Rich Cloud-based Web Applications with CloudBrowser 2.0

ABSTRACT

Designing modern web applications involves a wide spectrum of choices when it comes to deciding where the different tiers of application and framework code that constitute these distributed applications should be placed. These system design choices affect programmer productivity, ease of deployment, security, and performance, particularly with respect to latency and scalability.

In this paper, we propose and evaluate a design choice in which not only all application logic executes server-side, but most presentation logic as well. The client browser is reduced to a rendering and I/O engine, similar to a "thin client" or "dumb terminal," but retains the full expressiveness of rich, modern Internet applications.

We have developed CloudBrowser 2.0, a system that implements this distribution model using a scalable multiprocess approach. In this paper, we perform an evaluation of the benefits and costs of this approach when compared to both more traditional approaches as well as emerging alternatives. We focus on programmability and systems aspects including performance and latency.

1. INTRODUCTION

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.

This paper explores an alternative design 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 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-viewcontroller (MVC) frameworks such as AngularJS [4] 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 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.

We first introduced the idea of HTML-based server-centric execution in [reference elided]. In CloudBrowser 2.0, we provide the first multi-process implementation of the CloudBrowser concept which enables its use on multiprocessor machines and on clusters. We have also investigated and implemented several features commonly expected from a framework, such as authentication and administration. CloudBrowser 2.0 also includes a model for the deployment of applications and introduces the concept of an application instance, which allows multiple virtual browsers to share state. We have implemented a number of sample and benchmark applications and profiled them to better understand the intrinsic and extrinsic limitations of this design.

2. MOTIVATION

With the emergence of AJAX as a technique over ten years ago a shift commenced from traditional, page-based applications to the single-page applications in use today. Although hybrid forms have been emerging recently, architectural frameworks for these applications largely falls into two groups: client-centric and server-centric. Client-centric approaches focused on the creation of powerful JavaScript libraries that execute in the browser and which provide high-level abstractions such as two-way data bindings and custom directives to manage UI state.

However, the user interface programmer must decide when to retrieve the data from the server and when to save/update the server's state. Any type of "server-push" must be explicitly programmed. Moreover, such applications rarely remember presentation state across visits, and when they do, they do so using storage facilities (such as HTML5's `localstorage`) that is not available on other devices on which a user might use the application.

By contrast, server-centric approaches represent the user interface on the server. In a server-centric approach, the application programmer writes only code that executes on the server, whereas all client-side code is framework-provided code. A pioneering framework in this area was ZK [2]. This simplified model, along with having to maintain a single and much smaller codebase, is tied to significant productivity gains. Unfortunately, even in this approach, the user interface representation is recreated every time a user navigates to a page. If programmers wish to preserve UI state across visits, they have to represent it in session state or in persistent state such as a database, and retrieve it from there every time the view is rebuilt.

Our own experience includes the development of two substantial applications using a server-centric approach: [references elided] an application to allow students to develop ZK applications in the cloud, and a complex configuration management system. This experience convinced us of the feasibility and value of a server-centric approach. In particular, our users reported enjoying the user interface style we were able to present, in which their actions took immediate effect [reference elided].Notably, the latency introduced by server round trips did not present a barrier to usability.

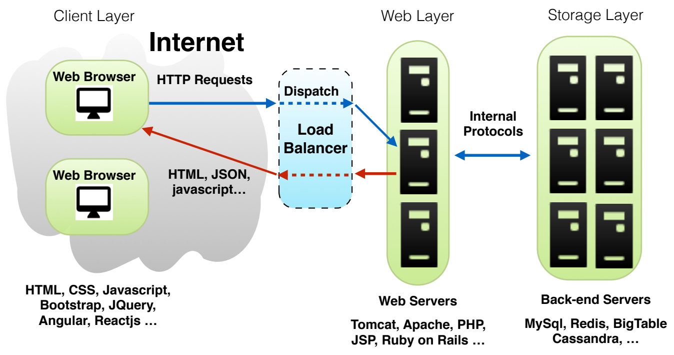


Figure 2: Traditional Scalable Multitier Application Design

We soon felt hampered by the lack of persistence across visits and no obvious way to solve it, other than to tie all variables controlling the application's view to model variables that were, in turn, stored persistently in some way. We hesitated to do that since we did not want to pollute our relatively clean model that represented the business data of our application with less critical state solely intended to improve usability.

