# BenchLab: An Open Testbed for Realistic Benchmarking of Web Applications

Emmanuel Cecchet, Veena Udayabhanu, Timothy Wood, Prashant Shenoy
*University of Massachusetts Amherst*
{cecchet,veena,twood,shenoy}@cs.umass.edu

**Abstract**

Web applications have evolved from serving static content to dynamically generating Web pages. Web 2.0 applications include JavaScript and AJAX technologies that manage increasingly complex interactions between the client and the Web server. Traditional benchmarks rely on browser emulators that mimic the basic network functionality of real Web browsers but cannot emulate the more complex interactions. Moreover, experiments are typically conducted on LANs, which fail to capture real latencies perceived by users geographically distributed on the Internet. To address these issues, we propose BenchLab, an open testbed that uses real Web browsers to measure the performance of Web applications. We show why using real browsers is important for benchmarking modern Web applications such as Wikibooks and demonstrate geographically distributed load injection for modern Web applications.

## 1. Introduction

Over the past two decades, Web applications have evolved from serving primarily static content to complex Web 2.0 systems that support rich JavaScript and AJAX interactions on the client side and employ sophisticated architectures involving multiple tiers, geographic replication and geo-caching on the server end. From the server backend perspective, a number of Web application frameworks have emerged in recent years, such as Ruby on Rails, Python Django and PHP Cake, that seek to simplify application development. From a client perspective, Web applications now support rich client interactivity as well as customizations based on browser and device (e.g., laptop versus tablet versus smartphone). The emergence of cloud computing has only hastened these trends---today's cloud platforms (e.g., Platform-as-a-Service clouds such as Google AppEngine) support easy prototyping and deployment, along with advanced features such as autoscaling.

To fully exploit these trends, Web researchers and developers need access to modern tools and benchmarks to design, experiment, and enhance modern Web systems and applications. Over the years, a number of Web application benchmarks have been proposed for use by the community. For instance, the research community has relied on open-source benchmarks such as TPC-W [16] and RUBiS [13] for a number of years; however these benchmarks are outdated and do not fully capture the complexities of today's Web 2.0 applications and their workloads. To address this limitation, a number of new benchmarks have been proposed, such as TPC-E, SPECWeb2009 or SPECjEnterprise2010. However, the lack of open-source or freely available implementations of these benchmarks has meant that their use has been limited to commercial vendors. CloudStone [15] is a recently proposed open-source cloud/Web benchmark that addresses some of the above issues; it employs a modern Web 2.0 application architecture with load injectors relying on a Markov model to model user workloads. Cloudstone, however, does not capture or emulate client-side JavaScript or AJAX interactions, an important aspect of today's Web 2.0 applications and an aspect that has implications on the server-side load.

In this paper, we present BenchLab, an open testbed for realistic Web benchmarking that addresses the above drawbacks. BenchLab's server component employs modern Web 2.0 applications that represent different domains; currently supported server backends include Wikibooks (a component of Wikipedia) and CloudStone's Olio social calendaring application, with support for additional server applications planned in the near future. BenchLab exploits modern virtualization technology to package its server backends as virtual appliances, thereby simplifying the deployment and configuration of these server applications in laboratory clusters and on public cloud servers. BenchLab supports Web performance benchmarking "at scale" by leveraging modern public clouds---by using a number of cloud-based client instances, possibly in different geographic regions, to perform scalable load injection. Cloud-based load injection is cost-effective, since it does not require a large hardware infrastructure and also captures Internet round-trip times. In the design of BenchLab, we make the following contributions:

- We provide empirical results on the need to capture the behavior of real Web browsers during Web load injection. Our results show that traditional trace replay methods are no longer able to faithfully emulate modern workloads and exercise client and server-side functionality of modern Web applications. Based on this insight, we design BenchLab to use real Web browsers, in conjunction with automated tools, to inject Web requests to the server application. As noted above, we show that our load injection process can be scaled by leveraging inexpensive client instances on public cloud platforms.

- Similar to CloudStone's Rain [2], BenchLab provides a separation between the workload modeling/generation and the workload injection during benchmark execution. Like Rain, BenchLab supports the injection of real Web traces as well as synthetic ones generated from modeling Web user behavior. Unlike Rain, however, BenchLab uses real browsers to inject the requests in these traces to faithfully capture the behavior of real Web users.
- BenchLab is designed as an open platform for realistic benchmarking of modern Web applications using real Web browsers. It employs a modular architecture that is designed to support different backend server applications. We have made the source code for BenchLab available, while also providing virtual appliance versions of our server application and client tools for easy, quick deployment.

The rest of this document is structured as follows. Section 2 explains why realistic benchmarking is an important and challenging problem. Section 3 introduces BenchLab, our approach to realistic benchmarking based on real Web browsers. Section 4 describes our current implementation that is experimentally evaluated in section 5. We discuss related work in section 6 before concluding in section 7.

## 2. Why Realistic Benchmarking Matters

A realistic Web benchmark should capture, in some reasonable way, the behavior of modern Web applications as well as the behavior of end-users interacting with these applications. While benchmarks such as TPC-W or RUBiS were able to capture the realistic behavior of Web applications from the 1990s, the fast paced technological evolution towards Web 2.0 has quickly made these benchmarks obsolete. A modern Web benchmark should have realism along three key dimensions: (i) a realistic server-side application, (ii) a realistic Web workload generator that faithfully emulates user behavior, and (iii) a realistic workload injector that emulates the actual "browser experience." In this section, we describe the key issues that must be addressed in each of these three components when constructing a Web benchmark.

### 2.1. Realistic applications

The server-side component of the benchmark should consist of a Web application that can emulate common features of modern Web applications. These features include:

*Multi-tier architecture:* Web applications commonly use a multi-tier architecture comprising at least of a database backend tier, where persistent state is stored, and a front-end tier, where the application logic is implemented. In modern applications, this multi-tier architecture is often implemented in the form of a Model-View-Controller (MVC) architecture, reflecting a similar partitioning. A number of platforms are available to implement such multi-tier applications. These include traditional technologies such as JavaEE and PHP, as well as a number of newer Web development frameworks such as Ruby on Rails, Python Django and PHP Cake. Although we are less concerned about the idiosyncrasies of a particular platform in this work, we must nevertheless pay attention to issues such as the scaling behavior and server overheads imposed by a particular platform.

*Rich interactivity:* Regardless of the actual platform used to design them, modern Web applications make extensive use of JavaScript, AJAX and Flash to enable rich interactivity in the application. New HTML5 features confirm this trend. In addition to supporting a rich application interface, such applications may incorporate functionality such as "auto complete suggestions" where a list of completion choices is presented as a user types text in a dialog or a search box; the list is continuously updated as more text is typed by the user. Such functions require multiple round trip interactions between the browser and the server and have an implication on the server overheads.

