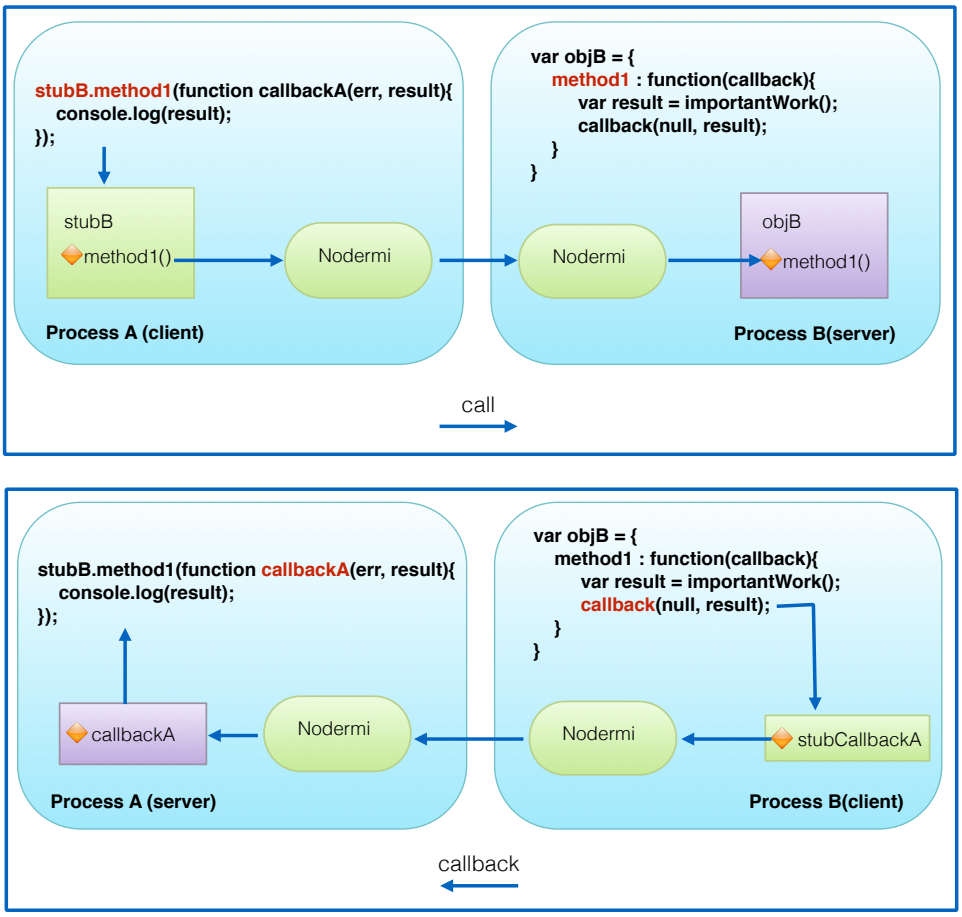

Figure 3.1: Nodermi remote method invocation example

that a method has completed - with or without returning a result, the method must be written in an asynchronous style. Thus, some existing methods will need to be changed to be able to invoke them remotely via nodermi.

Passing Arguments Nodermi copies most arguments by value, including built-in primitive types and objects. If an object refers to other objects, these objects are copied as well, so that an object's entire transitive closure is copied. Since JavaScript functions contain bindings to variables that are part of their closure, it is not possible to copy them. Instead, we create stubs for functions and methods before passing them to the remote object's method. If a method needs to invoke methods on objects passed as arguments, then such invocations

will cause the created stubs to initiate a remote method call to the original object. The client must create handlers to receive and process these remote method invocations. For instance, in Figure 3.1, when *Process B* calls the remote method *method1* in *Process A* with a function argument, a stub for the argument `callbackA` is created in *Process B* because it is not possible to copy function `callbackA` to *Process B*. The stub to `callbackA` is passed as an argument to `method1`. Then `method1` invokes the stub to `callbackA` as if it were an ordinary local function and the actual `callbackA` is invoked in *Process A*.

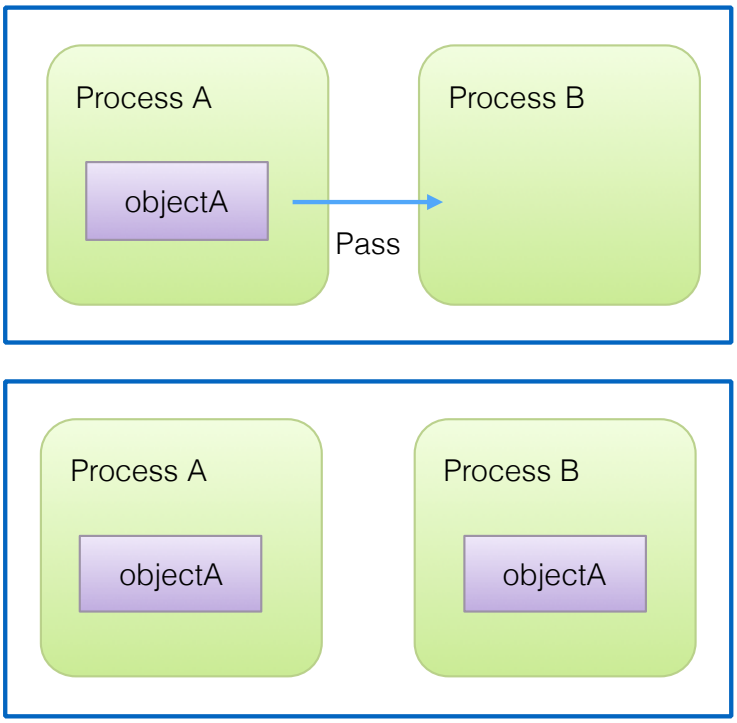

Figure 3.2: Passing arguments by value in remote method invocations. When `objectA` is passed to a remote method, the receiving process creates an exact copy of `objectA`.

For certain built-in types, e.g. `Date`, `Error`, `Buffer`, we copy their contents and recreate them via their constructors. Thus, method invocations on those objects are local, not remote. In this case, no stubs are created as emphasized in Figure 3.2. Besides built-in types, nodermi also allows programmers to register customized types to be copied and recreated via constructors. The objects of these customized types need to implement a method to

output their contents such that their constructor can recreate them. Unlike Java RMI[24]'s remote class loading mechanism, we do not support a way for the sender of such objects to provide the receiver with code for its methods. Rather, we use the receiver's versions of those classes.

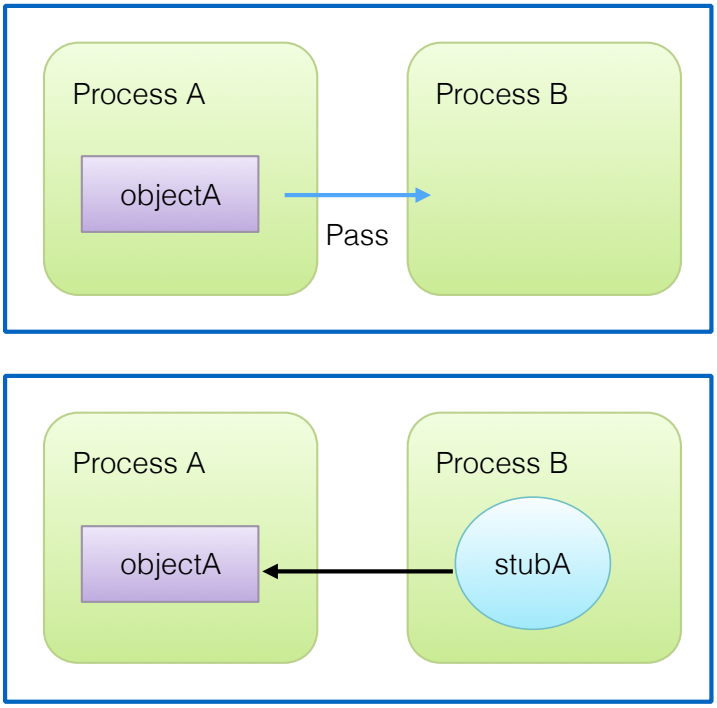

Figure 3.3: Passing by reference in remote method invocations. When an object is passed by reference to a remote object, the receiving process will create a stub to represent that object.

Nodermi allows programmers to pass reference to remote objects, represented by stubs, transparently as arguments to any method. If we treated such stubs representing remote references like other objects, we would create a remote stub referring to the passed stub. If the called method invokes a method on this stub, the corresponding remote method call would in turn initiate another remote method invocation to call the object the original stub represents. Since the remote object may in turn pass the received stub to other methods, the resulting chain of remote method invocations could be arbitrary long. To avoid this "zig-zag" problem, when we pass a stub to a remote method call, we create a new remote reference

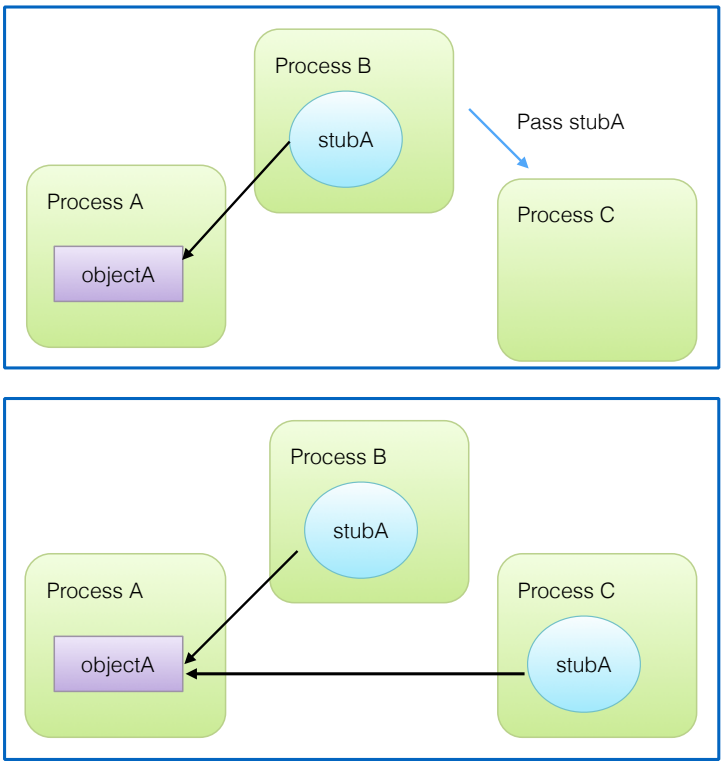

Figure 3.4: Passing stubs by reference in remote method invocations. When a stub is passed by reference to another process, nodermi creates a new stub to represent the stub's source object instead the stub itself.

referencing the stub's *source object* instead of referencing the stub itself (See Figure 3.4). Calling methods on this newly created stub will initiate communication directly with the remote object's host process. If the remote object resides in the same process as the receiver object, we pass a direct reference to the object instead.

Stub Generation. A stub's properties are a snapshot of the properties of its source object at the time the stub is created. Any subsequent changes to the stub's properties will not be synchronized with its source object. Vice versa, any changes to the properties on the source object have no effect on the stub either.

A stronger consistency model would propagate changes from the stubs to the original objects

and vice versa, as in DRuby[42]. However, it is not feasible to implement this semantics because in JavaScript reading or assigning a object property is a synchronous operation, thus we cannot engage in any communication to achieve such synchronization.

An alternative design would have been to disallow properties that are not methods. However, in this design, we would need to implement an asynchronous method for each read property of each object, requiring changes to many call sites. This would severely change the structure of existing code and introduce performance penalties from extra round trips. For most use cases in our system, providing a snapshot of a remote object does not affect the correctness of the program.

