

# Client-side Load Balancer using Cloud

Sewook Wee  
Accenture Technology Labs  
50 W. San Fernando Street, Suite 1200  
San Jose, California, USA  
sewook.wee@accenture.com

Huan Liu  
Accenture Technology Labs  
50 W. San Fernando Street, Suite 1200  
San Jose, California, USA  
huan.liu@accenture.com

## ABSTRACT

Web applications' traffic demand fluctuates widely and unpredictably. The common practice of provisioning a fixed capacity would either result in unsatisfied customers (underprovision) or waste valuable capital investment (overprovision). By leveraging an infrastructure cloud's on-demand, pay-per-use capabilities, we finally can match the capacity with the demand in real time. This paper investigates how we can build a large-scale web server farm in the cloud. Our performance study shows that using existing cloud components and optimization techniques, we cannot achieve high scalability. Instead, we propose a client-side load balancing architecture, which can scale and handle failure on a milli-second time scale. We experimentally show that our architecture achieves high throughput in a cloud environment while meeting QoS requirements.

## 1. INTRODUCTION

An infrastructure cloud, such as Amazon's EC2/S3 services [2], promises to fundamentally change the economics of computing. First, it provides a practically unlimited infrastructure capacity (e.g., computing servers, storage or network) on demand. Instead of grossly over-provisioning upfront due to uncertain demands, users can elastically provision their infrastructure resources from the provider's pool only when needed. Second, the pay-per-use model allows users to pay for the actual consumption instead of for the peak capacity. Third, a cloud infrastructure is much larger than most enterprise data centers. The economy of scale, both in terms of hardware procurement and infrastructure management and maintenance, helps to drive down the infrastructure cost further.

These characteristics make cloud an attractive infrastructure solution especially for web applications due to their variable loads. Because web applications could have a dramatic difference between their peak load (such as during flash crowd) and their normal load, a traditional infrastructure is ill-suited for them. We either grossly over-provision

for the potential peak, thus wasting valuable capital, or provision for the normal load, but not able to handle peak when it does materialize. Using the elastic provisioning capability of a cloud, a web application can ideally provision its infrastructure tracking the load in real time and pay only for the capacity needed to serve the real application demand.

Due to the large infrastructure capacity a cloud provides, there is a common myth that an application can scale up unlimitedly and automatically when application demand increases. In reality, our study shows that scaling an application in a cloud is more difficult, because a cloud is very different from a traditional enterprise infrastructure in at least several respects.

First, in enterprises, application owners can choose an optimal infrastructure for their applications amongst various options from various vendors. In comparison, a cloud infrastructure is owned and maintained by the cloud providers. Because of their commodity business model, they only offer a limited set of infrastructure components. For example, Amazon EC2 only offers 5 types of virtual servers and application owners cannot customize the specification of them.

Second, again due to its commodity business model, a cloud typically only provides commodity Virtual Machines (VM). The computation power and the network bandwidth is typically less than high-end servers. For example, all Amazon VMs are capable of at most transmitting at roughly 800Mbps, whereas, commercial web servers routinely have several Network Interface Cards (NIC), each capable of at least 1Gbps. The commodity VMs require us to use horizontal scaling to increase the system capacity.

Third, unlike in an enterprise, application owners have little or no control of the underlying cloud infrastructure. For example, for security reasons, Amazon EC2 disabled many networking layer features, such as ARP, promiscuous mode, IP spoofing, and IP multicast. Application owners have no ability to change these infrastructure features. Many performance optimization techniques rely on the infrastructure choice and control. For example, to scale a web application, application owners either ask for a hardware load balancer or ask for the ability to assign the same IP address to all web servers to achieve load balancing. Unfortunately, neither option is available in Amazon EC2.

Last, commodity machines are likely to fail more frequently. Any architecture design based on cloud must handle machine failures quickly, ideally in a few milli-seconds or faster, in order not to frequently disrupt service.

Because of these characteristics, cloud-hosted web applications tend to run on a cluster with many standard com-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SAC'10 March 22-26, 2010, Sierre, Switzerland.

Copyright 2010 ACM 978-1-60558-638-0/10/03 ...\$10.00.

modity web servers, thus requiring a scalable and agile load balancing solution. In this study, we propose a client-side load balancing architecture that not only leverages the strength of existing cloud components, but also overcomes the limitations posed above. More specifically, we present the following contributions.