*Scaling behavior:* To scale to a larger number of users, an application may incorporate techniques such as replication at each tier. Other common optimizations include use of caches such as memcached to accelerate and scale the serving of Web content. When deployed on platforms such as the cloud, it is even feasible to use functions like auto-scaling that can automatically provision new servers when the load on existing ones crosses a threshold.

*Domain:* Finally, the "vertical" domain of the application has a key impact on the nature of the server-side workload and application characteristics. For example, "social" Web applications incorporate different features and experience a different type of workload than say, Web applications in the financial and retail domains. Although it is not feasible for us to capture the idiosyncrasies of every domain, our open testbed is designed to support any application backend in any domain. We presently support two backends: Wikibooks [20] (a component of Wikipedia [21]) and CloudStone's Olio [12] social calendaring application, with support for additional server applications planned in the future.

### 2.2. Realistic load generation

Realistic load generation is an important part of a benchmark. The generated workload should capture real user behavior and user interactions with the application. There are two techniques to generate the workload for a benchmark. In the first case, we can use real workload data to seed or generate the workload for the benchmark; in the simplest case, the real workload is replayed during benchmark execution. The advantage of this approach is that it is able to capture real user behavior. However, real workload data may not always be available. Further, the data may represent a particular set of benchmark parameters and it is not always easy to change these parameters (e.g., number of concurrent users, fraction of read and write requests, etc) to suit the benchmarking needs. Consequently many benchmarks rely on synthetic workload generators. The generators model user behavior such as think times as well as page popularities and other workload characteristics. Cloudstone, for instance, uses a sophisticated Markov model to capture user behavior [15]. The advantage of synthetic workload generation is that it allows fine-grain control over the parameters that characterize the workload [8].

BenchLab does not include a custom Web workload generation component. Rather it is designed to work with any existing workload generator. This is done by decoupling the workload generation step from the workload injection. In many benchmarks, workload generation and injection are tightly coupled—a request is injected as soon as it is generated. BenchLab assumes that workload generation is done separately and the output is stored as a trace file. This trace data is then fed to the injection process for replay to the server application. This decoupling, which is also used by Rain [2], allows the flexibility of using real traces for workload injection (as we do for our Wikibooks backend) as well as the use of any sophisticated synthetic workload generator.

### 2.3. Realistic load injection

Traditionally Web workload injection has been performed using trace replay tools such as *httperf* that use one or a small number of machines to inject requests at a high rate to the application. The tools can also compute client-side statistics such as response time and latency of HTTP requests. This type of workload injection is convenient since it allows emulating hundreds of virtual users (sometimes even more) from a single machine, but it has limited use for many applications that adjust behavior based on a client's IP address. In some scenarios, such as testing real applications prior to production deployment, this can be problematic since many requests originating from the same IP address can trigger the DDoS detection mechanisms if any. More importantly, this approach does not realistically test IP-based localization services or IP-based load balancing.

An important limitation of trace replay-based techniques is that they fall short of reproducing real Web browser interactions as they do not execute JavaScript or perform AJAX interactions. As a result, they may even fail to generate requests that would be generated by a real browser. Even the typing speed in a text field can have an impact on the server load since each keystroke can generate a request to the server like with Google Instant. Such interactions are hard to capture using trace replay tools.

Modern applications also include browser-specific customizations; they may send out custom style sheets and custom JavaScript depending on the browser type. The same application may also send a vastly different version of a page to a mobile or a tablet browser than a traditional desktop-class browser.[1] Moreover, each browser has different optimizations to fetch the content of Web pages in parallel and to render them quickly. Thus, the browser mix can impact the load seen by the server applications, even for a fixed number of users.

Finally, the replay tools typically report the response time of individual requests, rather than page load times seen by a browser—typically a Web page can include tens of components, including style sheets, images, ads and others components, and the response time for a page should include the time to load and render all of these components from a browser standpoint.

To capture these subtleties, we argue for the use of real Web browsers to drive the load injection. This is achieved by using automated tools that interact with a browser UI like a real user would and to issue requests from the browser, using the traces generated by the workload generation process. Having a variety of real Web browsers with various configurations and plugins improves the accuracy of benchmarking the real user experience of a Web application.

## 3. BenchLab

BenchLab is an open testbed for Web application benchmarking. It can be used with any standard benchmark application as well as real Web applications (section 3.2). Applications can have multiple datasets and workloads (section 3.3), and load injection is performed by real Web browsers (section 3.4).

### 3.1. Overview

Figure 1 gives an overview of the BenchLab components and how they interact to run an experiment. The

[1] Typically web applications redirect users from mobile devices to a separate mobile version of the application. However some recent applications have embedded support for mobile browsers within the main application.

BenchLab WebApp is the central piece that controls experiments. It is a Java Web application that can be deployed in any Java Web container such as Apache Tomcat. The BenchLab WebApp provides a Web interface to interact with experimenters that want to manage experiments and automated Web browsers that are executing experiments.

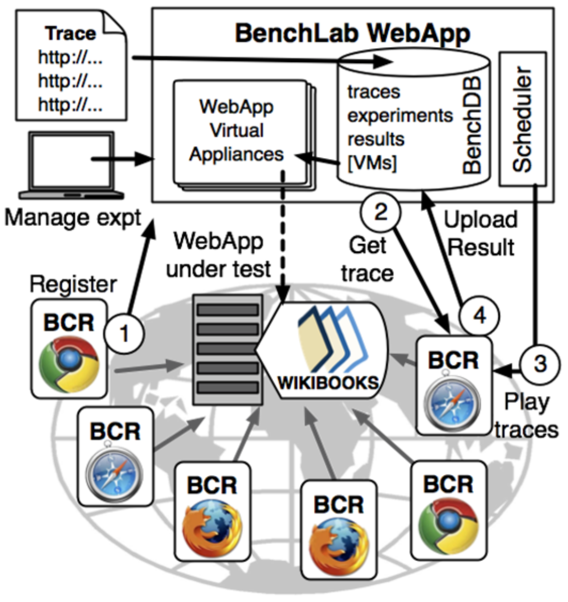


**Figure 1. BenchLab experiment flow overview.**

Web traces can be recorded from a live system or generated statically (see section 3.3). Trace files are uploaded by the experimenter through a Web form and stored in the BenchLab database. Virtual machines of the Web Application under test can also be archived so that traces, experiments and results can be matched with the correct software used. However BenchLab does not deploy, configure or monitor any server-side software. There are a number of deployment frameworks available that users can use depending on their preferences (Gush, WADF, JEE, .Net deployment service, etc) Server side monitoring is also the choice of the experimenter (Ganglia and fenxi are popular choices). It is the responsibility of the user to deploy the application to be tested. Note that anyone can deploy a BenchLab WebApp and therefore build his or her own benchmark repository.

An experiment defines what trace should be played and how. The user defines how many Web browsers and eventually which browsers (vendor, platform, version …) should replay the sessions. If the trace is not to be replayed on the server it was recorded, it is possible to remap the server name recorded in the URLs contained in the trace to point to another server that will be the target of the experiment.