A second limitation that influenced our thinking was the mismatch between server side representation (XUL) and the HTML used in the browser, which meant that debugging layout issues became difficult since it required understanding or reverse engineering how ZK's components were implemented. Thus born was the idea to keep the presentation state in HTML on the server.

Keeping presentation state on the server imposes a cost that not all types of applications may be willing to pay, even when the reduced development effort is taken into account. In (reference elided), we discuss the types of applications for which we believe the trade-off we propose may be worthwhile: personal and business applications such as tax preparation systems or configuration management systems. These applications are characterized by the following traits:

- They are single-page applications with a large navigation space in which the user has free reign, thus remembering the exact view state is crucial for a satisfactory user experience.
- Users expect that their actions take immediate effect, that is, almost every user input must trigger a communication with the server.
- Users expect that updates to application state done by others become immediately reflected in their UI.

2.1 Challenges

Successfully applying the idea of preserving presentation state using virtual browsers in server memory imposes a number of challenges which this work makes an attempt

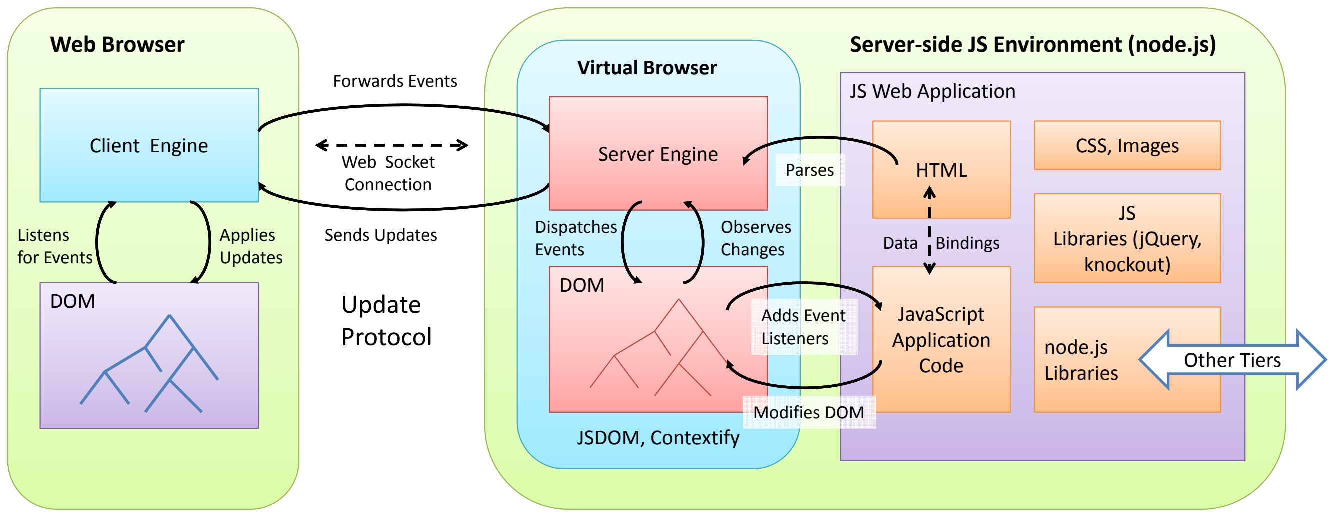


Figure 1: Single Process CloudBrowser Architecture Overview

to better understand. These are related to user interface implementation, latency, resource consumption, scalability, and behavior in the presence of faults.