1. **Propose client-side load balancing architecture:** Differing from previous proposals on client-side load balancing, our proposal is built on insights gained from our performance studies of cloud components. We leverage the strength of a cloud component (S3's scalability) to avoid any single point of scalability bottleneck.
2. **A practical implementation:** Previous implementations are not transparent to end users. We use JavaScript technology to handle all load-balancing details behind the scene. From a user's perspective, he is not able to distinguish a web site using client-side load balancing from a normal web site. We use JavaScript to get around current browsers' cross-domain security limitations.
3. **Realistic evaluation:** We evaluate the proposed architecture using a realistic benchmark suite—SPECweb-2005 [14]. Our evaluation shows that our proposed architecture can indeed scale linearly as demand increases.

In the rest of the paper, we show the limitations of cloud to host a web presence using the standard techniques. We then describe our client-side load balancing architecture and implementation. Last, we present the evaluation results.

## 2. PRIOR WORK

There are well established techniques to scale a web server farm in an owned infrastructure. We will briefly visit them here and point out their limitations if deployed in cloud.

### 2.1 Load Balancer

A standard way to scale web applications is by using a hardware-based load balancer [5]. The load balancer assumes the IP address of the web application, so all communication with the web application hits the load balancer first. The load balancer is connected to one or more identical web servers in the back-end. Depending on the user session and the load on each web server, the load balancer forwards packets to different web servers for processing. The hardware-based load balancer is designed to handle high-level of load, so it can easily scale.

However, a hardware-based load balancer uses application specific hardware-based components, thus it is typically expensive. Because of cloud's commodity business model, a hardware-based load balancer is rarely offered by cloud providers as a service. Instead, one has to use a software-based load balancer running on a generic server.

A software-based load balancer [8, 12, 1] is not a scalable solution, though. The scalability is usually limited by the CPU and network bandwidth capacity of the generic server that the load balancer runs on, and a generic server's capacity is much smaller than that of a hardware-based load

balancer. For example, in our test [11], we found that an Amazon EC2 instance can handle at most 400 Mbps combined ingress and egress traffic.

Even though some cloud platforms, such as Google App Engine [7], implicitly offer a hardware-based load balancer, we cannot easily get around their limitations because of the limited control we have. In our test, Google App Engine is only able to handle 10 Mbps in/out or less traffic because of its quota mechanism.

HTTP protocol [6] has a built-in HTTP redirect capability, which can instruct the client to send the request to another location instead of returning the requested page. Using HTTP redirect, a front-end server can load balance traffic to a number of back-end servers. However, just like a software load balancer, a single point of failure and scalability bottleneck still exist.

### 2.2 DNS Load Balancing

Another well established technique is DNS aliasing [10]. When a user browses to a domain (e.g., `www.website.com`), the browser first asks its local DNS server for the IP address (e.g., `209.8.231.11`), then, the browser contacts the IP address. In case the local DNS server does not have the IP address information for the asked domain, it contacts other DNS servers that have the information, which will eventually be the original DNS server that the web server farm directly manages. The original DNS server can hand out different IP addresses to different requesting DNS servers so that the load could be distributed out among the servers sitting at each IP address.

DNS load balancing has its drawbacks—load balancing granularity and adaptiveness—that are not specific to the cloud. First, it does a poor job in balancing the load. For performance reasons, a local DNS server caches the IP address information. Thus, all browsers contacting the same DNS server would get the same IP address. Since the DNS server could be responsible for a large number of hosts, the load could not be effectively smoothed out.

Second, the local DNS server caches IP address for a set period of time, e.g., for days. Until the cache expires, the local DNS server guides requests from browsers to the same web server. When traffic fluctuates at a time scale much smaller than days, tweaking DNS server settings has little effect. Traditionally, this drawback have not been as pronounced because the number of back-end web servers and their IP addresses are static anyway. However, it seriously affects the scalability of a cloud-based web server farm. A cloud-based web server farm elastically changes the number of web servers tracking the volume of traffic in minutes granularity. Days of DNS caching dramatically reduces this elasticity. More specifically, even though the web server farm increases the number of web servers to serve the peak load, IP addresses for new web servers will not be propagated to DNS servers that already have a cached IP address. Therefore, the requests relying on those DNS servers will keep being sent to the old web servers which overloads them while the new web servers remain idle. In addition, when a web server fails, the DNS entry could not be immediately updated. While the DNS changes propagate, users are not able to access the service even though there are other live web servers.

## 2.3 Layer 2 Optimization

In an enterprise where one can fully control the infrastructure, we can apply layer 2 optimization techniques to build a scalable web server farm that does not impose all drawbacks discussed above: expensive hardware, single performance bottleneck, and lack of adaptiveness.