The experiment can start as soon as enough browsers have registered to participate in the experiment or be scheduled to start at a specific time. The BenchLab WebApp does not deploy the application nor the client Web browsers, rather it waits for browsers to connect and its scheduler assigns them to experiments.

The BenchLab client runtime (BCR) is a small program that starts and controls a real Web browser on the client machine. The BCR can be started as part of the booting process of the operating system or started manually on-demand. The BCR connects the browser to a BenchLab WebApp (step 1 in Figure 1). When the browser connects to the WebApp, it provides details about the exact browser version and platform runtime it currently executes on as well as its IP address. If an experiment needs this browser, the WebApp redirects the browser to a download page where it automatically gets the trace for the session it needs to play (step 2 in Figure 1). The BCR stores the trace on the local disk and makes the Web browser regularly poll the WebApp to get the experiment start time. There is no communication or clock synchronization between clients, they just get a start time as a countdown in seconds from the BenchLab WebApp that informs them ‘experiment starts in x seconds’. The activity of Web browsers is recorded by the WebApp and stored in the database for monitoring purposes.

When the start time has been reached, the BCR plays the trace through the Web browser monitoring each interaction (step 3 in Figure 1). If Web forms have to be filled, the BCR uses the URL parameters stored in the trace to set the different fields, checkboxes, list selections, files to upload, etc. Text fields are replayed with a controllable rate that emulates human typing speed. The latency and details about the page are recorded (number of div sections, number of images, size of the page and title of the page) locally on the client machine. The results are uploaded to the BenchLab WebApp at the end of the experiment (step 4 in Figure 1).

Clients replay the trace based on the timestamps contained in the trace. If the client happens to be late compared to the original timestamp, it will try to catch up by playing requests as fast as it can. A global timeout can be set to limit the length of the experiment and an optional heartbeat can also be set. The heartbeat can be used for browsers to report their status to the BenchLab WebApp, or it can be used by the WebApp to notify browsers to abort an experiment.

## 3.2. Application backends

Our experience in developing and supporting the RUBiS benchmark for more than 10 years, has shown that users always struggle to setup the application and the different tools. This is a recurrent problem with benchmarks where more time is spent in installation and configuration rather than experimentation and measurement. To address this issue, we started to release RUBiSVA, a Virtual Appliance of RUBiS [13], i.e., a virtual machine with the software stack already configured and ready to use. The deployment can be automated on any platform that supports virtualization.

Virtualization is the de-facto technology for Web hosting and Web application deployment in the cloud. Therefore, we have prepared virtual appliances of standard benchmarks such as RUBiS, TPC-W [16] and CloudStone [15] for BenchLab. This allows reproducing experiments using the exact same execution environment and software configuration and will make it easier for researchers to distribute, reproduce and compare results.

As BenchLab aims at providing realistic applications and benchmarks, we have also made virtual appliances of Wikibooks [20]. Wikibooks provides a Web application with a similar structure to Wikipedia [21], but a more easily managed state size (GBs instead of TBs). More details about our Wikibooks application backend are provided in section 4.4.2.

## 3.3. Workload definitions

Most benchmarks generate the workload dynamically from a set of parameters defined by the experimenter such as number of users, mix of interactions, and arrival rate. Statistically, the use of the same parameters should lead to similar results between runs. In practice the randomness used to emulate users can lead to different requests to the Web application that require very different resources depending on the complexity of the operations or the size of the dataset that needs to be accessed. Consequently, the variance in the performance observed between experiments using the same parameters can be large. Therefore, it is necessary to decouple the request generation from the request execution so that the exact same set of requests can be replayed at will.

This can be done with little effort by instrumenting existing load generators and logging the requests that are made to the server. The resulting trace file can then be replayed by a generic tool like httperf or a more realistic injector like a real Web browser.

The traces used in BenchLab are based on the standard HTTP archive (HAR) format [9]. This format captures requests with their sub-requests, post parameters, cookies, headers, caching information and timestamps. Parameters include text to type in text fields, files to upload, boxes to check or buttons to click, etc. In the case of real applications, these traces can also be generated from an HTTP server access log to reproduce real workloads. As Web browsers automatically generate sub-requests to download the content of a page (cascading style sheets (.css), JavaScript code (.js), image files, etc), only main requests from the trace file are replayed.

Defining and generating workloads are beyond the scope of BenchLab. BenchLab focuses on the execution and replay of existing traces. Traces are stored in a database to be easily manipulated and distributed to injectors. Traces cannot be scaled up, they are either replayed completely or partially (a subset of the sessions in the trace). This means that if a trace contains 100 user sessions, it can be replayed by at most 100 clients. If a trace needs to be scaled, the user must use her workload generator to generate a scaled trace.

## 3.4. Web browser load injection

A central contribution of BenchLab is the ability to replay traces through real Web browsers. Major companies such as Google and Facebook already use new open source technologies like Selenium [14] to perform functional testing. These tools automate a browser to follow a script of actions, and they are primarily used for checking that a Web application's interactions generate valid HTML pages. We claim that the same technology can also be used for performance benchmarking.

BenchLab client runtime can be used with any Web browser supported by Selenium: Firefox, Internet Explorer, Chrome and Safari. Support for mobile phones with Webkit-based Web browsers is also under development. The functionality of BenchLab on the client side is limited to downloading a trace, replaying it, recording response times and uploading response times at the end of the replay. This small runtime is deployable even on devices with limited resources such as smartphones. Unlike commercial offerings, BenchLab clients can be deployed on public clouds or any other computing resource (e.g., desktop, smartphone).

Unlike traditional load injectors that work at the network level, replaying through a Web browser accurately performs all activities such as typing data in Web forms, scrolling pages and clicking buttons. The typing speed in forms can also be configured to model a real user typing. This is particularly useful when inputs are processed by JavaScript code that can be triggered on each keystroke. Through the browser, BenchLab captures the real user perceived latency including network transfer, page processing and rendering time.

## 4. Implementation

BenchLab is implemented using open source software and is also released as open source software for use by the community. The latest version of the software and documentation can be found on our Web site [3].

## 4.1. Trace recorder

We have implemented a trace recorder for Apache httpd that collects request information from a live system using the standard httpd logging mechanisms (mod_log_config and mod_log_post). We then process these logs to generate traces in HAR format. We have contributed a new Java library called HarLib to manage HAR traces in files and databases.

Additionally we can record HTML pages generated using mod_dumpio. This is useful to build tools that will check the consistency of Web pages obtained during replay against the originally captured HTML.