3.2 Design

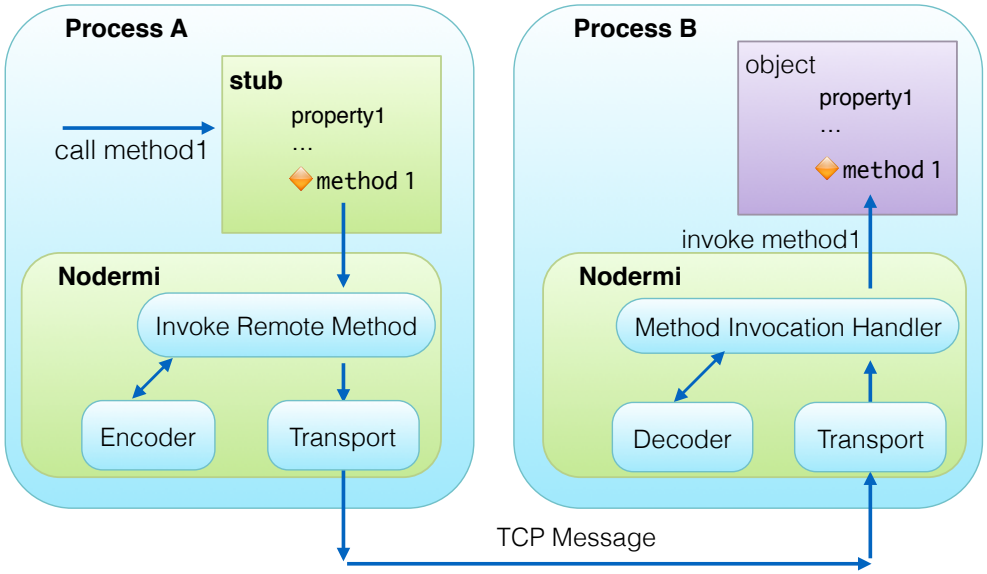

Figure 3.5: Overall Design of nodermi

As shown in Figure 3.5, a stub's methods call the *remote method invocation* module in nodermi. This module first encodes the arguments of the method call and other necessary information in a *method invocation message*. The transport layer then converts the message

to a binary string and sends it to the server process hosting the remote object. On the server side, the transport layer reads the message from the network and invokes a generic *method invocation handler*. The handler reads the description of the arguments by decoding the *method invocation message*. Then it recreates the arguments according to the semantics of argument passing (see Section 3.1). The handler uses the object's id to lookup the corresponding local object. Finally, the handler invokes the corresponding local method with the arguments it constructed earlier.

To locate the host process and source object of a stub, nodermi stores the host process's identifier and the source object's object id inside the stub so this information can be retrieved later by the *remote method invocation* module. The process identifier is a hostname/port pair that each process allocates for nodermi to listen for incoming TCP messages. The object id is a unique id nodermi assigns for each object that is remotely referenced.

Let us explain the message flow with the example shown in Figure 3.1. As shown in the code snippet in the top left of the figure, process A invokes `method1` with a function argument `callbackA`. nodermi creates a *method invocation message* containing the object id of `objB` and the method being called. The message also contains a description of the arguments: the object id of `callbackA` and the type of `callbackA`. After process B receives the message, it reconstructs the arguments for the method call: creating stub `stubCallbackA` for `callbackA` since it is a function. The stub `stubCallbackA` is a function and it stores `callbackA`'s id and process A's process identifier. After process B finds `objB` via its object id, process B invokes `method1` with `stubCallbackA` as an argument. `method1` then invokes `stubCallbackA` with `result`, which in turn initiates a remote method invocation to call `callbackA` in process A.

Bootstrap nodermi automatically creates stubs while passing arguments during remote method invocations on existing stubs, without requiring explicit calls to the framework API. To bootstrap the RPC communication and obtain an initial stub, nodermi provides

a `registerObj` method to register a local object under a name and a `retrieveObj` method to look up a remote object that has been registered via `registerObj` and create a stub for it.

Object Encoding When passing an object from one process to another, nodermi adopts several policies to control the encoding. Nodermi assumes properties whose names start with "_" are private properties and skips them during encoding; Nodermi skips properties that refer to certain types of objects, e.g. Sockets, which refer to native OS objects, which cannot be sent. Nodermi also provides a mechanism for programmers to specify which additional properties should be ignored during encoding. Finally, nodermi uses Protobuf [36], a very compact binary format, to serialize its messages.

3.3 Distributed Garbage Collection

JavaScript relies on garbage collection to reclaim memory taken by objects that can no longer be referenced by the program. However, the garbage collector cannot decide if a local object could still be referenced by another process via nodermi. In the example shown in Figure 3.1, before `method1` of `stubB` is called, the caller needs to hold a reference to `callbackA` to prevent it from being garbage collected.

If nodermi naively kept references to objects such as `callbackA` indefinitely, it would incur memory leaks as all objects that are even once referenced remotely will never be garbage collected even after the local and remote processes no longer use them.

Setting timeouts for the references of these objects and cleaning them automatically is not going to work either, since there is no guarantee as to when remote processes will no longer attempt to use a particular object. For example, when a process registers a listener function to a remote event publisher, the listener could be remotely triggered at anytime in the future.

Instead, we implemented a distributed garbage collection mechanism that ensures that the

entry to `callbackA` is deleted when the garbage collector is certain that it may no longer be invoked. Our mechanism is similar to the sequence reference counting algorithm designed by Birrell et al. [3].

The high level design of our distributed garbage collection algorithm is as follows. For each process, nodermi maintains an object map containing the objects that may still be remotely referenced. The object map prevents the local garbage collector from prematurely garbage collecting these objects. This map is also necessary for looking up local objects when handling *method invocation* messages.

Entries in the object map are removed when a nodermi process learns that there are no more remote stubs referencing the object. In turn, a nodermi process must inform the hosts of the objects for it has created stubs when those stubs become unreachable and subject to garbage collection.

Since JavaScript does not provide finalizers, we use weak references (implemented using node-weak [32]). Creation of a weak reference to a stub allows us to obtain notification via a callback when the garbage collector has freed the object. Weak references, unlike regular references do not prevent the garbage collector from doing so.

Thus, when a stub is no longer referenced by application code, it will be garbage collected and the corresponding weak reference's callback will execute. In this callback, nodermi sends the object's host process a *release message*.

Upon receiving a release message, nodermi knows that a remote reference to a local object no longer exists and will remove the corresponding reference from its object map. An object is garbage collected if the last entry to it from the object map is removed and there are no other local references to the object.

Figure 3.6 shows the structure of the object map where nodermi keeps the remotely referenced objects alive for other processes. Figure 3.7 shows the stub map in which nodermi stores weak references to stubs. The object map contains the local objects and information about

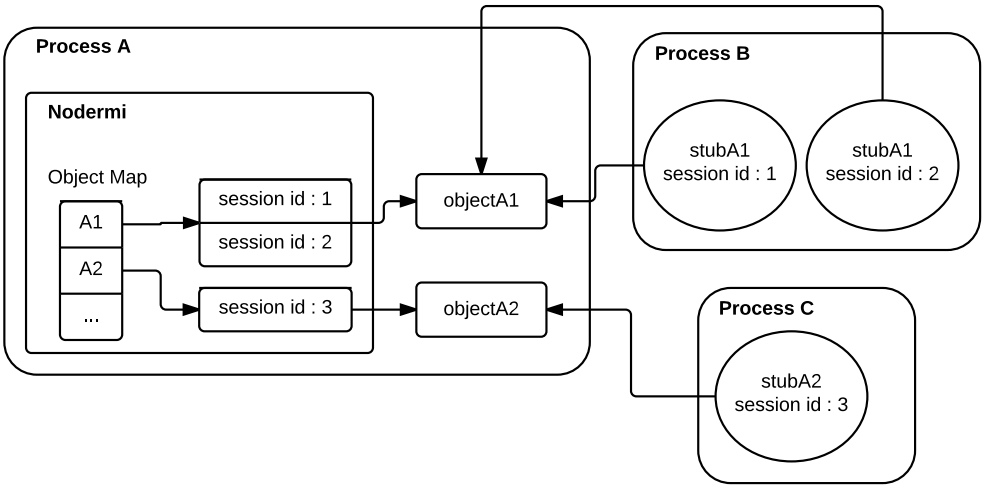

Figure 3.6: Nodermi object map. Nodermi holds strong references to local objects that are remotely referenced.

their remote references. A remote reference is represented by a session id, a unique id generated by nodermi every time it transmits objects to remote processes. When a local object is transmitted, the current session id is added to the object map indicating a new remote reference was created. On the receiving side, the session id is written to the *stub*s created in this transmission. When a *stub* is garbage collected, the *release message* will contain the remote object's id and the stub's session id.

When a process passes a *stub* to another process and the *stub* is not from the receiving process, then a new remote reference must be created to the stub's source object. In this case, we need to add a record to the stub's host's object map to represent this new remote reference. To implement this, before passing the *stub* the sender first sends a *reference message* to ask the stub's host to generate a new session id and put that session id into the host's object map. After that, the sender passes the *stub* along with the new session id to the receiver. The receiver then creates a new *stub* representing the original object and the new *stub* contains the new session id.

A process could terminate without sending releases messages to release its remote references. This leaves records of defunct remote references in other processes' object maps, thus creating

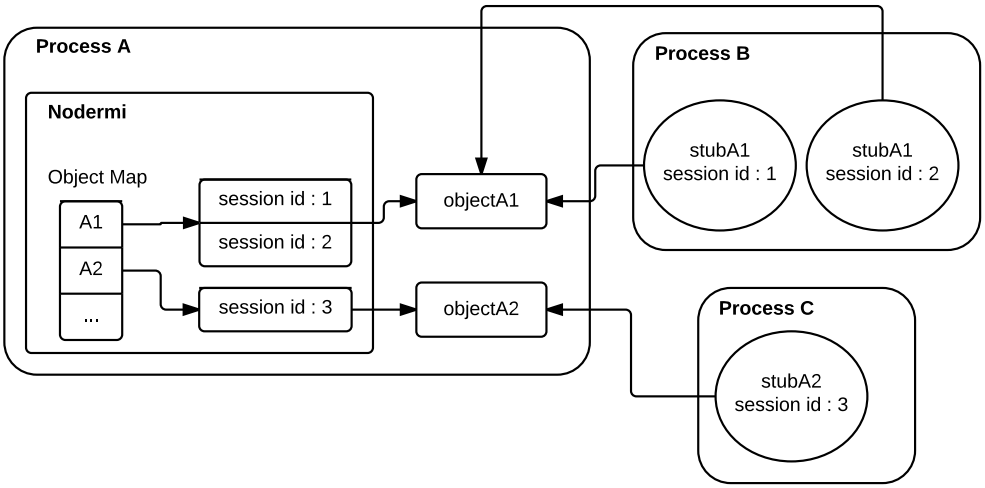

Figure 3.7: Nodermi stub map. Nodermi creates weak references to stubs so it can receive notifications when stubs are no longer used in the system.

potential memory leaks by holding up objects that are no longer remotely referenced. To remove these dead remote references, nodermi cleans a remote process's records in the object map if it detects that the remote process has terminated. To detect termination, nodermi uses a heartbeat mechanism that pings a process every 60 seconds, unless there is activity from that process. Nodermi keeps a table with the last response time of each remote process, which is updated every time it receives a message from a remote process. When the table entry for a process is not updated for more than 60 seconds, nodermi will send a ping message to that process. If the ping message is not answered within some timeout, all records of that process will be removed from the object map. Thus, a terminated process will be detected and all its remote references will be removed from the system.

It is possible for remote references to form distributed cyclic references as shown in Figure 3.8. Nodermi does not support garbage collect distributed cyclic references. We think distributed cyclic references are rare in practice and the programmers can always break the cycle by nullify the properties that introduce the cycles.

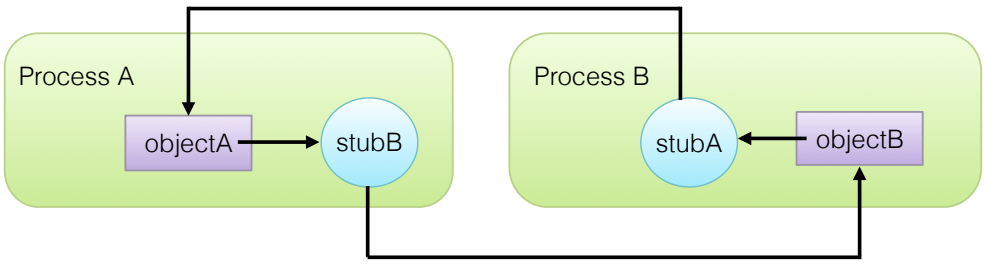

Figure 3.8: Distributed cyclic reference. `objA` references `stubB`, `stubB` is a remote reference of `objB`, `objB` references `stubA` which is a remote reference of `objA`. Neither of them can be garbage collected.