There are several variations of layer 2 optimization. One way, referred to as direct web server return [4], is to have a set of web servers, all have the same IP address, but different layer 2 addresses (MAC address). A browser request may first hit one web server, which may in turn load balance the request to other web servers. However, when replying to a browser request, any web server can directly reply. By removing the constraint that all replies have to go through the same server, we can achieve higher scalability. This technique requires the ability to dynamically change the mapping between an IP address and a layer 2 address at the router level.

Another variation, TCP handoff [9], works in a slightly different way. A browser first establishes a TCP connection with a front-end dispatcher. Before any data transfer occurs, the dispatcher transfers the TCP state to one of the back-end servers, which takes over the communication with the client. This technique again requires the ability for the back-end servers to masquerade the dispatcher's IP address.

Unfortunately, this ability could open doors for security exploits. For example, one can intercept all packets targeting for a host by launching another host with the same IP address. Because of the security concerns, Amazon EC2 disables all layer 2 capabilities so that any layer 2 technique to scale an application will not work in Amazon cloud.

## 2.4 Client Load Balancing

The concept of client side load balancing is not new [3]. One existing approach, the earlier version of NetScape, requires modification to the browser. Given the diversity of web browsers available today, it is difficult to make sure that the visitors to a web site have the required modification. Smart Client [18], developed as part of the WebOS project [15][16], requires Java Applets to perform load balancing at the client. Unfortunately, it has several drawbacks. First, Java Applets require the Java Virtual Machine, which is not available by default on most browsers. This is especially true in the mobile environment. Second, if the user accidentally agrees, a Java Applet could have full access to the client machine, leaving open a big security vulnerability. Third, many organizations only allow administrators to install software, so users cannot view applets by default. Fourth, a Java Applet is an application; the HTML page navigation structure is lost if navigating within the applet. Last, Smart Client still relies on a central server to download the Java Applet and the server list, which still presents a single point of failure and scalability bottleneck.

## 3. NEW ARCHITECTURE

Since traditional techniques for designing a scalable web server farm would not work in a cloud environment, we need to devise new techniques which leverage scalable cloud components while getting around their limitations.

## 3.1 Cloud Components' Performance Characteristics

We have run an extensive set of performance tests on a number of cloud components to understand their performance limits. The results are reported in a technical report [11] and they are not reproduced here due to space limitations (we do not cite it to facilitate double-blind review). A few key observations from the performance study motivated our design. They are listed as follows:

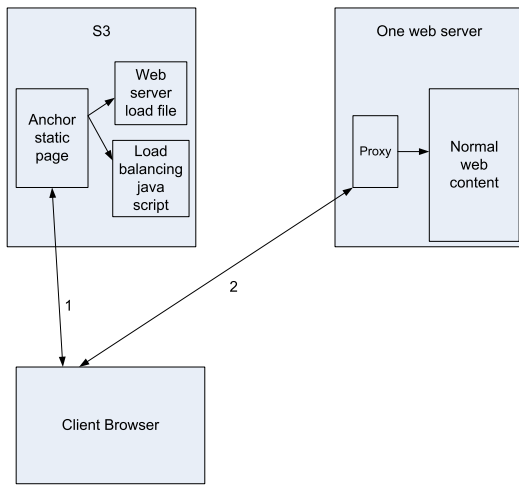
- When used as a web server, a cloud virtual machine has a much smaller capacity than a dedicated web server. For example, in our test, an *m1.small* Amazon instance is only able to handle roughly 1000 SPECweb2005 [14] sessions. In contrast, as reported on SPECweb2005 website, a dedicated web server can handle up to 11,000 sessions.
- When used as a software load balancer, a cloud virtual machine has an even more limited capability. For example, because of traffic relaying, any Amazon instances can only handle 400Mbps client traffic, half of what the network interface can support (800Mbps). Similarly, we found Google App Engine cannot support high traffic when used as a load balancer. Thus, a different mechanism is needed in order to scale higher.
- Amazon S3 is designed for static content distribution. It has built in a high-degree of scalability. In our tests, it can sustain at least 16Gbps throughput. However, S3 can only host static files; it has no ability to run any server side processing such as JSP, CGI, or PHP.

## 3.2 Overview and Originality

We present a new web server farm architecture: client-side load balancing. With this architecture, a browser decides on a web server that it will communicate with amongst available back-end web servers in the farm. The decision is based on the information that lists the web server IP addresses and their individual loads.