A virtual browser does not lay out HTML elements or compute their styles in the way that the rendering engine in the actual browser will, as doing so would be cost-prohibitive and not help at all when the user switches devices. As such, it cannot support inquiries by code about computed or user-determined attributes such as what is the screen size, or what is the height of an element after its layout has been computed. Recent trends in web development have made this much less of a problem; for instance, in applications that use responsive design (e.g. the Bootstrap library), programmers are discouraged from directly inquiring about computed style attributes.

Secondly, since events are dispatched on the server, the delay of a wide area network roundtrip is added to the processing delay, along with any request queuing that may occur. According to the human computer interface literature, the resulting latency must not be larger than 100-150ms for the user to be perceived negatively. We treat this as a cut-off for the feasibility of a server-centric design.

A third concern is resource consumption, in particular memory and CPU consumption. Virtual browsers will consume memory while they exist, and consume CPU time while handling events. The use of high-level JavaScript libraries exacerbates this effect because they are often designed for use on powerful client machines, although optimizations in terms of CPU and memory for use on less powerful mobile devices helps us.

A fourth concern which this paper focuses on, is how to implement scalability across multiple processes and/or machines. Related to that is the behavior of the system if server processes fail or must be restarted for other reasons. Fig. 2 shows a canonical architecture used for scalable, multi-tier web applications, of which numerous variations exist. Scalability is achieved by having a load balancer spread individual HTTP requests across multiple web or application servers, which in turn talk to databases or other storage providers. In this model, the load balancer must be able to dispatch requests to any web server as web servers can be added and removed at will. To handle a request, a web server needs to have access to session state, which is replicated and kept consistent across the web servers. Though session state may be cached in a web server's memory, it is stored in a way that allows for server processes to restart. At the same time, loss of session state is not catastrophic as essential user data is stored in a separate storage layer.

A scalable virtual browser environment can be implemented in similar fashion; however, here the load balancer is restricted in the choice of web server to which to forward requests as soon as a server process is chosen on which the virtual browser is allocated. A crash or restart of such a process leads to the loss of all presentation state. To prevent this from becoming catastrophic, applications developers must, as before, decide which part of their state is deemed essential data that must be stored in the separate storage layer. This can be done by accessing existing storage layers, but we also added a facility that allows applications to save a snapshot of their state and resume from it later.

3. DESIGN AND IMPLEMENTATION

Full details of a single process implementation of the CloudBrowser architecture are available in [reference elided]We highlight only the essential design here. Figure 1 shows the relationship between the client engine running in the user's browser and the virtual browser running server side. When the user visits the application, the client engine code is downloaded and 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 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 web socket communication. Stylesheets, images, etc. are provided to the client through a resource proxy.

3.1 Application Deployment Model

We designed an application deployment model for CloudBrowser 2.0 that allows different instantiation strategies. This deployment model addresses a problem that traditional web applications so far have not faced, which is to manage the lifetime of virtual browsers as users create them, connect to them, and disconnect from them. The underlying goal is to minimize the level of awareness on both the side of the application developer and the applications' users.

Application programmers can create CloudBrowser applications in the same way in which they create the client-side portion of a client-centric application, using low or high level JavaScript libraries such as jQuery [5] or AngularJS. A descriptor in the application's manifest describes their application's required instantiation strategy. When a user instantiates an application, the allocated application instance object represents metadata about this application instance, such as ownership and access permissions, along with application data that the application's code can directly access.

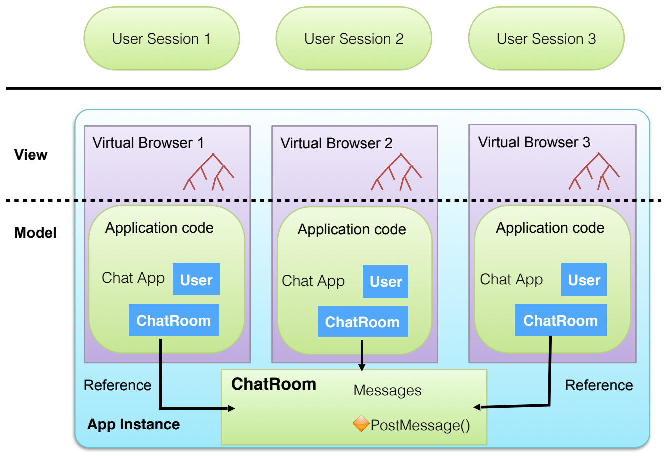


Figure 3: This figure shows how multiple virtual browsers can directly, and seamlessly share relevant application data, in this case chat messages, which then become part of the model that drives the presentation MVC framework.