Chapter 4

Design and Implementation

This chapter first discusses the overview of the multi-process architecture of CloudBrowser 2.0. Then we discuss how client requests are dispatched and how load balancing is implemented. After that we discuss the implementation of different CloudBrowser 2.0 processes and how they interact with each other. In the end we discuss how CloudBrowser applications access the framework internal objects.

As shown in Figure 4.1, CloudBrowser 2.0 consists of a single master process, multiple reverse proxies, and multiple worker processes. All processes communicate via nodermi, which in turn uses standard TCP/IP sockets in its transport layer, so they can be located on a shared-memory multiprocessor machine or on different machines in a cluster. Worker processes host application instances and virtual browsers. The master process is responsible for the request dispatch logic which decides how to distribute the client load to workers. The actual dispatching is implemented by the reverse proxies, which forward users' requests to workers and copy workers' responses back to users. All reverse proxy processes are bound to the socket that accepts client requests, allowing the OS to distribute pending client connections in a round-robin fashion. The reverse proxy can relay both HTTP requests/responses as well as the bidirectional WebSocket protocol (see Section 2.2.2). Once the client has established a WebSocket connection with the server side, the majority of traffic will be WebSocket

messages for which there is relatively little per-message overhead.

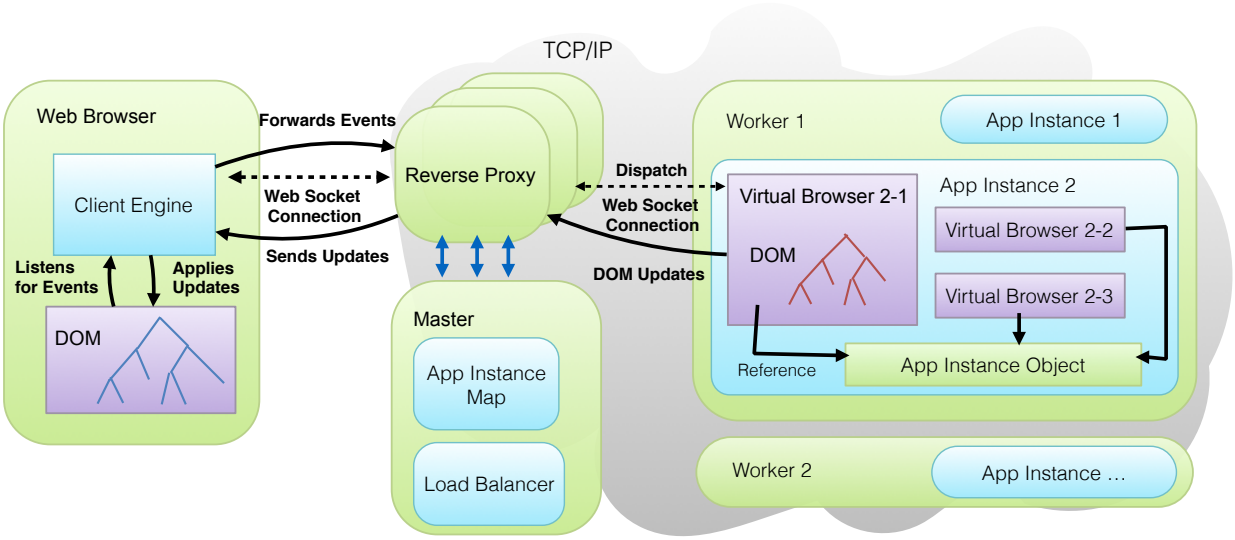

Figure 4.1: Multiprocess Process CloudBrowser Architecture Overview

4.1 Request Dispatch

When a request is being dispatched, the reverse proxy uses information contained in the request URL to make an appropriate forwarding decision. The system exposes three types of URLs for the users to access CloudBrowser applications. The formats of these URLs are as follows:

Application URL

`http://example.com/[app]`.

`[app]` represents an application's mount point. For example, if an application's mount point is *chat*, then its Application URL is `http://example.com/chat`.

App Instance URL

`http://example.com/[app]/a/[appInstanceId]`.

`[appInstanceId]` represents an App Instance's id. For example, if an App Instance's id is *appins1* and its application's mount point is *chat*, then its App Instance URL is `http://example.com/chat/a/appins1`.

Browser URL

`http://example.com/[app]/a/[appInstanceId]/b/[browserId]`.

`[browserId]` represents a virtual browser's id. For example, if a virtual browser's id is *browser1* and its App Instance id is *appins1* and its application's mount point is *chat*, then the virtual browser's Browser URL is `http://example.com/chat/a/appins1/b/browser1`.

From the user's point of view, the *Application URL* is similar to the homepage URL in a traditional web application. For a *singleAppInstance* or *singleInstancePerUser* application, requesting *Application URL* redirects the user to the only virtual browser he can access, the virtual browser is created automatically if it did exist before the user request. For a *singleBrowserPerUser* or *multiInstance* application, requesting *Application URL* redirects the user to the landing page (Figure 4.2) in which he can navigate to any of his virtual browsers. The user may prefer use *Application URL*s because it is easy to remember and he can access his virtual browsers from *Application URL*s without bookmarking individual virtual browsers' URLs.

App Instance URLs are used for joining a specific, already created App Instance. For *singleBrowserPerUser* applications where each user can create only one virtual browser for a given App Instance, the system directs the user to his own virtual browser in the specified App Instance when handling *App Instance URL*.

For *multiInstance* applications, it is the user's responsibility to decide whether they wish to join an existing virtual browser or create a new one, the system will lead the user to a landing page to make that decision. For *singleBrowserPerUser* and *multiInstance* applications, users can share an App Instance with others by sharing its *App Instance URL*.

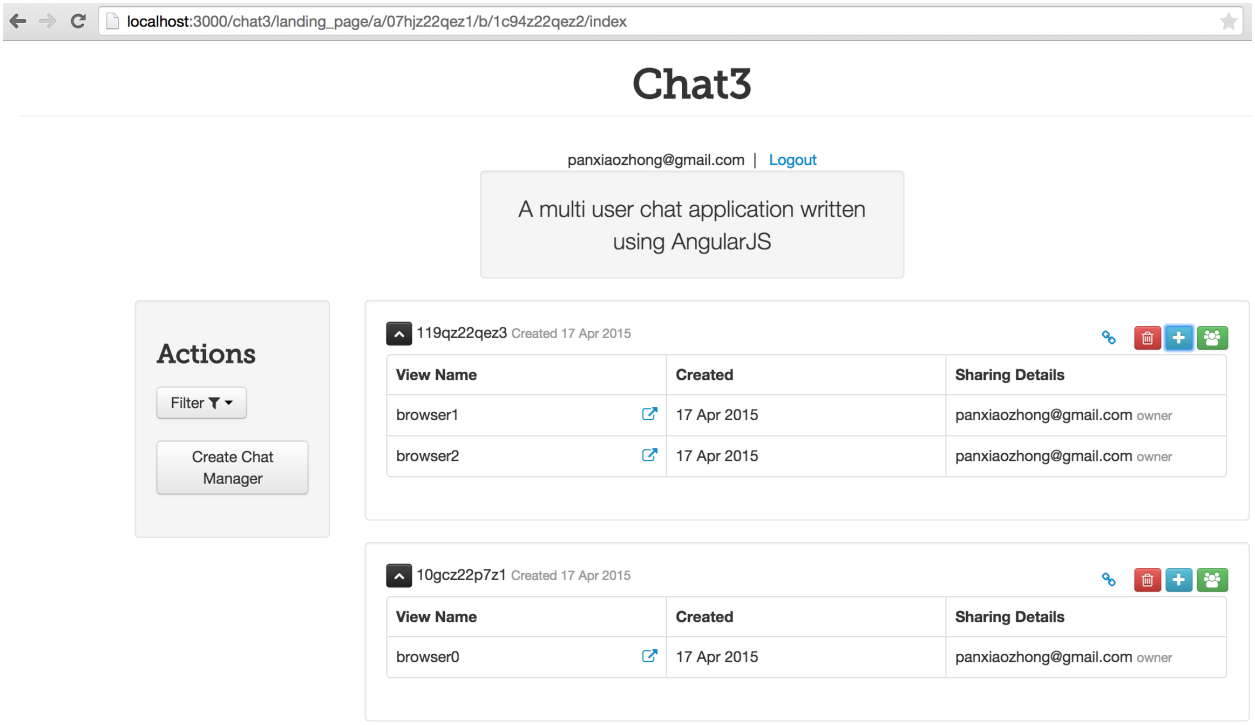

Figure 4.2: User interface of a landing page. The user can manage all his App Instances and virtual browsers including those shared by others in the landing page.

App Instance URLs are not meaningful for *singleAppInstance* and *singleInstancePerUser* applications because there is only one App Instance for a given user.

Browser URLs are for connecting to a specific virtual browser. A user can bookmark a *Browser URL* so he can revisit that specific virtual browser in the future. The user can also share a virtual browser with other people by sharing its *Browser URL*.

As discussed in Section 2.3.1, multiple virtual browsers may share data structures belonging to an App Instance. CloudBrowser 2.0 colocates every App Instance and its virtual browsers in the same worker process. Once an App Instance has been created in a particular worker process, all future requests to *App Instance URL*s and *Browser URL*s for that instance's ID must be routed to that worker. When the user requests an *Application URL*, the system needs either allocate a virtual browser or redirect the user to a landing page according to

the application's instantiation strategy. Because the landing page itself is a virtual browser, the system always allocates a virtual browser for the user when handling user's *Application URL* request. Then the system redirects the user with the virtual browser's URL.

Although the reverse proxy processes are in charge of the actual forwarding, the master process keeps track of the forwarding map. Thus, when a client sends a HTTP request, a reverse proxy process will accept this HTTP request and ask the master where to dispatch this request.

If the request URL is a *Browser URL* or *App Instance URL*, the master extracts App Instance id from the URL and returns the associated worker via an App Instance id to worker lookup table. This lookup table is updated every time an App Instance is created or removed. For an *App Instance URL* request, the worker will continue to find the associated virtual browser for the user and send back a redirect with the virtual browser's URL.

If the request URL is an *Application URL*, the system's behavior varies according to the application's instantiation strategy. For example, for a *singleInstancePerUser* application, the system can either create a new App Instance for the user if the user does not have an App Instance or use the user's existing App Instance if otherwise. The instantiation strategy specific logic is handled by workers, not the master, because it requires access to authentication information. Thus, the master can pick any worker using a load balance algorithm (detailed in Section 4.2), and the selected worker will then process the request according to the instantiation strategy. After consulting the authentication information, this worker will redirect the client to a specific Browser URL. The virtual browser corresponding to that URL may be located in another worker process.

When a worker process receives a Browser URL request, it will locate the corresponding virtual browser and responds with an HTML document that contains information to bootstrap the client engine. The client engine will create a WebSocket connection to establish an RPC channel to the worker process. First, the client engine sends a WebSocket handshake request to initiate the WebSocket connection. The handshake request URL contains the associated

App Instance id and virtual browser id so the master can find the corresponding worker for this request. After the worker sends back the handshake response, the connection between the worker and the reverse proxy that performed the handshake, as well as the connection between the reverse proxy and the user remains open. The reverse proxy will relay all subsequent WebSocket messages without parsing them and without requiring repeated calls to the master to obtain forwarding information.

As the client engine renders the client-side view of the page, it may trigger requests for auxiliary resources such as images or spreadsheets. These requests will contain the App Instance id in the URL, which the master uses to forward those requests to the appropriate worker.