The new architecture gets around limitations posed by cloud components, and achieves a high degree of scalability with infrastructure components currently available from cloud providers.

Compared to a software load balancer, our architecture has no single point of scalability bottleneck. Instead, the browser decides on the back-end web server from the list of servers and communication flows directly between the browser and the chosen back-end web server. Compared to DNS aliasing technique, our architecture has a finer load balancing granularity and adaptiveness. Because client's browser makes the decision, we can distribute traffic in session granularity. Moreover, changes in web server list due to auto-scaling or failures can be quickly propagated to browsers (in milli-seconds) so that the clients would not experience extended congestion or outage. This is because the server information is not cached in days but rather it is updated whenever a session is created. We achieve high scalability without requiring sophisticated control on the infrastructure as layer 2 optimization does. Instead, IP addresses of web servers and their individual load information are sufficient.



**Figure 1: Browser, S3 and web server interaction in our client-side load balancing architecture**

### 3.3 Implementation

Our performance study showed that, being a purposely designed platform, S3 has a high degree of scalability when delivering static content. This is not surprising. In order to serve millions of customers, S3 has adopted a distributed implementation that can easily scale. However, S3 is not suitable as a web hosting platform because modern web sites have to deal with both static and dynamic contents. Amazon S3 does not support dynamic content processing, such as CGI, PHP, or JSP.

In our architecture, we use back-end web servers for dynamic content processing, and use client-side load balancing technique to distribute the traffic across the back-end web servers. We host the client-side logic (written in JavaScript) as well as the list of web servers and their load information in S3 in order to avoid a single point of scalability bottleneck.

The detailed architecture is shown in Figure 1. For each dynamic page, we create an anchor static page. This anchor page includes two parts. The first part contains the list of web servers' IP addresses and their individual load (such as CPU, memory, network bandwidth, etc.) information. They are stored in a set of JavaScript variables that can be accessed directly from other JavaScript code. The second part contains the client-side load balancing logic, again written in JavaScript. We need one anchor page for each dynamic page in case a user browses to the URL directly, however, the contents of the anchor pages are all the same.

All anchor pages are hosted in S3 to achieve scalability. We use S3's domain hosting capability, which maps a domain (by DNS aliasing) to S3. We create a bucket with the same name as the domain name (e.g., www.website.com). When a user accesses the domain, the request is routed to S3 and S3 uses the host header to determine the bucket from which to retrieve the content.

When a client browser loads an anchor page, the browser executes the following steps in JavaScript:

1. Examine the load variables to determine to which web server it should send the actual request. The current algorithm randomly chooses a web server where

the probability of choosing any one is inversely proportional to its relative load. The weighted random distribution algorithm is designed to avoid all client browsers flash to the same web server at the same time, as a deterministic algorithm would do.

2. JavaScript sends a request to a proxy on the target web server. The proxy is currently implemented as another PHP web server running on the same machine but at a different port. The JavaScript sends over two pieces of information encoded as URL parameters. First, it sends the browser cookie associated with the site (document.cookie). Second, it sends the URL path (location.pathname).
3. The proxy uses the cookie and URL path to re-construct a new HTTP request; and sends the request to the actual web server.
4. The web server processes the request; invokes dynamic script processor such as CGI, PHP, or JSP, as necessary; and returns the result back to the proxy.
5. The proxy wraps around the result in a JavaScript.
6. The client browser executes the returned JavaScript from the proxy; updates the page display; and updates the cookies if a set-cookie header has been returned.

An example of anchor page is shown below. It simply includes two JavaScript sections along with an empty "ToBeReplaced" tag which will be replaced later with the real page content.

```

<html>
<head><title></title>
<script type="text/javascript">
<!--
// the load balancing logic
</script>
<script type="text/javascript">
// the server list and load
// information in JavaScript
// variables
</script></head>
<body onLoad="load();">
<span id="ToBeReplaced"> </span>
</body></html>
  
```

As described above, the goal of the load balancing logic is to choose a back-end web server based on the load, send the client request (cookie along with URL path) to the proxy, receive the returned JavaScript and update the current HTML page. The JavaScript file returned from the proxy looks like the following.

```

function page() {
    return "<HTML page content>";
}
function set-cookie() {
    <set cookie if instructed
        by web server>
}
  
```

The load balancing logic calls the page() function to replace the current page content (the "ToBeReplaced" tag) with the

string returned by `page()`. It also calls the `set-cookie()` function which contains the logic to set the client side cookie if the web server has indicated so. The `set-cookie()` function may be empty if the server does not set any cookie.