As an example, consider a scenario for a Chat application developed using AngularJS, depicted in Figure 3. A system administrator of a CloudBrowser deployment would install the application, which give users the ability to create application instances. To start a chat site, a user would create an application instance and share its URL with chat participants. As the participants join the chat site, a virtual browser is created on demand for each participant, which is connected to the application instance (the users can bookmark their virtual browser's URL to later return.) The shared application instance data in such an application consist of the chatroom(s), users and their associated messages.

The hierarchy that results from applications, application instances, and virtual browsers is depicted in Figure 4. This

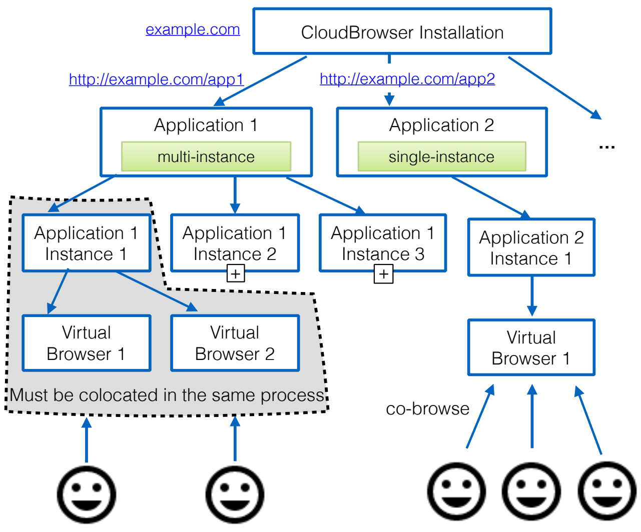


Figure 4: Hierarchy of applications, application instances, and virtual browsers. Note that a single virtual browser may be broadcast to multiple clients (cobrowsing).

figure shows the general case in which an application might allow multiple instances, and in which each user can create multiple virtual browsers. We also found it useful to support more specialized instantiation modes, which are related to the user's authentication state. These include *singleAppInstance*, *singleUserInstance*, and *multiInstance*, and correspond to a single instance for all users (as in a traditional webpage), a single instance per authenticated user, and multiple instances per user. Only the last instantiation mode exposes users to a management interface for virtual browsers that allows them to create new virtual browsers, or reconnect to existing ones when the connect/disconnect from the web applications.

3.2 Distributed Implementation

Figure 5 shows the distributed design of CloudBrowser 2.0, which consists of a single master process, multiple reverse proxies, and multiple worker processes. The number of workers is determined by the number of CPUs available, due to the event-based (nonblocking) implementation of the node.js platform on which CloudBrowser is implemented. All processes communicate via standard TCP/IP sockets, so they can be located on a shared multiprocessor machine or in a cluster.

The master process maintains a table of which application instances have been created and which worker is responsible for which application instance. When a new application instance is created, the master will apply a load balancing algorithm to decide on which worker to place this instance. We support two load balancing strategies: first, the master can assign application instances to workers in a simple round-robin fashion. However, since application instances may vary widely in terms of the actual cost they impose on a worker, we also implemented a load-based scheme in which workers periodically report a measure of current load to the master.