In this design, the reverse proxies ask the master for the dispatch decision for every HTTP request. However, HTTP requests are required only when users reconnect to virtual browsers - most of the actual interaction with a virtual browser is performed using RPC messages carried over WebSocket connections. Thus, the overhead of inter-process communication between the reverse proxies and the master to obtain forwarding information affects only a small portion of the network traffic.

We also provide an option to embed a reverse proxy instance inside the master process. If the system is configured with one single embedded reverse proxy, reverse proxy and master can communicate directly. However, in this mode, the system can support fewer concurrent users than when using multiple reverse proxies.

4.2 Load Balancing

The load balancing algorithm is invoked in two scenarios: First, when the user requests an *Application URL*, the master needs to find a worker to handle this request. Second, it is invoked when the system is about to create an App Instance. Although such action can originate in any worker, the load balancing algorithm is performed by the master.

For other user requests the routing is determined by the App Instance-to-worker map so there is no need for load balancing. In particular, the creation of virtual browsers is not subject to load balancing. Virtual browsers have to be placed in the same worker that hosts its App Instance.

We support two load balancing strategies: first, the master can assign the load to workers in a simple round-robin fashion. However, since App Instances may vary widely in terms of the actual cost they impose on a worker and App Instances can be terminated, the round-robin assignment works well only for cases in which resource use is uniformly distributed. We also implemented a load-based scheme in which workers periodically report a measure of current load to the master. The master will select the worker with the lowest load when making load balancing decisions. We have found the amount of heap memory that is currently in use a good measure of a worker's momentary load.

In the load-based mode, the master's knowledge of a worker's load is not always up-to-date as the worker's load can change before the master receives the next report from the worker. We have found that this can lead to very unbalanced load distributions. For example, when the system is creating a burst of App Instances, they are all assigned to the same worker that has the lowest load at that moment and this worker ends up hosting a disproportionate load. To mitigate this issue, after the load balancing algorithm selects a worker, the algorithm makes a projection of the worker's load after accepting the new load. The projected value is used as the worker's load value until the master sees the worker's next report. We have found it unnecessary to exactly predict the amount of incremental load as long as we do not significantly underpredict. Based on our evaluation, a virtual browser in a non-trivial application takes about 6M. The master projects a worker's load increase by 10M for every new App Instance assignment assuming every App Instance will create 1 or 2 virtual browsers. For most cases, the master overestimates the actual load increase. It is not a problem because the master corrects its estimate as soon as it sees the worker's next report.

4.3 Master Implementation

This section describes the main modules in the master and how they interact with worker processes.

Application Manager

The Application Manager is responsible for maintaining the state of applications. It reads application bundles (discussed in Section 2.3.1) and initializes the data structures used to represent applications.

Worker Manager

The Worker Manager is responsible for the request dispatch and the load balancing algorithm.

Reverse Proxy Manager

The Reverse Proxy Manager starts multiple reverse proxy processes as child processes of the master and handles the communication between the master and the reverse proxies.

After initialization, the master exposes the *Application Manager* and *Worker Manager* objects to the workers via nodermi's `registerObj` method (discussed in Chapter 3).

The *Application Manager* includes methods related to application management, such as listing applications, creating new applications, removing applications, etc. The applications are represented by application objects. An application object has methods to manage the internal state of an application, including methods to change the application's authentication policy, register a new App Instance, remove an App Instance, etc. Since the *Application Manager* is registered in nodermi, workers can obtain remote references to it; the application objects returned by its methods are also automatically represented as nodermi remote references. The master's Application Manager coordinates with the Application Manager objects in each worker as described in Section 4.4.

The *Worker Manager* contains methods for workers to report their load and to register the URL at which they can be reached and . It maintains a table of App Instance id to worker URL for dispatching requests. This table is updated every time an App Instance is registered or removed from the *Application Manager*.

4.4 Worker Implementation

This section describes the implementation of worker processes. The key components of a worker process include:

Application Manager Manages applications' metadata, application instantiation logic, local App Instances and local virtual browsers. It provides methods to lookup local App Instances and virtual browsers by ids. It also processes incoming HTTP requests.

Workers pass references to their *Application Manager*s and application objects to the master's *Application Manager* via remote method invocation. The master uses remote references to these worker objects to push application-related changes to the workers. For instance, the master calls a worker's `addApplication` method remotely to push new applications to that worker and remove an application from that worker by calling the worker's `removeApplication` method remotely.

Session Manager Manages HTTP sessions and users' login states. The sessions are stored in a database server that is accessible for all workers. The users' login states are stored in session properties as discussed in Section 2.3.2.

HTTP Server Listens for HTTP requests from reverse proxies and dispatches the requests to the *Application Manager*.

Permission Manager Manages users' permissions to applications, App Instances and virtual browsers. If an application enables authentication, it can call the *Permission*

Manager to grant or revoke access to App Instances and virtual browsers for specific users.

The system will reject HTTP requests that violate the permission settings in the *Permission Manager*. For instance, if a user requests a *Browser URL* of a browser he does not have permission to connect, the request will be rejected.

If the application does not set customized permissions, the *Permission Manager* enforces a default permission policy which is useful for most cases. For example, a user cannot access the virtual browsers that were created by other users. To go beyond this default policy, the landing pages of *multiInstance* and *singleBrowserPerUser* applications include interfaces for users to manage the permission settings of their App Instances and virtual browsers.

After creating its internal components, the worker process will try to obtain remote references to the master's *Application Manager* and *Worker Manager* via nodermi, then it fetches all the applications' metadata and registers itself to the master by remote method invocation. If the master is not running, the worker will keep on retrying until these remote references are obtained.

When a worker receives an HTTP request, it parses the request to extract App Instance id, virtual browser id and retrieves associated user information from session.

If the request does not have an App Instance id, the worker either allocates a virtual browser in the system or redirects the user to a landing page. The App Instance associated with the virtual browser could be located at the worker itself or at another worker. If an App Instance needs to be created, the worker will ask the master to find a worker to create the new App Instance, a virtual browser will also be created in the new App Instance. Because user interactions are handled by virtual browsers, the worker then will redirect the user to the virtual browser's URL.

If the request contains an App Instance id but does not have a virtual browser id, the worker either allocates a virtual browser for the request or redirect the request to a landing page. In the first case, the worker either picks an existing virtual browser in the App Instance or creates a new one based on the application instantiation strategy, then the worker sends back a redirect response with the virtual browser's *Browser URL*.

If the request has a virtual browser id, the worker finds the matching local virtual browser to handle the request. For *Browser URL* requests, the worker sends back the initial HTML document to bootstrap the client engine. For resource requests, the virtual browser will retrieve the corresponding resource files either from the local file system or from other servers based on their URL, then send back the content. For WebSocket handshake requests, the worker will send back a handshake response message and add the WebSocket connection to the virtual browser, which keeps track of connected clients. The subsequent WebSocket messages will be directly handled by RPC methods of the virtual browser.

4.5 Secure Access to Framework Objects

Applications frequently need access to our framework's internal objects. This is particularly true for CloudBrowser's administration interfaces, which are themselves implemented as CloudBrowser applications. For instance, in our administration dashboard one can view all the installed applications and upload new applications. General applications also require the help of the framework to accomplish common tasks. For example, in a chat room application, a user can terminate a chat room (i.e. application instance) and close all its associated chat windows (i.e. virtual browser), which are actions that are managed by the application manager objects discussed in Sections 4.3 and 4.4.

We designed an API layer through which the applications can call the framework's methods. We do not allow applications to call framework code directly for the following reasons:

Security The application code in a given virtual browser runs with the permissions of the virtual browser's creator. When the application does not require user identification, then the creators of the application's virtual browsers are set to be the application's owner.

The operations issued by the application code need to conform with the permission settings in the *Permission Manager* (discussed in Section 4.4). For example, only the system administrator is allowed to shutdown all applications. These checks take place in the API layer.

Isolation Application code must not manipulate internal objects in unexpected ways.

Compatibility We wish to decouple the application code and the framework code so that changes made to the framework code do not break the applications.

Resource Reclamation Framework objects must be available for garbage collection after they are no longer needed by the framework, and the objects created by application code to be available for garbage collection after their associated virtual browsers and App Instances are closed. If applications were allowed direct access to the framework objects, it would be impossible to fulfill this requirement since the objects created by application code and framework objects could hold references to each other.

4.5.1 Design

The CloudBrowser API includes four JavaScript classes covering essential features applications need to access framework objects.

APIBrowser Represents a virtual browser. It has methods to manipulate the associated virtual browser, including setting the browser's permissions, closing the browser, etc.

APIAppInstance Represents an App Instance. It has methods to manipulate the associated App Instance and list the App Instance's virtual browsers as *APIBrowser* objects.

APIApplication Represents an application object. It has methods to manipulate the associated application and list the application's App Instances as *APIAppInstance* objects.

APICloudBrowser It has methods to manage applications and list applications as *APIApplication* objects. For every virtual browser one APICloudBrowser object is created and injected as a singleton global variable in the application code namespace so that it can be used by application code.

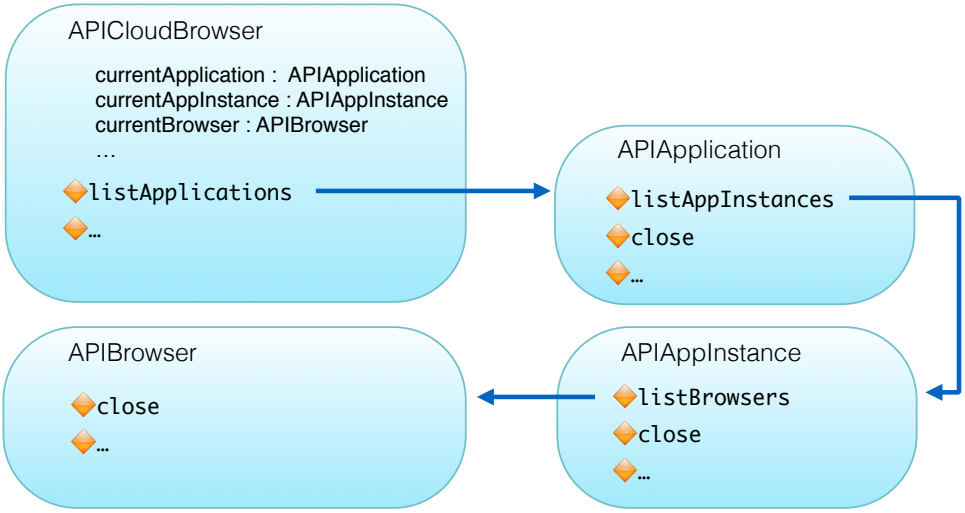

Figure 4.3: API class design

As shown in Figure 4.3, based on the initially provided *APICloudBrowser* object, application code can obtain API objects representing applications, application instances and virtual browsers. In most applications, the current application object, the current App Instance and the current virtual browser are frequently accessed in application code, thus the *APICloudBrowser* object provides API object properties to these objects directly so that the application code can access them without having to call a chain of methods. Besides framework objects, the *APICloudBrowser* object also contains properties that provide auxiliary services such as sending emails.

4.5.2 Implementation

API objects are implemented using a proxy design pattern [17]: when their methods are called, they will first perform permission checking if necessary and then call the corresponding methods of the framework objects they represent. The proxy design pattern requires that the proxy objects keep references to the objects they represent. In languages that feature encapsulation via field modifiers, such references are store in private fields so that the caller of the proxy object cannot obtain a reference to the underlying object being represented. Since there is no language support for private properties in JavaScript, we cannot store the internal objects as properties of the API objects because the application code could then obtain a reference to internal objects using object inspection.