Even though JavaScript has the capability to load additional content by using `XMLHttpRequest`, most browsers would not allow `XMLHttpRequest` to a different server for security reasons. Since we are making a request to a web server other than S3, we have to adopt an architecture like shown above.

To avoid further round trip delays to fetch the anchor page, we keep the load balancing logic JavaScript in the browser memory. When a user clicks any link on the current page, the JavaScript intercepts the request, loads the corresponding link from the proxy, and replaces the current page content with the new page. If the web server fails during a user session (the proxy is not responding), the JavaScript randomly chooses another web server and resends the request. This optimization (saving JavaScript in memory) also ensures that all requests from the same session go to the same back-end server to make it easy to maintain states across requests.

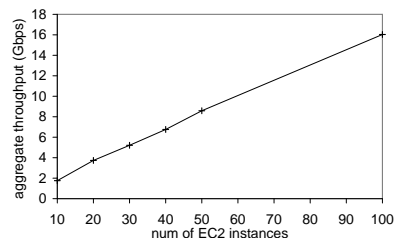
## 4. EVALUATION

In this section, we evaluate the scalability of the new client-side load balancing architecture described in Section 3. We believe that the architecture is scalable because 1) each client independently runs the computation to decide the corresponding back-end server, and 2) it removes the network bandwidth bottleneck of server-side load balancing.

To assess the scalability of the architecture, we use the SPECweb2005 benchmark [14], which is designed to simulate a real web application. It consists of three workloads: *banking*, *support*, and *ecommerce*. As its name suggests, the *banking* workload simulates the web server front-end of an online banking system. It handles all communications through SSL for security reasons and most of the requests and responses are short. If a web server terminates a SSL connection, the session has to be persistence, i.e., all future requests should be sent to the same web server in order to be decrypted. On the other hand, the *support* workload simulates a product support website where users download large files such as documentation files and device drivers. All communications are through regular HTTP and the largest file to download is up to 40MB. The *ecommerce* workload simulates an E-commerce website where users browse the site’s inventory (HTTP) and buy items (SSL). Therefore, in terms of workload characteristics, it is a combination of the above two. In this study, we focus on the *banking* workload because its many short requests and its session persistence requirement stress a load balancer the most among the three workloads.

The SPECweb2005 benchmark consists of two components: the web application and the traffic generator. The web application implements the backend application logic. All pages are dynamically generated and a user can choose from either a PHP implementation or a JSP implementation. The traffic generator generates simulated user sessions to interact with the backend web application, where each session simulates an individual browser. The traffic generator could run on several servers in order to spread out the traffic generating workload.

The performance metric for the benchmark is the num-



**Figure 2: Aggregate S3 throughput as a function of the number of EC2 instances who simultaneously query S3.**

	Average Latency per Request (second)
without CLB	0.407
CLB per request	0.561
CLB per session	0.483

**Table 1: Average latency per request in seconds. SPECweb2005 *Banking* workload with 1000 simultaneous sessions was evaluated.**

ber of simultaneous sessions that the web server can handle while meeting its QoS requirement. For each test, the load generator generates a number of simultaneous sessions, as specified by the user, and it collects the response time statistics for each session. A test passes if 95 % of the pages return within `TIME_TOLERABLE` and 99 % of the pages return within `TIME_GOOD`, where `TIME_TOLERABLE` and `TIME_GOOD` are specified by the benchmark and they represent the QoS requirement. To find the maximum number of sessions, we have to try a number of choices of the number of user sessions until we find one that passes the QoS requirement. The traffic generator is hosted in Amazon EC2 since our Labs’ WAN network is not fast enough to support high traffic simulation.

### 4.1 Scalability of S3

In the client-side load balancing architecture, the system that delivers the JavaScript as well as the static content becomes a single point of contact for all clients. Therefore, the scalability of the overall architecture relies on the scalability of the hosting platform.

In this section, we assess the scalability of Amazon S3. Since the SPECweb2005 benchmark dynamically generates the web pages, we cannot evaluate S3 directly using the benchmark. Instead, we host a large number of static files on S3, and we launch a number of EC2 *m1.small* instances, each has 10 simultaneous TCP sessions sequentially requesting these files one by one as fast as it is able to. Figure 2 shows the aggregate throughput as a function of the number of *m1.small* instances. As shown in the graph, S3 throughput increases linearly. At 100 instances, we achieved 16 Gbps throughput. Since we are accessing S3 from EC2, the latency is all below the `TIME_GOOD` and `TIME_TOLERABLE` parameters in SPECweb2005.