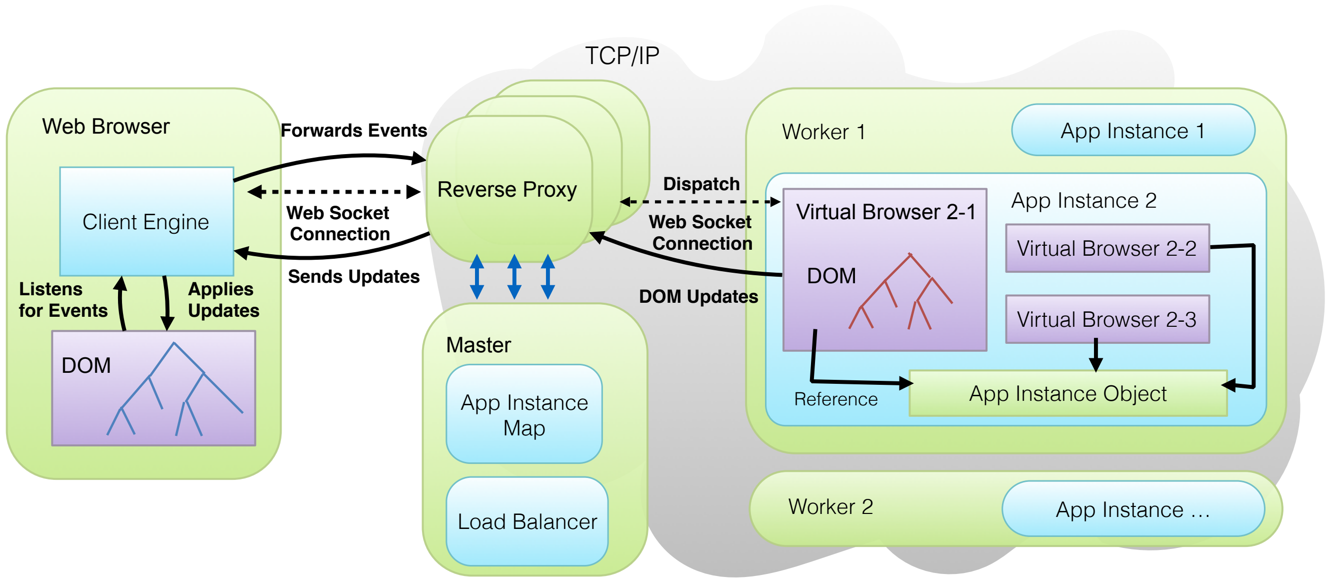


Figure 5: Multiprocess Architecture Overview

Multiple 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. When a proxy process accepts a client, it parses the incoming HTTP request to extract the application id, which is part of the request's URL. Each proxy process maintains a table of known mappings from application ids to workers. If the requested id is already cached in that table, the request is directly relayed to the worker. Otherwise, the master is contacted to find out which worker is assigned to that application id, triggering an initial assignment if necessary. The reverse proxy can relay both HTTP requests/responses as well as the bidirectional web socket protocol after the connection has been upgraded. Once established, the majority of traffic will be web socket messages for which there is relatively little per-message overhead.

3.3 Interprocess Communication

Masters, proxies, and workers need to be able communicate with one another. This communication takes places at the level of JavaScript method calls. To facilitate the invocation of JavaScript methods that belong to objects residing in other processes, we developed `nodermi`, a remote method invocation (RMI) library [reference elided].

`nodermi`uses JavaScript's reflection facilities to transparently create client-side stubs for remote objects on the fly. Since JavaScript does not use static typing and heavily uses variadic functions, it serializes method arguments dynamically via reflection. Our system supports all primitive JavaScript types, objects, and arrays; it can handle cyclic object graphs. It also recognizes when arguments refer to remote objects, in which case the receiving side will create its own stub that directly refers to the remote object. If an argument is a function, a stub will be created on the remote side that, when invoked, will make a RPC request back to the requestor.

4. 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.

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. Like a (patient) human user, the script will not send the next event until after the previous event resulted in the expect DOM update. To simulate human processing speed, a programmable "thinking delay" can be introduced between receiving an expected response and sending the next event. To create test scenarios from actual applications, we run those applications using a real browser and record the client/server interactions in a log, which our test tools can reply.