### 4.2. Browser based load injection

We use the Selenium/Webdriver [14] framework that provides support for Firefox, Internet Explorer and Chrome on almost all the platforms (Linux, Windows, MacOS) where they are available. Safari support is experimental as well as Webkit based browsers for Android and iPhone. The BenchLab client runtime (BCR) is a simple Java program interfacing with Selenium. We currently use Selenium 2.0b3 that includes Webdriver. The BCR can start any Firefox, IE or Chrome browser installed on the machine and connect it to a BenchLab WebApp. On Linux machines that do not have an X server environment readily available, we use X virtual frame buffer (Xvfb) to render the browser in a virtual X server. This is especially useful when running clients in the cloud on machines without a display.
When a browser is assigned to an experiment, the BCR downloads the trace it has to replay through the browser and stores it in a local file. The information about the experiment, trace and session being executed by the browser is encoded by the BenchLab WebApp in cookies stored in the Web browser.
The BCR parses the trace file for the URLs and encoded parameters that are then set in the corresponding forms (text fields, button clicks, file uploads, etc.). When a URL is a simple "GET" request, the BCR waits according to the timestamp before redirecting the browser to the URL. When a form has to be filled before being submitted, the BCR starts filling the form as soon as the page is ready and just waits before clicking the submit button. As we emulate the user typing speed it can take multiple minutes to fill some forms like edits to a wiki page with Wikipedia.
The BCR relies on the browser's performance profiling tools to record detailed timings in HAR format. This includes network level performance (DNS resolution, send/wait/receive time…) and browser level rendering time. The entire HTML page and media files can be recorded for debugging purposes if the client machine has enough storage space. An alternative compressed CSV format is also available to record coarser grain performance metrics on resource constrained devices.
We have built Xen Linux virtual machines with the BCR and Firefox to use on private clouds. We also built Amazon EC2 AMIs for both Windows and Linux with Firefox, Chrome and Internet Explorer (Windows only for IE). These AMIs are publicly available.

### 4.3. BenchLab WebApp

The BenchLab WebApp is a Java application implemented with JSP and Servlets. It uses an embedded Apache Derby database. Each trace and experiment is stored in separate tables for better scalability. Virtual machines of Web applications are not stored in the database but we store a URL to the image file that can point to the local file system or a public URL such as an S3 URL if the images are stored in the Amazon Simple Storage Service.
The user interface is intentionally minimalist for efficiency and scalability allowing a large number of browsers to connect. BenchLab makes a minimal use of JavaScript and does not use AJAX to keep all communications with clients purely asynchronous. Similarly no clock synchronization is needed nor required.
As the BenchLab WebApp is entirely self-contained, it can easily be deployed on any Java Web application server. We currently use Apache Tomcat 6 for all our experiments. We have tested it successfully on Linux and Windows platforms, but it should run on any platform with a Java runtime.
The BenchLab WebApp acts as a repository of traces, benchmark virtual machines and experiments with their results. That data can be easily downloaded using any Web browser or replicated to any other BenchLab WebApp instance.

### 4.4. Application backends

We provide Xen virtual machines and Amazon AMIs of the CloudStone benchmark and the Wikibooks application on our Web site [3]. As BenchLab does not impose any deployment or configuration framework, any application packaged in a VM can be used as a benchmark backend.

#### *4.4.1. CloudStone*

CloudStone [15] is a multi-platform, multi-language benchmark for Web 2.0 and Cloud Computing. It is composed of a load injection framework called Faban, and a social online calendar Web application called Olio [12]. A workload driver is provided for Faban to emulate users using a Markov model.
We have chosen the PHP version of Olio and packaged it in a virtual machine that we will refer to as *OlioVM*. OlioVM contains all the software dependencies to run Olio including a MySQL database and the Java Webapp implementing a geocoding service.
Faban is packaged in another VM with the load injection driver for Olio. We refer to this VM as *FabanVM*. Faban relies on the Apache HttpClient v3 (HC3) library [1] for the HTTP transport layer to interact with the Web application. We have instrumented Faban to record the requests sent to HC3 in order to obtain trace files with all needed parameters for interactions that require user input in POST methods. FabanVM is not used for load injection in our experiments but only to generate traces that can be replayed using our replay tool. The replay tool is a simple Java program replaying HTTP requests using the HC3 library.
As part of this work, we fixed a number of issues such as the workload generator producing invalid inputs for

the Olio calendaring applications (e.g., invalid phone numbers, zip codes, state name). We process trace files to fix erroneous inputs and use these valid input traces in all experiments except in section 5.3.3 where we evaluate the impact of invalid inputs.

#### *4.4.2. Wikibooks*

Wikibooks [20] is a wiki of the Wikimedia foundation and provides free content textbooks and annotated texts. It uses the same Wikimedia wiki software as Wikipedia which is a PHP application storing its data in a MySQL database. Our Wikibooks application backend includes all Wikimedia extensions necessary to run the full Web site including search engine and multimedia content.

The Wikibooks virtual appliance is composed of two virtual machines. One virtual machine contains the Wikimedia software and all its extensions and the other VM runs the database. Database dumps of the Wikibooks content are freely available from the Wikimedia foundation in compressed XML format. We currently use a dump from March 2010 that we restored into a MySQL database. Real Wikibooks traces are available from the Wikibench Web site [19].

Due to copyright issues, the multimedia content in Wikibooks cannot be redistributed, and therefore, we use a multimedia content generator that produces images with the same specifications as the original content but with random pixels. Such multimedia content can be either statically pre-generated or produced on-demand at runtime.

### 4.5. Limitations

Our current implementation is limited by the current functionality of the Selenium/Webdriver tools we are using. Support for Firefox on all platforms and Internet Explorer on Windows are overall stable though performance may vary on different OS versions. The Chrome driver does not support file upload yet but it provides experimental access to Webkit browsers such as Safari and Android based browsers.

Our prototype does not support input in popup windows but we are able to discard JavaScript alert popups when erroneous input is injected into forms.

The current BenchLab WebApp prototype does not implement security features such as browser authentication, data encryption or result certification.

## 5. Experimental Results

### 5.1. Experimental setup and methodology

For all our experiments, the Web applications run on an 8-core AMD Opteron 2350 server, 4GB RAM with a Linux 2.6.18-128.1.10.el5xen 64 bit kernel from a standard CentOS distribution. We use the Xen v3.3.0 hypervisor. The server is physically located in the data center of the UMass Amherst campus.

CloudStone is configured with 1 virtual CPU (vCPU) and 512MB of memory for OlioVM. The Olio database is initialized for 500 users. FabanVM is allocated 1 vCPU and 1024MB of memory and runs on a different physical machine. Wikibooks VMs are both allocated 4 vCPUs and 2GB of RAM.

Experiments using Amazon EC2 resources use Linux small instances with a CentOS distribution and the BenchLab client runtime controlling Firefox 3.6.13. The BenchLab Web application runs in Tomcat 6 on a laptop located on the UMass Amherst campus.

We have written a Java replay tool similar to httperf that can replay Faban traces through the Apache HttpClient 3 library. We have validated the tool by replaying traces generated by Faban and comparing the response time and server load with the ones obtained originally by Faban.

### 5.2. Realistic application data sets

In this experiment we illustrate the importance of having benchmark applications with realistic amounts of application state. The CloudStone benchmark populates the database and the filestore containing multimedia content according to the number of users to emulate. The state size grows proportionally to the number of users. Table 1 shows the dataset state size from 3.2GB for 25 users to 44GB for 500 users.