We use closures to implement an equivalence to private properties [9] as shown in Listing 4. We store internal objects as local variables in the API objects' constructors and define API methods inside the constructors. The API methods can reference internal objects directly because they are part of its closure, even after the constructor has returned. The application code cannot get references to internal objects because they are not stored as properties and they cannot be revealed by reflection.

```
1  // constructor of an API class
2  function APIClass(internalObj){
3      // 1. hide internal object from application code
4      // 2. weak reference to the internal object
5      var _internalObj = weak(internalObj);
6      // define API methods
7      this.someMethod = function(){
8          // permission checking code
9          checkPermission(currentUser, 'someMethod');
10         //3. proxy method invocation
11         _internalObj.someMethod();
12     };
13 }
```

Listing 4: Code snippet of API classes

The internal objects' life-cycle should be managed by the framework. When applications obtain references to the internal objects via the API layer, they should not prevent the internal objects from being reclaimed. For instance, when a virtual browser object is referenced indirectly by an application, the framework should be able to close it and after that the garbage collector should be able to reclaim the virtual browser object. To avoid holding references to the internal objects that are no longer needed by the framework, the API objects only keep weak references to the internal objects (See Listing 4 line 5). We use a Node.js package node-weak [32] to implement weak reference. Unlike ordinary references, objects referenced only by weak references are free to be garbage collected. In this way, when an internal object is no longer referenced inside the framework, it is available for garbage collection even it is still referenced by the API layer. After an internal object is garbage collected, the API objects that used to represent it are invalid, they will throw exceptions when their methods are called so that the applications will know the corresponding internal objects ceased to exist.

The framework may store references to objects created by application code. Right now these references are created by the event register methods provided by the API for the application to register listeners associated with internal objects. For example, the application can register an event listener to notify the current user when someone shares an virtual browser with him. Because these listeners are registered via the API, the API layer manages the references to these listeners. When a virtual browser's `close` method is called, the API unregisters all event listeners created by the virtual browser. In this way, when a virtual browser is closed, the framework won't keep references to application-created objects that would otherwise cause leaks and prevent full reclamation.

Chapter 5

Evaluation

Our main focus in this evaluation is to demonstrate that the multiprocess implementation in CloudBrowser 2.0 can scale with the number of available cores on which to place separate workers. Our secondary focus is to investigate the cost of using different client libraries on the throughput we achieve.

5.1 Methodology

To test CloudBrowser 2.0 with existing applications, we developed a small expect-like language in which to describe scenarios. A client test tool interprets test scripts written in this language and makes RPC calls to a server in the same way the client engine would in actual deployment, then checks whether the server is making the expected RPC calls back that would reflect requests to the client engine to update the DOM the user sees. However, to minimize CPU consumption, the tool does not maintain a DOM in the same way a real browser would.

Like a (patient) human user, the benchmark tool will not send the next event until after the previous event resulted in the expected DOM update. Our test language allows us to express

```
1  {"type" : "textInput", "target" : "node40", "textType":"clientId",
2      "endEvent" : "enterKey", "keyEvent" : "basic"}
3  ===
4  {"type": "expect","expect":
5      [{"event": "DOMCharacterDataModified", "containsText" : [{"type" : "previousInputValue"}]}]}
6  ===
7  {"type" : "eventGroup", "count" : 300,
8  "events" : [
9      {"type" : "textInput", "target" : "node68", "textType" : "random", "endEvent" : "enterKey",
10     "keyEvent" : "basic", "tag": "textarea"},
11     {"type" : "expect",
12         "expect" :[
13             {"event": "DOMCharacterDataModified",
14             "containsText" : [{"type" : "previousInputValue"}]}
15         ],
16         "wait" : {"type":"random", "max":10000, "min":5000}
17     }
18 ]
19 }
20 ===
```

Listing 5: Configuration language for the benchmark tool

repetition and has the ability to check the correctness of expected responses. Listing 5 is an example benchmark tool configuration written in this language. Line 1–5 describes a test scenario in which the simulated user first types a string in a input box specified by id, then waits for a DOM event update from the server that echoes the previously input string. Line 7–20 describes a loop in which the simulated user repeats a sequence of actions 300 times. To simulate human processing speed, a programmable "thinking delay" can be introduced between receiving an expected response and sending the next event, as shown in line line 16.

To create test scenarios from actual applications, we run those applications using a real browser and record the client/server interactions in a log. We then transform this log manually into a test script that is later run by our test tool. To avoid limiting throughput when the test client becomes CPU bound, we can start multiple instances of the client.

5.1.1 Experimental Testbed

We used a dual-socket, 8 core Intel Xeon 2.27GHz processor with 38GB of RAM on the machine on which we run CloudBrowser 2.0's master, worker, and reverse proxies. We colocate the first reverse proxy with the master in the same process. This system is connected via Gigabit Ethernet to a 8 core AMD machine with 16GB of RAM on which the test client(s) execute. Both machines run Ubuntu 14.04 with a stock Linux 3.13 kernel. We use Node.js 0.10.33 and the version of jsdom 2.0 we customized. We use the round robin load balancing strategy discussed in Section 4.2 to ensure an exactly predictable distribution of application instances onto workers.

5.2 Click Application

Our first benchmark is a simple click application that increments a counter on a page whenever the user clicks. It is written without any libraries, making direct use of the JavaScript DOM API. It only has 14 lines of HTML and 5 lines of JavaScript code. As such it most closely measures the overhead of our framework only, e.g. the overhead of making RPC calls, serialization and deserialization, and dispatching events into the server-side DOM, observing any DOM changes occurring as a result, and relaying those to the client.

We consider two possible scenarios, a fast "Back-to-back" click application with no thinking times between clicks, and a more "Human-paced" click application in which there is a think time drawn from a uniform distribution in the range of 1 to 2 seconds. We perform 5 runs for each data point and report the mean. For all benchmarks, the variance was small. The vast majority of data points incurred a relative standard deviation below 5%, the maximum relative standard deviation was 11%.

We measure throughput in terms of operations per second and average latency. Figure 5.1 shows the throughput of the application for different numbers of workers. In those cases, all

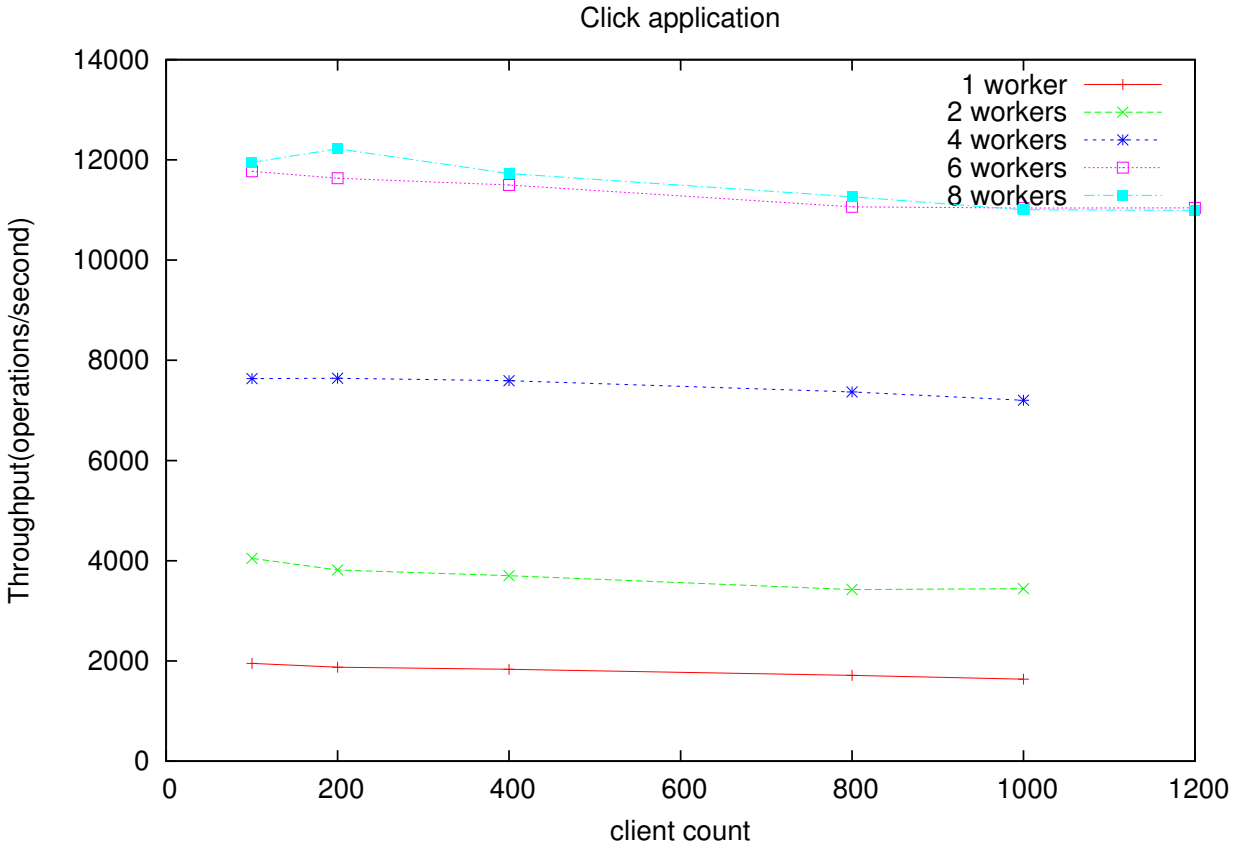

Figure 5.1: Throughput of "Back-to-back" click application.

workers become CPU bound, adding more workers that can use additional cores increases throughput near linearly up to 6 workers. In this scenario, a single reverse proxy is sufficient to support up to 8,000 operations per second at which point it becomes CPU bound; for the 6 and 8 worker cases, we add a second reverse proxy process. Since the reverse proxy processes run on the same machine as the workers, throughput does not increase beyond 6 workers since this machine has only 8 cores. We carefully monitor the CPU usage on the benchmark client machine, adding new test driver instances as needed, up to 8.

For the back-to-back scenario, throughput is limited by the CPU capacity available to the workers for 100 clients or more. Throughput stays relatively constant as the number of client increases, but the observed latency increases, which is shown in Figure 5.2. Note that we

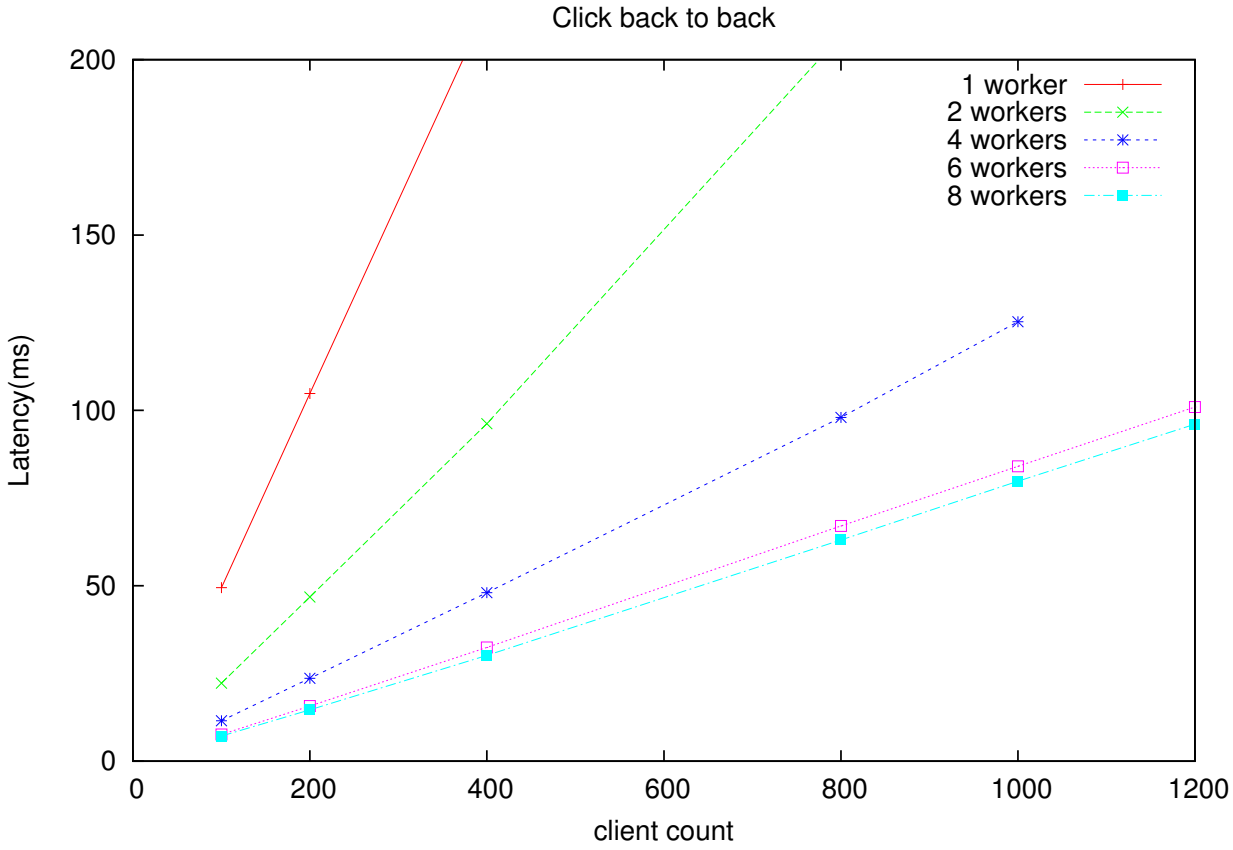

Figure 5.2: Latency of "Back-to-back" click application.

consider a latency of more than 100ms unacceptable [30], which is why we cut off the y-axis accordingly. As such, in this scenario, a single worker may support up to 200 clients, and 6 workers can handle about 1,200 clients before latency increases to unacceptable levels.