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 proxies. We colocate the first 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 static load balancing strategy discussed in Section 3.2 to ensure an exactly predictable distribution of application instances onto workers.

4.1 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 error below 5%, the maximum relative standard error was 11%.

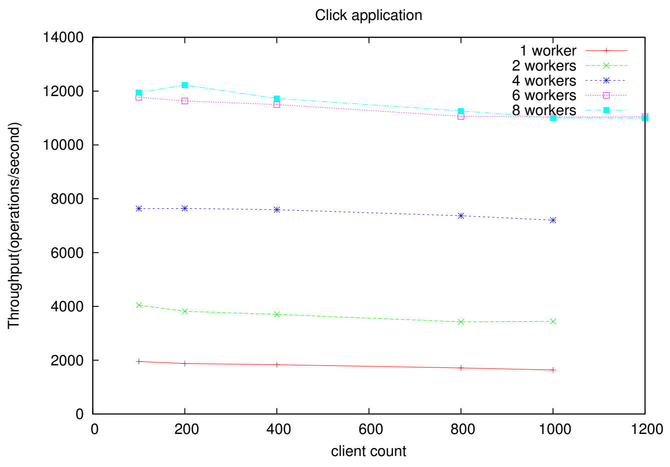


Figure 6: Throughput of "Back-to-back" click application.

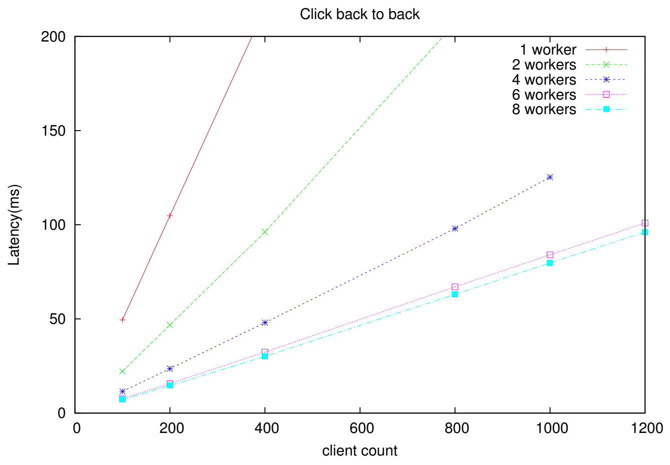


Figure 7: Latency of "Back-to-back" click application.

We measure throughput in terms of operations per second and average latency. Figure 6 shows the throughput of the application for different numbers of workers. In those cases, all 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 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 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.

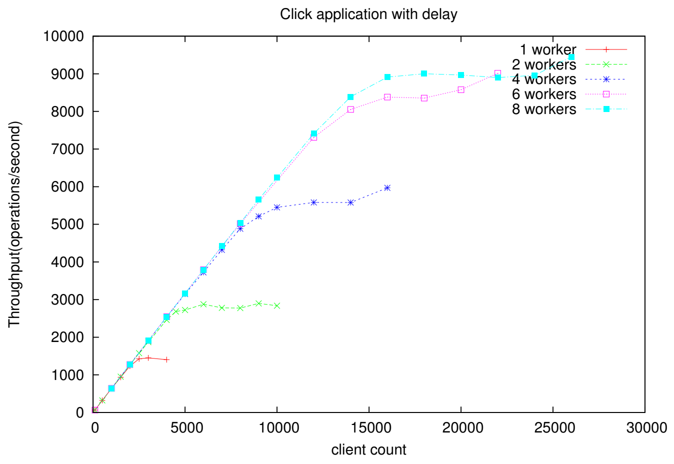


Figure 8: Throughput of click application, after introducing artificial delay.

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 7. Note that we consider a latency of more than 100ms unacceptable [7], 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 8 and 9. 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 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.