**Table 1. CloudStone Web application server load observed for various dataset sizes using a workload trace of 25 users replayed with Apache HttpClient 3.**

| Dataset size | State size (in GB) | Database rows | Avg CPU load with 25 users |
|---|---|---|---|
| 25 users | 3.2 | 173745 | 8% |
| 100 users | 12 | 655344 | 10% |
| 200 users | 22 | 1151590 | 16% |
| 400 users | 38 | 1703262 | 41% |
| 500 users | 44 | 1891242 | 45% |

We generated a load for 25 users using the Faban load generator and recorded all the interactions with their timestamps. We then replayed the trace using 25 emulated browsers and observed the resource usage on the CloudStone Web application (Olio) when different size data sets were used in the backend. The results in Table 1 show the CPU load observed in the Web Application VM. Note that in this experiment the trace is replayed through the Apache HttpClient 3 library and not using a real Web browser. The average CPU load on the server is 8% with the 25 user dataset but it reaches 45% for the exact same workload with a 500 user dataset. This is mainly due to less effective caching and less efficient database operations with larger tables.

Real applications like Wikipedia wikis have databases of various sizes with the largest being the English Wikipedia database which is now over 5.5TB. This experiment shows that even for a modest workload accessing the exact same working set of data, the impact on the server load can vary greatly with the dataset size. It is therefore important for realistic benchmarks to provide realistic datasets.

## 5.3. Real browsers vs emulators

### *5.3.1. Complexity of Web interactions*

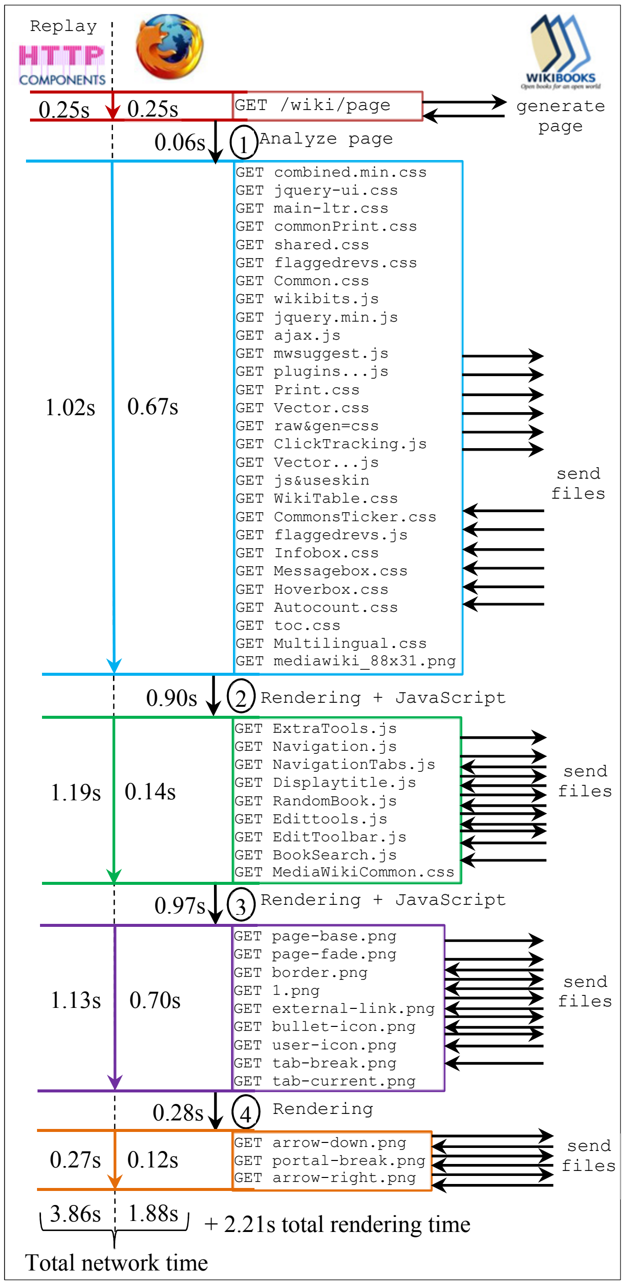
Real Web applications have complex interactions with the Web browser as shown in Table 2. While accessing the home page of older benchmarks such as RUBiS or TPC-W only generates 2 to 6 requests to fetch the page content. Their real life counterpart, eBay.com and amazon.com require 28 and 141 browser-server interactions, respectively. A more modern benchmark application such as CloudStone's Olio requires 28 requests which is still far from the 176 requests of the most popular social network Web site Facebook. When the user enters http://en.wikibooks.org/ in his favorite Web browser, 62 requests are generated on his behalf by the Web browser to fetch the content of the Wikibooks home page. Even if modern HTTP client libraries such as Apache HttpComponents Client [1] provide a good implementation of HTTP transport very similar to the one used in Web browsers, other functionalities such as caching, JavaScript execution, content type detection, request reformatting or redirection may not be accurately emulated.

**Table 2. Browser generated requests per type when browsing the home page of benchmarks and Web sites.**

| Benchmark | HTML | CSS | JS | Multimedia | Total |
|---|---|---|---|---|---|
| RUBiS | 1 | 0 | 0 | 1 | 2 |
| eBay.com | 1 | 3 | 3 | 31 | 28 |
| TPC-W | 1 | 0 | 0 | 5 | 6 |
| amazon.com | 6 | 13 | 33 | 91 | 141 |
| CloudStone | 1 | 2 | 4 | 21 | 28 |
| facebook.com | 6 | 13 | 22 | 135 | 176 |
| wikibooks.org | 1 | 19 | 23 | 35 | 78 |
| wikipedia.org | 1 | 5 | 10 | 20 | 36 |

To further understand how real browsers interact with real applications, we investigate how Firefox fetches a page of the Wikipedia Web site and compare it to an HTTP replay. The workflow of operations and the corresponding timings are shown in Figure 2. The times for each block of GET operations correspond to the network time measured by our HTTP replay tool (on the left) and Firefox (on the right). Times between blocks correspond to processing time in Firefox.