When introducing think times to simulate "human-paced" clients, we obtain the throughput and latency shown in Figures 5.3 and 5.4. In those cases, a larger number of clients can be supported, and maximum throughput will not be reached until the number of clients ramps up. For each worker configuration, the maximum acceptable latency is reached slightly before maximum throughput is reached. In these experiments, we do not increase the number of clients further if latency has already reached unacceptable levels. We must note that the latency reported here does not include a wide-area network (WAN) delay; which when taken

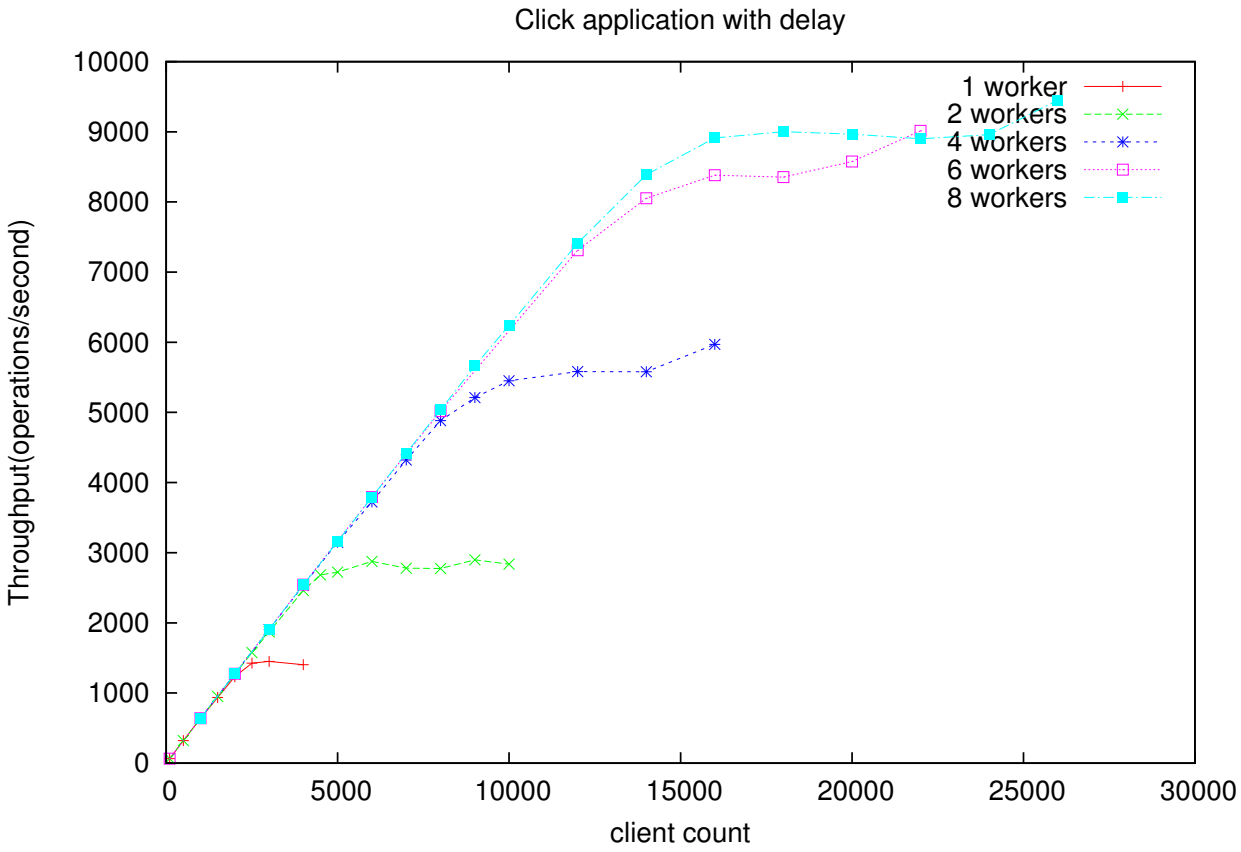

Figure 5.3: Throughput of click application, after introducing artificial delay.

into account would reduce the bounds on acceptable processing delay. Nevertheless, we conclude that our method of scaling to multiple processes is effective and that the reverse proxy in particular does not become a bottleneck.

5.3 Chat Application

A key part of the productivity promise for using the server-centric CloudBrowser framework is the ability to reuse high-level libraries such as AngularJS with little or no changes, so that applications prototyped in AngularJS can be directly executed in virtual browsers. These libraries are designed for client-side use, however. We prototyped a Chat application to

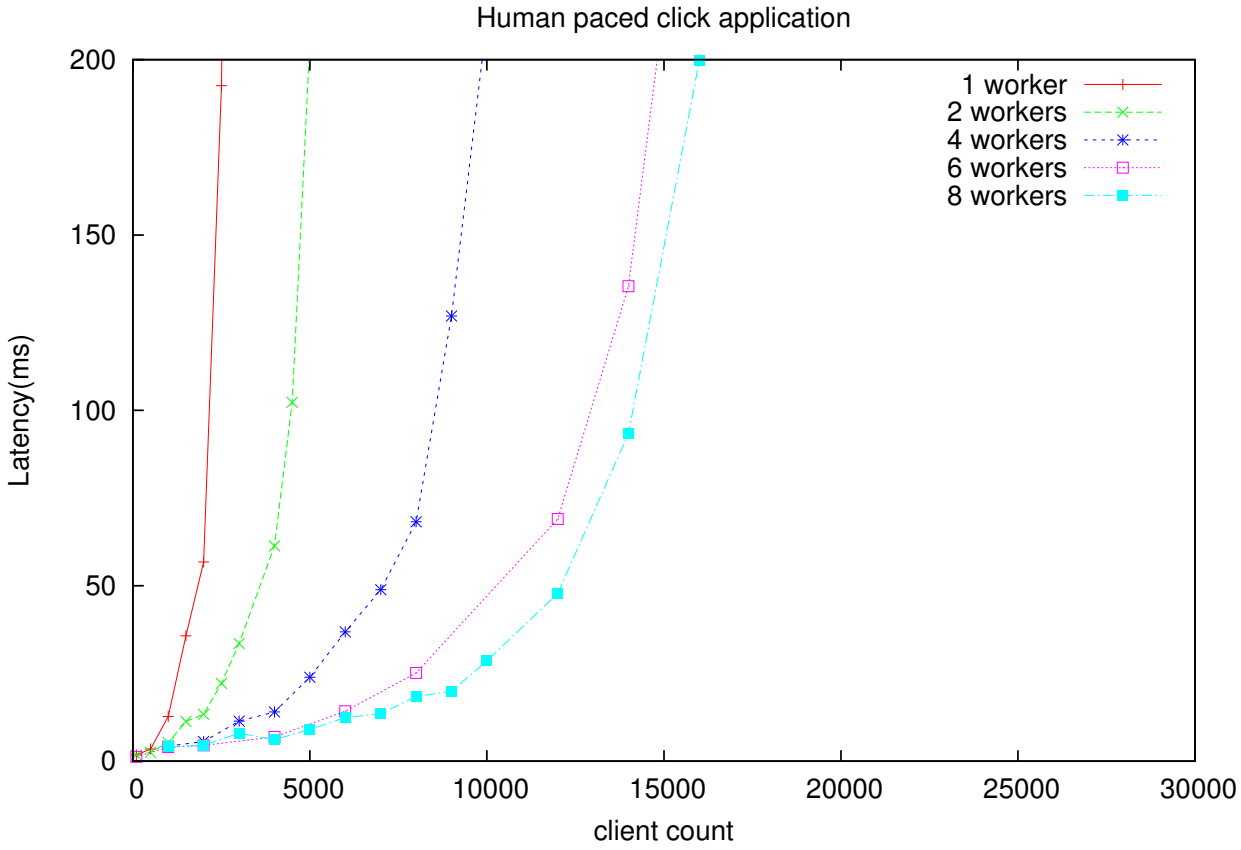

Figure 5.4: Latency of click application, after introducing artificial delay.

investigate the overhead of this usage scenario, a screenshot is shown in Figure 5.5. The application is only 207 lines of HTML and JS code while providing features such as creating chatrooms, joining them, chatting and changing the desired display name.

We make use of shared application instance data as discussed in Section 2.3.1 to hold the last 50 chat messages. Each application instance is visited by 5 simulated users, each instantiating their own virtual browser. Each user sends 100 chat messages in our script, consisting of a sentence of 15-20 characters. We set the think time to 5-10 seconds between messages. We define latency as the time taken to hit enter and when the message appears in the chat window.

Figure 5.6 shows the latency perceived by our benchmark tool for different numbers of

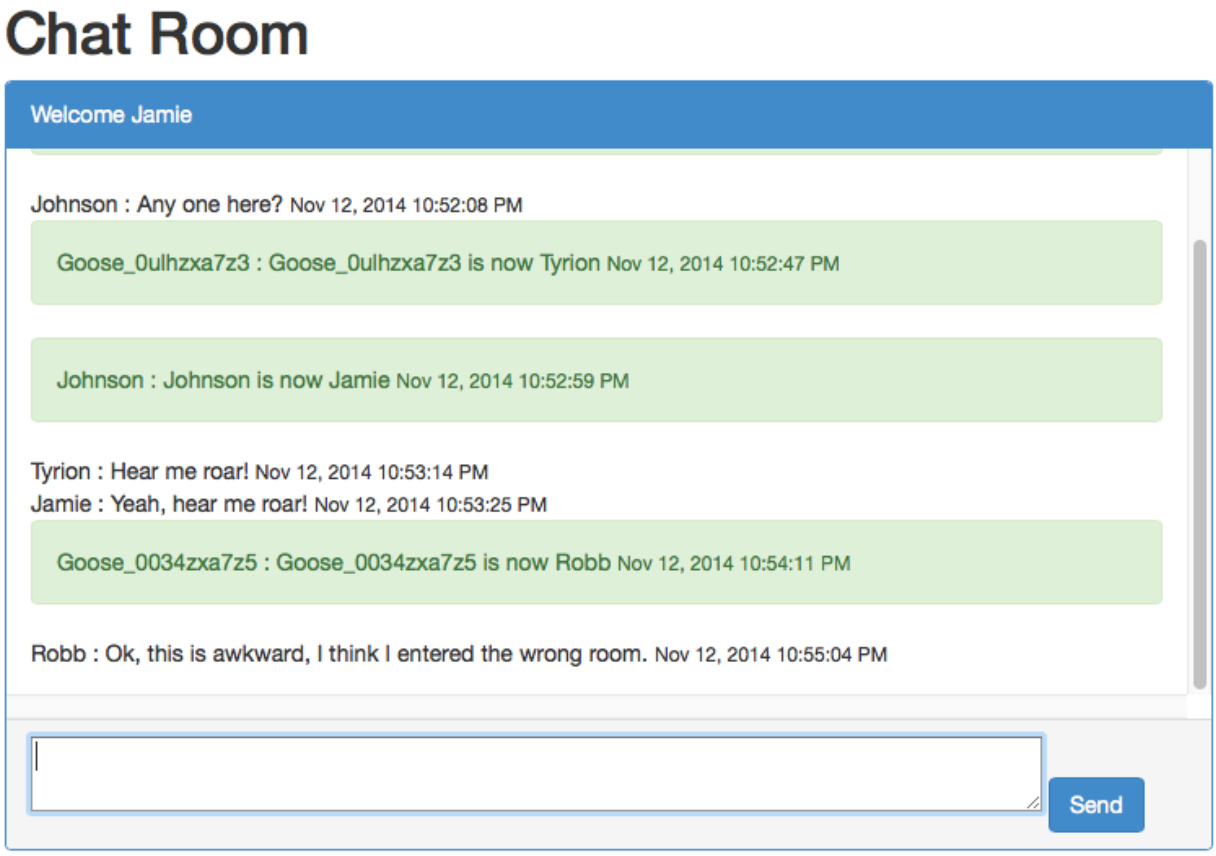

Figure 5.5: Chat Room Application

clients. While scaling to multiple workers remains effective, the use of AngularJS imposes a significant cost, reducing the number of concurrent users that can be supported with the same hardware. Most of the time is spent in the AngularJS framework, which we discovered through profiling. Optimizations made in successive revisions of AngularJS heavily impacted our performance; for instance, a single commit to optimize needless re-execution of so-called filters in AngularJS improved benchmark performance by 20%. As such, our numbers provide limited information about the performance envelope of server-side environments; rather, they provide a snapshot into the current state of engineering high-level libraries such as AngularJS.