4.2 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

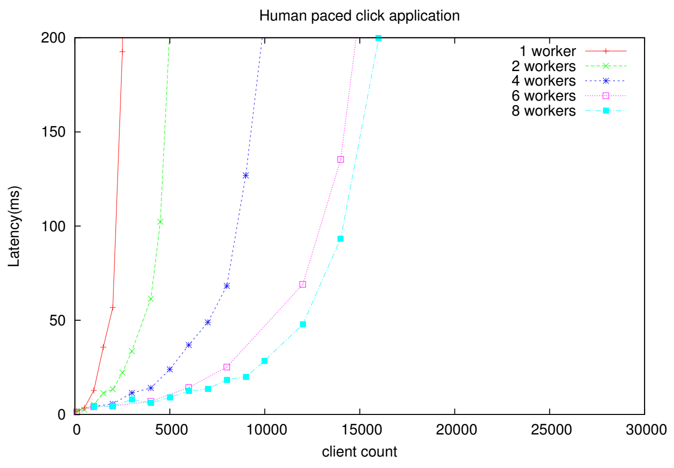
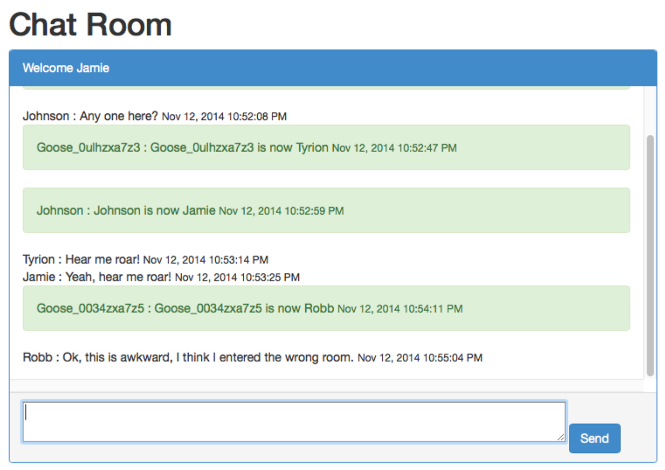


Figure 9: Latency of click application, after introducing artificial delay.



Chat Room

Welcome Jamie

Johnson : Any one here? Nov 12, 2014 10:52:08 PM

Goose_0ulhzxa7z3 : Goose_0ulhzxa7z3 is now Tyrion Nov 12, 2014 10:52:47 PM

Johnson : Johnson is now Jamie Nov 12, 2014 10:52:59 PM

Tyrion : Hear me roar! Nov 12, 2014 10:53:14 PM
Jamie : Yeah, hear me roar! Nov 12, 2014 10:53:25 PM

Goose_0034zxa7z5 : Goose_0034zxa7z5 is now Robb Nov 12, 2014 10:54:11 PM

Robb : Ok, this is awkward, I think I entered the wrong room. Nov 12, 2014 10:55:04 PM

Figure 10: Chat Room Application

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 investigate the overhead of this usage scenario, a screenshot is shown in Figure 3. 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 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.

Fig.11 shows the latency perceived by benchmark tool under different numbers of clients. While scaling to multiple workers remains effective, the use of AngularJS imposes a

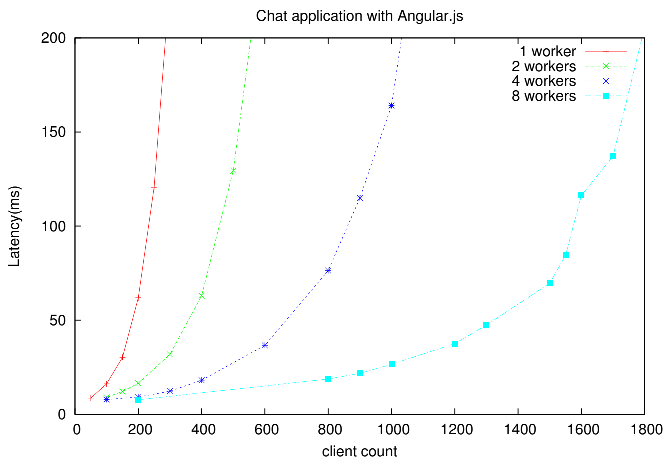


Figure 11: Latency of chat application with Angular.js.