First we observe that the complexity of the application forces the browser to proceed in multiple phases. After sending the requested URL to the Web application, the browser receives an HTML page that it analyzes (step 1 on Figure 2) to find links to JavaScript code and additional content to render the page (.css, images…). Firefox opens six connections and performs the content download in parallel. It then starts to render the page and execute the JavaScript onLoad operations (step 2). This requires additional JavaScript files to be downloaded and another round of code execution (step 3).



```
GET /wiki/page
```

```
GET combined.min.css
GET jquery-ui.css
GET main-ltr.css
GET commonPrint.css
GET shared.css
GET flaggedrevs.css
GET Common.css
GET wikibits.js
GET jquery.min.js
GET ajax.js
GET mwsuggest.js
GET plugins...js
GET Print.css
GET Vector.css
GET raw&gen=css
GET ClickTracking.js
GET Vector...js
GET js&useskin
GET WikiTable.css
GET CommonsTicker.css
GET flaggedrevs.js
GET Infobox.css
GET Messagebox.css
GET Hoverbox.css
GET Autocount.css
GET toc.css
GET Multilingual.css
GET mediawiki_88x31.png
```

```
GET ExtraTools.js
GET Navigation.js
GET NavigationTabs.js
GET Displaytitle.js
GET RandomBook.js
GET Edittools.js
GET EditToolbar.js
GET BookSearch.js
GET MediaWikiCommon.css
```

```
GET page-base.png
GET page-fade.png
GET border.png
GET 1.png
GET external-link.png
GET bullet-icon.png
GET user-icon.png
GET tab-break.png
GET tab-current.png
```

```
GET arrow-down.png
GET portal-break.png
GET arrow-right.png
```

**Figure 2. Time breakdown of a Wikibooks page access with Firefox 3.6.13 and HTTP replay.**

Finally images are downloaded reusing the same six connections and a final rendering round (step 4) triggers the download of the 3 final images at the bottom of the page. The total page loading time in Firefox is 4.09s with 1.88s for networking and 2.21s for processing and rendering. The single threaded HTTP replay tool is not able to match Firefox's optimized communications and does not emulate processing, thus generating different access patterns on the server leading to a different resource usage.

Finally, Table 3 shows how typing text in a Web page can result in additional requests to a Web application. When the user enters text in the search field of Wikibooks, a request is sent to the server on each keystroke to provide the user with a list of suggestions. The typing speed and the network latency influence how many requests are going to be made to the server.

**Table 3. JavaScript generated requests when typing the word 'Web 2.0' in Wikibooks' search field.**

```
GET /api.php?action=opensearch&search=W
GET /api.php?action=opensearch&search=Web
GET /api.php?action=opensearch&search=Web+
GET /api.php?action=opensearch&search=Web+2
GET /api.php?action=opensearch&search=Web+2.
GET /api.php?action=opensearch&search=Web+2.0
```

In Table 3's example, the user starts by typing 'W' causing a request to the server. She then quickly types the letter 'e' before the response has come back. When she types the next letter 'b' a second request goes to the server. Each following keystroke is followed by another request. This shows that even replaying in a Web browser needs to take in to consideration the speed at which a user performs operations since this can have an impact on how many requests are issued to the server directly affecting its load.

### *5.3.2. Latencies and server load*

In this experiment, we inject the same 25 user CloudStone workload from the Amazon EC2 East coast data center to our server running at UMass Amherst. The emulator runs the 25 users from one virtual machine whereas 25 server instances each running one Firefox Web browser inject the load for the realistic injection. Figure 3 shows the latencies observed by the emulator and by the Web browsers.

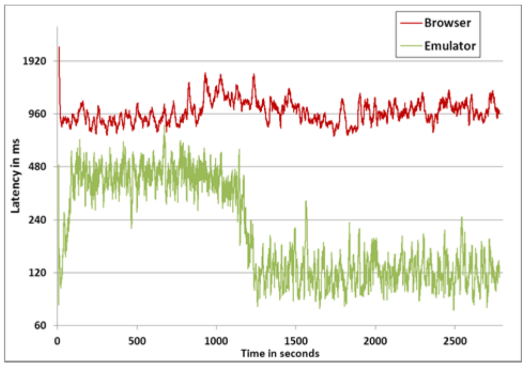


**Figure 3. Browser vs Emulator measured latencies for the same load of 25 users using CloudStone between EC2 East coast and UMass Amherst.**

The emulator has to mimic all the requests that a real Web browser would issue. Therefore a lot of small queries to fetch style sheets (.css) or JavaScript (.js) have small latencies. Some more complex pages are not fetched as efficiently on multiple connections and result in much higher latencies when the latencies of sub-requests are added up.

The latencies observed by the Web browsers vary significantly from the ones observed by the emulator because not only do they account for the data transfer time on the network, but also because they include the page rendering time that is part of the user perceived latency. Another key observation is that the showEvent interaction that displays information about an event in Olio's online calendar makes use of the Yahoo map API. As the emulator does not execute any JavaScript, all interactions with the real Yahoo Web site and its map API are not accounted in the interaction time. When a real browser is used, it contacts the Yahoo map Web site and displays the map with the location of the event. The page rendering time is then influenced not only by the response time of the Olio Web application but also with the interactions with Yahoo's Web site.

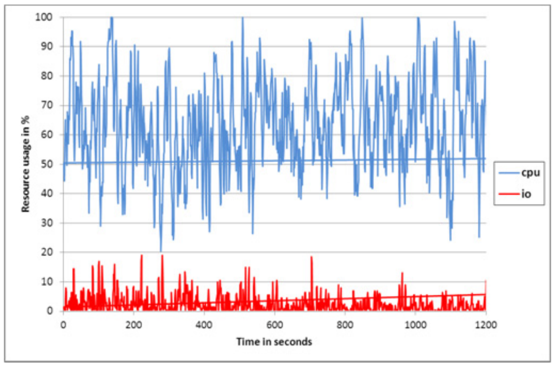


**Figure 4. Server side CPU and disk IO usage with an emulator injecting a 25 user load from EC2 East coast to CloudStone at UMass Amherst.**

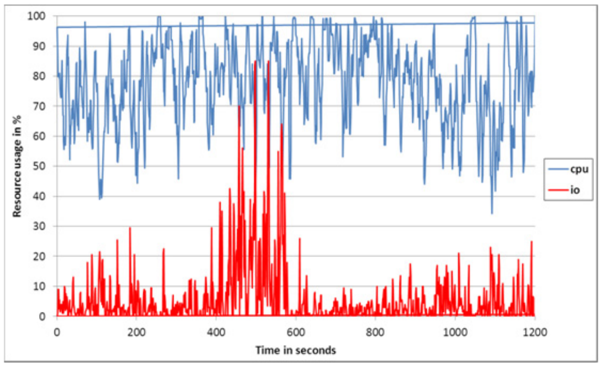


**Figure 5. Server side CPU and disk io usage with 25 Firefox browsers from EC2 East coast to Cloud-Stone at UMass Amherst.**

Figure 4 and Figure 5 show the average CPU usage measured by vmstat every second on the Web application server for the emulated and realistic browser injection, respectively. While the user CPU time oscillates a lot in the emulated case it averages 63.2%. The user CPU time is more steady and constantly higher with the browser injected load at an average of 77.7%. Moreover, we notice peaks of disk IO during the experiment (for example at time 500 seconds), indicating that the IO subsystem of the server is more stressed when serving the browser generated requests.

### *5.3.3. Impact of JavaScript on Browser Replay*

When a user fills a form, JavaScript code can check the content of the form and validate all the fields before being submitted to the application server. While the server has to spend more resources to send all the JavaScript code to the client, it does not have to treat any malformed requests with improper content that can be caught by the JavaScript validation code running in the Web browser.