To eliminate the impact of AngularJS, we also reprototyped the same application with a lower-level library, jQuery, which increased its size significantly (and made it significantly

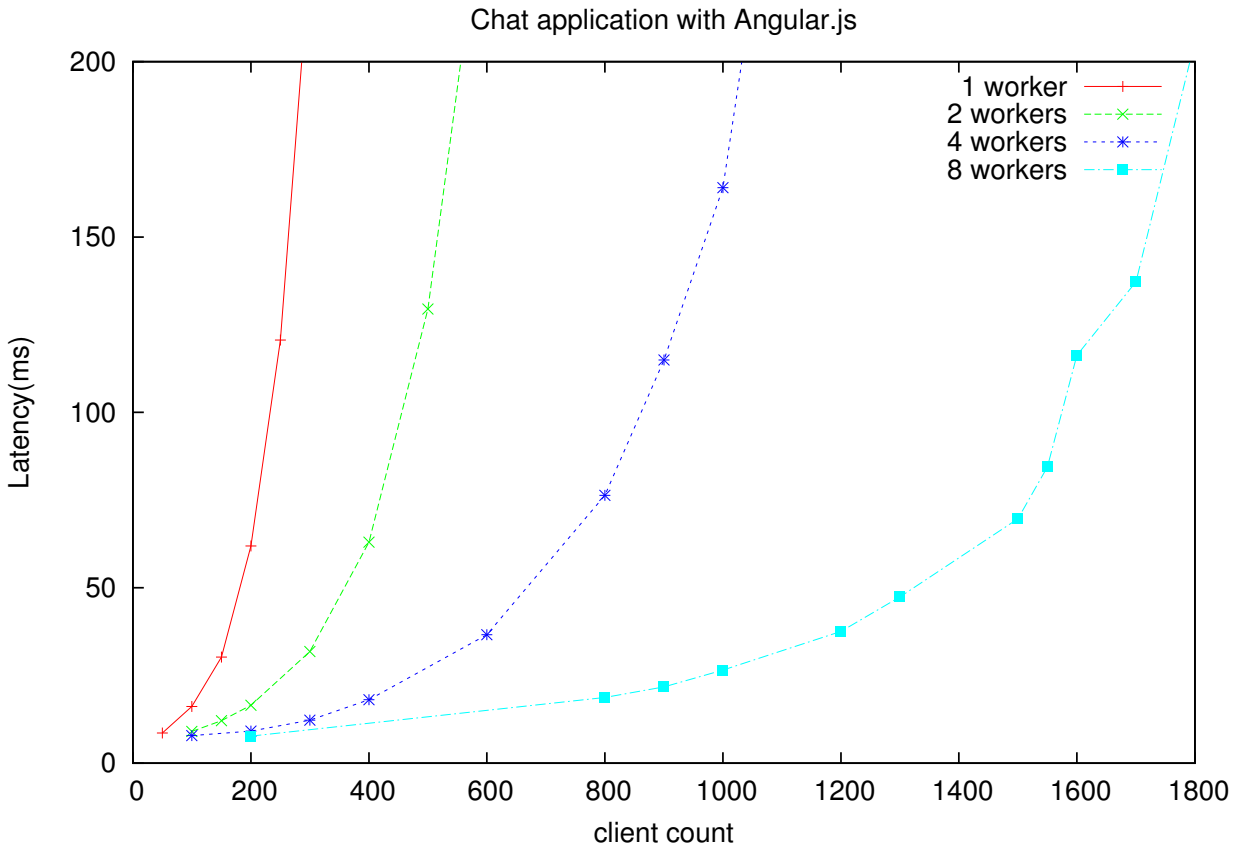

Figure 5.6: Latency of chat application with Angular.js.

less readable). Avoiding AngularJS's method of dirty checking to identify model changes roughly doubled performance, as shown in Figure 5.7. AngularJS's developers expect an order of magnitude speedup through direct VM support via `Object.observe()` in future JavaScript VMs [33].

A second example of a disproportionate impact of specific implementation limitations is the handling of style attributes. For instance, jQuery's `toggle()` method changes an element's visibility by changing a style attribute. To obtain the current value, it uses the `getComputedStyle()` function, which triggers the application of CSS selectors. This is a well- known performance bottleneck [29] in browser layout engines and not optimized at all in jsdom. A third example we encountered is the original implementation of a frequently

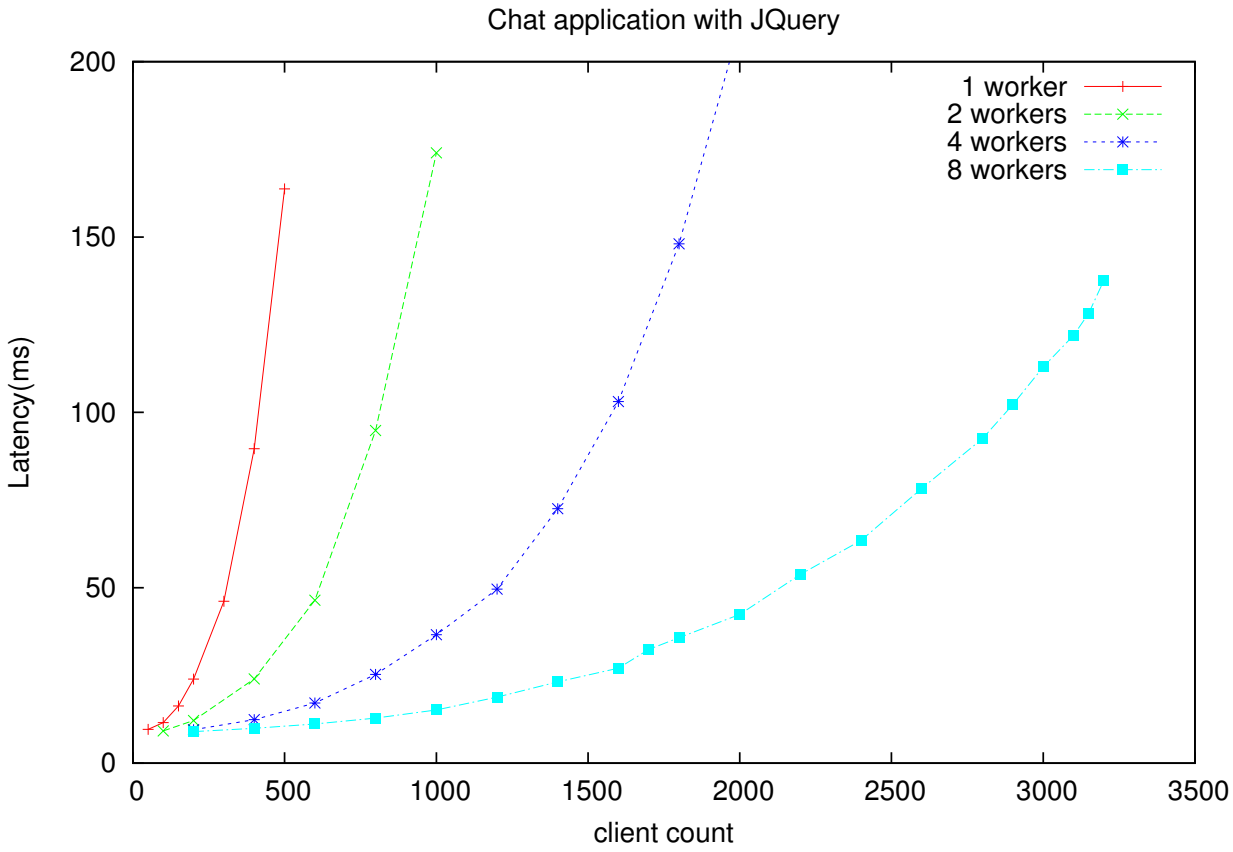

Figure 5.7: Latency of chat application with JQuery.

called function `lengthFromProperties` which determines the number of children of a DOM node. Profiling showed 30% of CPU spent in this function, which disappeared after an upgrade that substituted a constant time $O(1)$ implementation.

Chapter 6

Related Work

Our work on CloudBrowser bears a relationship to several related works which this chapter discusses, which include remote procedure call frameworks related to nodermi, as well as web frameworks and thin-client systems that share similar features or goals with CloudBrowser.

6.1 Remote Procedure Call Frameworks

In this section we discuss RPC frameworks that present remote method invocations, particularly those that do so using an object-oriented style. We are interested in how these frameworks abstract the remote method invocations, the effort that these frameworks require developers to make, and their special features and drawbacks.

Dnode [11] is a RPC framework for Node.js. Like in nodermi, remote method invocation is asynchronous and remote references are created automatically as a result of remote method calls. However, dnode does not distinguish remote references from local objects when passing them as arguments of remote methods. As a consequence, dnode may create a remote reference representing another remote reference, causing remote method calls to result in long chains of inter-process communications (i.e., the "zig-zag" problem discussed in Section 3.1).

Pyro [10] is an object-oriented RPC framework for the Python programming language. Unlike nodermi and dnode, Pyro synchronizes property modifications between remote references and their source objects. This makes remote references act more like regular references. But if used unwisely, remote property access can slow down the program. There are two major drawbacks for pyro : First, pyro does not automatically create remote references for objects passed to remote methods, programmers need to manually register these objects. If an object is not registered, it is recreated as a `dict` structure (a hash map equivalent data structure in Python) in the remote process, which discards all of its methods. Since remote method calls look like local method calls, it is tricky for the programmers to identify whether an object needs to be registered or not. Second, pyro lacks a garbage collection mechanism, as a result, objects passed to remote methods are kept alive as long the service is running.

Druby(drb) [42] is an object oriented RPC framework that is part of Ruby's standard library. Like pyro, it is the programmer's responsibility to explicitly tell the framework that an object needs to be passed as a remote reference. It also requires users to retain references to objects that are remotely referenced to prevent these objects from being prematurely garbage collected. By contrast, nodermi automatically creates remote references and takes care of distributed garbage collection.

Network objects [3] is a distributed programming paradigm that provides RPC in an object-oriented style. Java RMI [24] is an object-oriented RPC framework for the Java programming language. Unlike the RPC frameworks discussed before, these two are designed for use with a statically typed language. Both require that programmers implement classes that inherit from some predefined interfaces in order to pass their objects by reference to remote processes. Nodermi, on the other hand, automatically infers whether an object needs to be passed by reference. The remote references in these systems do not have properties, so the properties of a remote object could only be accessed via remote method calls. In nodermi, although remote object's properties are only snapshots of the original properties, they can be read directly.