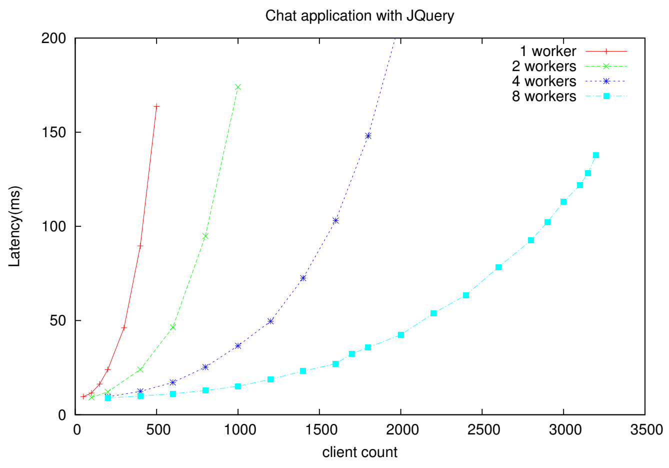


Figure 12: Latency of chat application with JQuery.

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 reexecution 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 less readable). Avoiding AngularJS's method of dirty checking to identify model changes roughly doubled performance, as shown in Figure 12. AngularJS's developers expect an order of magnitude speedup through direct VM support via `Object.observe()` in future JavaScript VMs [8].

5. RELATED WORK

ZK [2], which we use as part of our motivation in Section 2 is a Java-based server-centric web framework that is in wide use. ZK applications are constructed using components, which are represented using the ZK User Interface Markup Language (ZUML). ZUML components are translated into HTML and CSS when a page is rendered. A client-side library handles synchronization between the client's view of and interaction with components and their server-side representation. Our extensive experience deploying applications with ZK ([references elided]) inspired the work on CloudBrowser. As discussed, 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.

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 [9] is a Java-based AJAX component framework similar to ZK, although it uses HTML instead of ZUML to express server documents, along with the Java W3C implementation. Unlike CloudBrowser, it also does not maintain the server document state across visits, and cannot make use of existing JavaScript libraries.

The Google Web Toolkit [3] 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 [6], 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. Like with client-centric frameworks, no presentation state is kept on the server. Asynchronous updates to model data is pushed to clients.

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.

Lastly, our system shares ideas with traditional thin-client and remote display systems, going back to "dumb terminals" based on the X Window System [10]. 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.

6. CONCLUSION

This paper discusses a scalable, multiprocess implementation of web application framework that maintains rich presentation state server-side. This approach has many advantages, from simplified and accelerated application development, improved user experience across visits without requiring programmer effort, and potentially increased security. As such, it has the potential to lead to applications that truly "live" in the cloud, rather than merely interactive web pages.

Our results show that this design can be implemented in a scalable way such that adding more resources results in a proportional increase in capacity. However, it also shows that the use of code and libraries originally intended for client-side use on the server can be very expensive, primarily in terms of CPU time. These limitations will narrow the scope of applications for which a pure server-centric approach such as ours is applicable.

7. ACKNOWLEDGEMENTS

We are removing funding acknowledgements for the purpose of maintaining anonymity for the double-blind review process. Our framework is available under an open source license on github.

8. REFERENCES

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

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

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

[4] Hevery, M. Building web apps with Angular, 2009.

[5] jQuery Foundation jquery.org. jQuery. `http://jquery.com/`, 2015. [Online; accessed 02-February-2015].

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

[7] Nielsen, J. *Usability Engineering*, 1st ed. Morgan Kaufmann, Sept. 1993.

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

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

[10] Scheifler, R. W., and Gettys, J. The X window system. *ACM Trans. Graph. 5*, 2 (Apr. 1986), 79–109.