In this experiment, we use the addPerson interaction of CloudStone that registers a new user with her profile information. The JavaScript code in that page generates a query to the server when the user name is typed in to check if that name is available or already taken. Other fields such as telephone number are checked for format and other fields are checked for missing information. Entering a zip code generates a query to the Geocoder service that returns the corresponding city and state names that are automatically filled in the corresponding fields of the form.

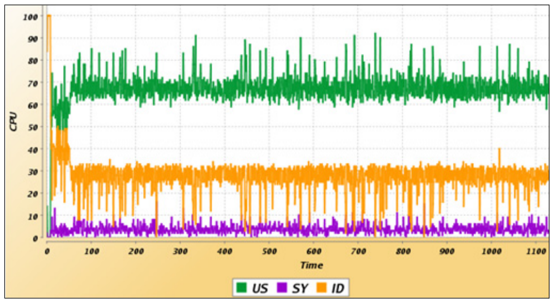


**Figure 6. Server side CPU load (user, system, idle) with emulated load injection of 25 virtual users executing CloudStone's addPerson interaction with valid or bad inputs.**

The original Olio workload driver generated malformed data that does not pass JavaScript checks but are accepted by the Olio application that does not check data sent from the client. We found this bug in the Olio application that inserts improper data in the database. We use two traces: one with the original malformed data and another one with valid inputs where we fixed the problems found in the original trace.

We emulate 25 clients from a server located in EC2 East coast's data center and run both traces. Figure 6 shows the load observed on the OlioVM when using emulated users. The CPU utilization is steady at around 70% for both traces. As the application does not check data validity and the emulator does not execute JavaScript, there is no change between good and bad inputs.

Figure 7 shows the CPU load observed on the server when the valid input trace is injected through Firefox. As forms are filled with data, additional queries are issued by JavaScript as mentioned earlier. This causes heavy weight write queries to be interlaced with lighter read queries. These additional context switches between the PHP scripts running the application and the Tomcat container running the Geocoder service cause significantly different resource usage in the Web application virtual machine.

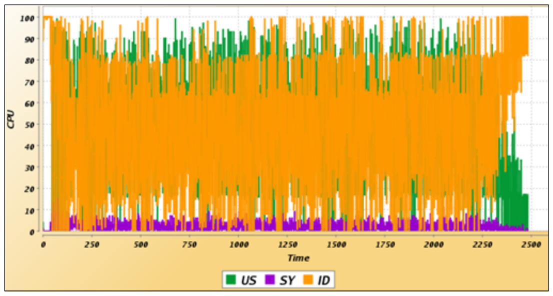


**Figure 7. Server side CPU load (user, system, idle) using Firefox (25 browsers running from EC2 East coast) executing CloudStone's addPerson interaction with valid inputs.**

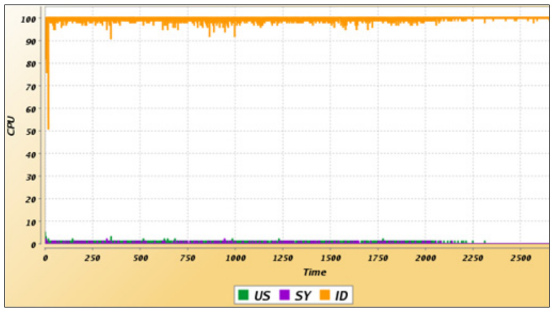


**Figure 8. Server side CPU load (user, system, idle) using Firefox (25 browsers running from EC2 East coast) executing CloudStone's addPerson interaction with erroneous inputs.**

Figure 8 shows the load seen by the Web application server when the trace with invalid inputs is injected through Firefox. As the JavaScript code checks for erroneous inputs, the requests never reach the application server. The only activity observed by the application is to answer the login uniqueness checks.

## 5.4. LAN vs WAN

In these experiments, we evaluate the impact of WAN based load injection vs traditional LAN based injection.

#### *5.4.1. Local vs remote users*

We observed the response time for a trace of 25 emulated users injected with our replay tool from a machine on the same LAN as the server and from a machine on Amazon EC2 East coast data center. As expected, the latencies are much higher on the WAN with 149ms average vs 44ms on the LAN. However, we observe that the latency standard deviation more than doubles for the WAN compared to the LAN.

The CPU usage for the WAN injection has already been presented in Figure 4 with an average CPU load of 54.8%. The CPU usage for the LAN experiment shows a highly varying CPU load but at a much lower 38.3% average. We attribute most of these differences to the increased connection times that require more processing to perform flow control on the connections, more context switches between longer lived sessions and more memory pressure to maintain more session states and descriptors simultaneously open. The exact root causes of these variations in server resource usage between LAN and WAN needs to be investigated further but we think that BenchLab is an ideal testbed for the research community to conduct such experiments.

#### *5.4.2. Geographically dispersed load injection*

We investigate the use of multiple data centers in the cloud to perform a geographically dispersed load injection. We re-use the same 25 user Cloudstone workload and distribute Web browsers in different Amazon data centers as follows: 7 US East coast (N. Virginia), 6 US West coast (California), 6 Europe (Ireland) and 6 Asia (Singapore). Such a setup (25 distributed instances) can be deployed for as little as $0.59/hour using Linux micro instances or $0.84/hour using Windows instances. More powerful small instances cost $2.30/hour for Linux and $3.00/hour for Windows. Figure 9 shows the latency reported by all Web browsers color coded per region (y axis is log scale).

**Table 4. Average latency and standard deviation observed in different geographic regions**

| | US East | US West | Europe | Asia |
|---|---|---|---|---|
| Avg latency | 920ms | 1573ms | 1720ms | 3425ms |
| Std deviation | 526 | 776 | 906 | 1670 |

As our Web application server is located in the US East coast, the lowest latencies are consistently measured by browsers physically located in the East coast data center. Average latency almost doubles for requests originating from the West coast or Europe. Finally, as expected, the Asian data center experiences the longest latencies. It is interesting to notice that the latency standard deviation also increases with the distance from the Web application server as summarized in Table 4. The server average CPU usage is 74.3% which is slightly less than when all browsers were located in the East coast (77.7%).

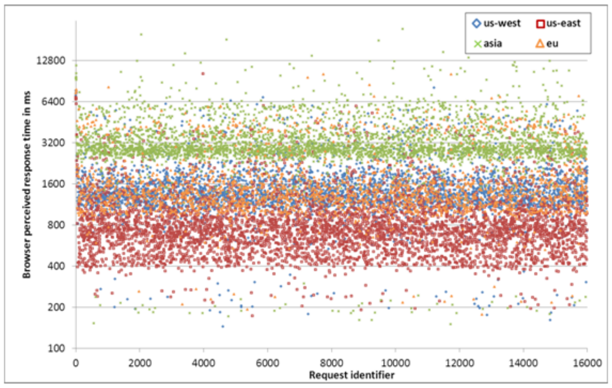


**Figure 9. Browser (Firefox) perceived latency in ms (y axis is log scale) on a Cloudstone workload with users distributed as follows: 7 US East coast, 6 US West coast, 6 Europe and 6 Asia. Server location is UMass Amherst (US East coast, Massachusetts).**

### 5.5. Summary

We have shown that real modern Web applications have complex interactions with Web browsers and that the state size of the application can greatly affect the application performance. Traditional trace replay tools cannot reproduce the rich interactions of browsers or match their optimized communication with Web application servers. Therefore the load observed on application servers varies greatly when the same workload is injected through an emulated browser or a real Web browser. This is further accentuated when JavaScript code generates additional queries or performs error checking that prevents erroneous inputs to reach the server.