Pyro, druby, network objects and Java RMI all default to synchronous APIs for I/O and communication. Likewise, in these frameworks, remote method invocations block until the remote processes completes the execution of the original methods.

CORBA [45] is a RPC standard that facilitates implementation of cross-platform RPC systems. CORBA requires programmers to specify the remote object's interface using an interface description language (IDL) and provide IDL bindings for the implementation language, which is commonly generated by tool (IDL compiler). Nodermi generates remote stubs dynamically without IDL.

6.2 Web Frameworks

There is a large number of web frameworks that are designed to simplify the development of web applications. We select a few web frameworks that are particularly related to CloudBrowser in the way that they provide programming models other than the request/response programming paradigm used in traditional web frameworks. Following [8], we categorize web frameworks that keep representation state on the server as server-centric frameworks and web frameworks in which the server side is not aware of representation state as client-centric.

6.2.1 Server-centric Web Frameworks

ZK [8] 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. ZK does not maintain a representation of the server document across page reloads, which means that all presentation state must be tied to session or persistent state. To scale ZK to multiple processes or servers,

this session state must be replicated across servers using application servers' session replicate support (discussed in Section 2.1.1) which broadcast changes to session to all the servers.

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. By contrast, CloudBrowser uses identical, HTML-based documents on the client and the server.

ItsNat [40] 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.

JavaServer Faces (JSF) is a component-based web framework that is part of the Java Enterprise Edition Platform [24]. Developers use XML to define application views and the framework abstracts away server-client communication using actions and action listeners. Although application developers do not need to know JavaScript, component developers need to use JavaScript to write interactive AJAX components. To create customized user interfaces that override the default behaviors of existing components, developers do need to understand JavaScript, CSS, and HTML as well as have JSF-specific knowledge about the request processing life-cycle, the details of component rendering, and others.

6.2.2 Client-centric Web Frameworks

Google Web Toolkit [18] 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.

A closely related project that pursues similar goals in simplifying the development of rich web

applications is Meteor [28], a full stack platform for building web and mobile applications. Meteor provides mechanisms that tie a client's presentation state directly to model state that is kept in a server-side database, which in turn is partially replicated on the client. As such, user input can be handled, optimistically, before the server roundtrip has completed. If an update is rejected by the server, the optimistic application is undone. Asynchronous updates to model data is pushed to clients. Like other client-centric frameworks, no presentation state is kept on the server.

Meteor will have lower server-side cost, and thus higher scalability, than CloudBrowser, as well as the ability to reduce user-perceived latency because of its optimistic processing. However, we believe that this comes at a high price of complexity and increased risk. For instance, programmers must be extremely careful to not leak sensitive data to the client, and they may easily expose (unintentionally) sensitive business logic to the client. Meteor also does not focus on the problem of simplifying the retention of presentation state across visits, either making that state ephemeral, or requiring the programmer use session state or store all necessary model variables in the database.

Moreover, because Meteor is integrated with its template engine, it does not play well with third party JavaScript libraries. This situation is getting better as more and more third party libraries get official integration support from Meteor and unofficial integration solutions from the community.

6.3 Thin-client Systems

Our system shares ideas with traditional thin-client and remote display systems such as VNC [39], Windows remote desktop, Citrix XenDesktop, and "dumb terminals" based on the X Window System [41]. We note that VMWare's Horizon product provides a way to see the user interface of actual virtual machines via a browser, with its BLAST protocol. 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.

Mosh [47] is a remote terminal application that synchronizes client screen states with server side terminal emulators. Because terminal emulators are text based, the cost of maintaining screen states is not significant. Mosh is able to provide prompt feedbacks to users in spite of network latency by guessing the effects of users' keystrokes and applying the guessed effects on users' screens immediately. Unlike CloudBrowser which preserves the entire user interface state of an applications, Mosh only keeps the current screen's state. Another difference from our system is that Mosh does not support reconnecting to existing sessions, the users are presented with a new window when they switch devices.

Chapter 7

Conclusion

7.1 Future Work

We plan to continue to optimize our system by incorporating future updates of the JavaScript language, JavaScript runtime engine and third party dependencies. The upcoming new ECMAScript 7 standard promises a number of useful features, such as native support for observing changes to objects [13]. A preliminary evaluation showed that dirty checking in AngularJS becomes 20 to 40 times faster using this feature [33], greatly improving our system's capacity to host AngularJS applications.

We also plan to implement a failover mechanism for worker processes. In the current implementation, when a worker process terminates, the master still dispatches requests to that worker. We need to implement a failure detection mechanism to detect failed workers and a method to migrate the failed workers' App Instances and virtual browsers to living workers.

Finally, as a PaaS platform we need a better admin interface to enable developers to monitor and diagnose their applications. Currently, the admin interface displays only very basic information for deployed applications. We need to expose application logs and performance related statistics to application developers.

7.2 Summary

Maintaining the rich presentation state of a web application server-side has many potential benefits, ranging from simplified application development, improved user experience, to increased security. There is strong interest, primarily outside of academia, in the development of platforms that simplify cloud-based application development.

We have prototyped and extensively tested this approach, focusing on a scalable multiprocess version of our CloudBrowser platform. The key limitations we encountered were not aspects of systems architecture or design, but the overhead cost of running rich frameworks, primarily in terms of CPU time. As these costs shift with the further refinement of these frameworks, we expect that the use of this approach become more realistic.

Bibliography

[1] Apple Inc. Connect your iPhone, iPad, iPod touch, and Mac using Continuity. `http://support.apple.com/en-us/HT6337`, 2014. [Online; accessed 29-November-2014].

[2] Tim Berners-Lee. The world wide web: A very short personal history. *Tim Berners-Lee*, 7, 1998.

[3] Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber. Network objects. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, pages 217–230, Asheville, NC, USA, 1993.

[4] Andrew D Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems (TOCS)*, 2(1):39–59, 1984.

[5] Bert Bos, Tantek elik, Ian Hickson, and Hkon Wium Lie. Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification. `http://www.w3.org/TR/CSS21/`, 2015. [Online; accessed 19-March-2015].

[6] Thomas Brisco. Dns support for load balancing. `http://tools.ietf.org/html/rfc1794`, 1995. [Online; accessed 08-April-2015].

[7] Valeria Cardellini, Emiliano Casalicchio, Michele Colajanni, and Philip S Yu. The state of the art in locally distributed web-server systems. *ACM Computing Surveys (CSUR)*, 34(2):263–311, 2002.

[8] Henri Chen and Robbie Cheng. *ZK: Ajax without the Javascript Framework*. Apress, Berkely, CA, USA, 2007.

[9] Douglas Crockford. Private Members in JavaScript. `http://javascript.crockford.com/private.html`. [Online; accessed 21-April-2015].

[10] Irmen de Jong. Pyro - Python Remote Objects. `http://pythonhosted.org/Pyro4/`, 2015. [Online; accessed 23-February-2015].

[11] dnode. `http://substack.net/dnode`, 2015. [Online; accessed 23-February-2015].

[12] ECMA ECMAScript, European Computer Manufacturers Association, et al. Ecmascript language specification, 2011.

[13] ECMAScript Object.observe spec proposal. `http://arv.github.io/ecmascript-object-observe/`, 2015. [Online; accessed 22-March-2015].

[14] Ian Fette and Alexey Melnikov. The websocket protocol. `http://tools.ietf.org/html/rfc6455`, 2011. [Online; accessed 21-March-2015].

[15] Roy Fielding and Julian Reschke. Hypertext transfer protocol (HTTP/1.1): Semantics and content, 2014.

[16] Brad Fitzpatrick. Distributed caching with memcached. *Linux journal*, 2004(124):5, 2004.

[17] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[18] Google, Inc. Google Web Toolkit (GWT). `http://code.google.com/webtoolkit/`.

[19] Filip Hanik and Peter Rossbach. Apache Tomcat 6.0 (6.0.43) - Clustering/Session Replication HOW-TO. `http://tomcat.apache.org/tomcat-6.0-doc/cluster-howto.html`, 2015. [Online; accessed 20-March-2015].

[20] Dick Hardt. The oauth 2.0 authorization framework. `http://tools.ietf.org/html/rfc6749.html`, 2012. [Online; accessed 07-April-2015].

[21] Misko Hevery. Building web apps with angular, 2009.

[22] Ian Hickson. Html living standard. web hypertext application technology working group (whatwg), 2012.

[23] Elijah Insua. JSDOM. `http://jsdom.org`.

[24] Sun Java System Application Server Standard and Enterprise Edition 7 2004Q2 Developer's Guide to Web Applications. `http://docs.oracle.com/cd/E19644-01/817-5451/index.html`, 2014. [Online; accessed 29-November-2014].

[25] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[26] Brian McDaniel and Godmar Back. The CloudBrowser web application framework. In *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*, pages 141–156. ACM, 2012.

[27] Caolan McMahon. caolan/async GitHub. `https://github.com/caolan/async`, 2015. [Online; accessed 10-April-2015].

[28] Meteor. `https://www.meteor.com/`, 2015. [Online; accessed 19-January-2015].

[29] Leo A. Meyerovich and Rastislav Bodik. Fast and parallel webpage layout. In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 711–720, Raleigh, NC, USA, 2010.

[30] Jakob Nielsen. *Usability Engineering*. Morgan Kaufmann, 1st edition, September 1993.

[31] node http proxy. nodejitsu/node-http-proxy GitHub. `https://github.com/nodejitsu/node-http-proxy`, 2014. [Online; accessed 29-November-2014].

[32] TooTallNate/node-weak. `https://github.com/TooTallNate/node-weak`, 2015. [Online; accessed 16-February-2015].

[33] Object.observe() and AngularJS. `https://mail.mozilla.org/pipermail/es-discuss/2012-September/024978.html`, 2015. [Online; accessed 03-February-2015].

[34] Xiaozhong Pan. nodermi. `https://github.com/bladepan/nodermi`, 2014. [Online; accessed 29-November-2014].

[35] PHP Manual. `http://php.net/manual/en/index.php`, 2014. [Online; accessed 29-November-2014].

[36] Protocol Buffers - Google Developers. `https://developers.google.com/protocol-buffers/`, 2014. [Online; accessed 1-March-2015].

[37] redis. `http://redis.io/`, 2015. [Online; accessed 09-April-2015].

[38] Will Reese. Nginx: the high-performance web server and reverse proxy. *Linux J.*, 2008(173), September 2008.

[39] Tristan Richardson, Quentin Stafford-Fraser, Kenneth R Wood, and Andy Hopper. Virtual network computing. *Internet Computing, IEEE*, 2(1):33–38, 1998.

[40] Jose Maria Arranz Santamaria. ItsNat: Natural AJAX. component based Java web application framework. `http://itsnat.sourceforge.net`.

[41] Robert W. Scheifler and Jim Gettys. The X window system. *ACM Trans. Graph.*, 5(2):79–109, April 1986.

[42] Masatoshi Seki. Ruby 2.2.0 Standard Library Documentation. `http://ruby-doc.org/stdlib-2.2.0/libdoc/drb/rdoc/DRb.html`, 2015. [Online; accessed 23-February-2015].

[43] Stefan Tilkov and Steve Vinoski. Node. js: Using javascript to build high-performance network programs. *IEEE Internet Computing*, 14(6):0080–83, 2010.

[44] V8 JavaScript Engine. `https://code.google.com/p/v8/`, 2014. [Online; accessed 11-March-2015].

[45] Steve Vinoski. Corba: integrating diverse applications within distributed heterogeneous environments. *Communications Magazine, IEEE*, 35(2):46–55, 1997.

[46] Chris Wilson and Vidur Apparao. Document object model (dom) level 2 style specification. *W3C Recommendation. November*, 2000.

[47] Keith Winstein and Hari Balakrishnan. Mosh: An interactive remote shell for mobile clients. In *USENIX Annual Technical Conference*, pages 177–182, 2012.