Finally, we have shown the influence of LAN vs WAN load injection using real Web browsers deployed in a public cloud. Not only the latency and its standard deviation increase with the distance but the load on the server significantly differs between a LAN and a WAN experiment using the exact same workload.

## 6. Related Work

Benchmarks such as RUBiS [13] and TPC-W [16] have now become obsolete. BenchLab uses CloudStone [15] and Wikibooks [20] as realistic Web 2.0 application backends. The Rain [2] workload generation toolkit separates the workload generation from execution to be able to leverage existing tools such as httperf. BenchLab uses the same concept to be able to replay real workload traces from real applications such as Wikibooks or Wikipedia [17] in Web browsers.

Browser automation frameworks have been developed primarily for functional testing. BenchLab uses real Web browsers for Web application benchmarking. Commercial technologies like HP TruClient [6] or Keynote Web performance testing tools [7] offer load injection from modified versions of Firefox or Internet Explorer. BrowserMob [4] provides a similar service using Firefox and Selenium. However, these proprietary products can only be used in private clouds or dedicated test environments. BenchLab is fully open and can be deployed on any device that has a Web browser. By deploying browsers on home desktops or cell phones, BenchLab can be used to analyze last mile latencies.

Server-side monitors and log analysis tools have been used previously to try to model the dependencies between file accesses and predict the full page load times observed by clients [10][18]. BenchLab's use of real Web browsers allows it to accurately capture the behavior of loading Web pages composed of multiple files, and could be used by Web service providers to monitor the performance of their applications. When deployed on a large number of machines (home desktops, cell phones, cloud data centers…), BenchLab can be used to reproduce large scale flash crowds. When client browsers are geographically dispersed, BenchLab can be used to evaluate Content Delivery Network (CDN) performance or failover mechanisms for highly available Web applications.

BenchLab is designed to be easily deployed across wide geographic areas by utilizing public clouds. The impact of wide area latency on Web application performance has been studied in a number of scenarios [5]; BenchLab provides a standardized architecture to allow application developers and researchers to measure how their systems perform in real WAN environments. We believe that BenchLab can be used to measure the effectiveness of WAN accelerators (CDNs or proxy caches) as well as validate distributions modeling WAN load patterns.

## 7. Conclusion

We have demonstrated the need to capture the behavior of real Web browsers to benchmark real Web 2.0 applications. We have presented BenchLab, an open testbed for realistic benchmarking of modern Web applications using real Web browsers. BenchLab employs a modular architecture that is designed to support different backend server applications. Our evaluation has illustrated the need for 1) updated Web applications with realistically sized datasets to use as benchmarks, 2) real browser based load injection tools that authentically reproduce user interactions and 3) wide area benchmark client deployments to match the network behavior seen by real applications. We believe that BenchLab meets these needs, and we hope that it will help the research community improve the realism and accuracy of Web application benchmarking. We are making all the BenchLab software (runtime, tools, Web applications…) available to the community under an open source license on our Web site [3].

## 8. Acknowledgement

The authors would like to thank the anonymous reviewers and our shepherd Geoffrey Voelker for their valuable feeback. We would also like to thank Fabien Mottet and Vivien Quema from INRIA Rhône-Alpes, Guillaume Pierre from Vrije University and Vimal Mathew from UMass, for their contributions to BenchLab. This research was supported in part by grants from Amazon AWS, UMass President's Science & Technology fund, and NSF grants CNS-0916972, CNS-083243, CNS-0720616 and CNS-0855128.

## 9. References

[1] Apache HttpComponents – http://hc.apache.org/

[2] A. Beitch, B. Liu, T. Yung, R. Griffith, A. Fox and D. Patterson – *Rain: A Workload Generation Toolkit for Cloud Computing Applications* – Technical Report UCB/EECS-2010-14, February 10, 2010.

[3] BenchLab - http://lass.cs.umass.edu/projects/benchlab/

[4] BrowserMob - http://browsermob.com/performance-testing

[5] S. Chen, K.R. Joshi, M.A. Hiltunen, W.H. Sanders and R.D. Schlichting – *Link Gradients: Predicting the Impact of Network Latency on Multitier Applications* – INFOCOM 2009, pp.2258-2266, 19-25 April 2009.

[6] HP - TruClient technology: Accelerating the path to testing modern applications – Business white paper, 4AA3-0172ENW, November 2010.

[7] R. Hughes and K. Vodicka – Why Real Browsers Matter – Keynote white paper, http://www.keynote.com/docs/whitepapers/why_real_browers_matter.pdf.

[8] D. Krishnamurthy, J. A. Rolia and Shikharesh Majumdar – *A Synthetic Workload Generation Technique for Stress Testing Session-Based Systems* – IEEE Transaction on Software Engineering. 32, 11 - November 2006.

[9] HTTP Archive specification (HAR) v1.2 - http://www.softwareishard.com/blog/har-12-spec/.

[10] Z. Li, M. Zhang, Z. Zhu, Y. Chen, A.G. Greenberg, and Y. Wang - *WebProphet: Automating Performance Prediction for Web Services* – NSDI, 2010, pp.143-158.

[11] E. M. Nahum, M.C. Rosu, S. Seshan and J. Almeida – *The effects of wide-area conditions on WWW server performance* – SIGMETRICS 2001.

[12] Olio – http://incubator.apache.org/olio/

[13] RUBiS Web site – http://rubis.ow2.org.

[14] Selenium - http://seleniumhq.org/

[15] W. Sobel, S. Subramanyam, A. Sucharitakul, J. Nguyen, H. Wong, A. Klepchukov, S. Patil, A. Fox and D. Patterson – *Cloudstone: Multi-platform, multi-language benchmark and measurement tools for Web 2.0* – Cloud Computing and its Applications CCA-08, 2008.

[16] TPC-W Benchmark, ObjectWeb implementation, http://jmob.objectWeb.org/tpcw.html

[17] G. Urdaneta, G. Pierre and M. van Steen – *Wikipedia Workload Analysis for Decentralized Hosting* – Elsevier Computer Networks, vol.53, July 2009.

[18] J. Wei, C.Z. Xu - *Measuring Client-Perceived Pageview Response Time of Internet Services* – IEEE Transactions on Parallel and Distributed Systems, 2010.

[19] WikiBench - http://www.wikibench.eu/

[20] Wikibooks – http://www.wikibooks.org

[21] Wikipedia – http://www.wikipedia.org