### 4.2 Latency Impact

As the client-side load balancing (CLB) architecture requires clients to hit S3 to fetch a JavaScript file, it adds

	S3	CloudFront
Comcast network	93ms	45ms
Stanford campus	82ms	4ms
Work network	237ms	139ms
EC2	172ms	4ms

**Table 2: Download latency from several edge locations**

an extra round trip delay. Even though the architecture is scalable—maintains reasonable latency per request regardless of the number of clients—it should keep the extra round trip delay short enough to achieve the QoS requirement.

To evaluate the impact of S3 round trip, we use SPECweb-2005 *banking* workload with 1000 simultaneous sessions in three different setups: 1) direct communication between a client and a web server without a load balancer as a baseline, 2) CLB with S3 round trip per every HTTP request as a naive implementation, and 3) CLB with S3 round trip only once per new session, which is the current implementation described in Section 3.

As shown in Table 1, S3 round trip adds about 154 milliseconds of delay on average, which contributes 7.7% to the TIME\_GOOD QoS requirement (2 seconds). However, when we apply the optimization that makes a S3 round trip only when it creates a new session,<sup>1</sup> the extra latency becomes negligible; the latency variation originating from EC2 instances or cloud components’ dynamic performance variation is greater than the measured difference.

Even though we perform the SPECweb test only in EC2, we believe the latency is not a problem even when accessed from outside of Amazon. Unfortunately, we do not have access to a large number of external servers to perform load testing from outside of EC2. Instead, we evaluate the latency from a small set of external servers.

We measure the time it takes to download the JavaScript file which includes both the load balancing logic and the server load information. Along with the HTTP header, the file is about 1500 bytes. We use Wbox [17] to test the download latency.

We perform the test from the following locations:

- From a home network connected to Comcast’s cable network in Sunnyvale, California.
- From our office network located in San Jose. Our traffic is routed to the Chicago head quarter before it is sent to the Internet.
- From a server located on Stanford University campus.
- From within Amazon EC2.

Table 2 shows the average latency for 10 runs of downloading the same 1500 bytes anchor file from both S3 and CloudFront – Amazon’s content distribution network.

As expected, our office network experiences the longest delay due to its routing through our Chicago office. The Comcast network and Stanford campus both experience a much shorter delay. However, it is surprising to note that

<sup>1</sup>Based on our measurement, SPECweb2005 *banking* makes 4.41 requests per session on average.

	Sessions	Latency (sec)
Single node	1,000	0.407
12 node cluster	12,000	0.403

**Table 3: Latency comparison: a client-side load balancer adds negligible overhead to per request latency**

EC2 experiences a delay that is on the same order of magnitude as our office network. Therefore, our evaluation in EC2 is a realistic lower bound.

The Amazon CloudFront offering enables S3 to be used as geographically distributed content distribution servers. One can simply enable CloudFront on a bucket by issuing a web service API call, and the content is automatically cached in geographically distributed servers. In Table 2, we also show the latency from CloudFront. The delay is always significantly smaller than its S3 counterpart. We note that CloudFront currently has a longer time scale (minutes to create a distribution). Since we are aiming at a milli-seconds time scale for load balancing and failure recovery, CloudFront is not yet suitable for our client-side load balancing architecture.

The results in Table 1 show that, even using S3 directly (instead of CloudFront), the latency added by the additional round trip is acceptable. It is a small fraction of the 8 second rule [13]; thus, it is unlikely to impact the user perceived performance.

### 4.3 Overall Scalability

Finally, we assess the overall scalability of client-side load balancing with the SPECweb2005 *banking* workload. We measure the average latency per request both on a single web server without load balancing serving 1,000 simultaneous sessions; and on a cluster with 12 web servers serving 12,000 simultaneous sessions.

Table 3 shows that the cluster sustains roughly the same average latency per request as the single web server. During the experiment for the cluster, the 12 web servers experience roughly the same load, suggesting that the load balancing algorithm can effectively smooth out the load. Based on S3’s scalability as measured in Section 4.1, we believe that client-side load balancing can further scale beyond 12,000 sessions, but we stop here because of economic constraints.

## 5. CONCLUSIONS

A cloud is an attractive infrastructure solution for web applications since it enables web applications to dynamically adjust its infrastructure capacity on demand. A scalable load balancer is a key building block to efficiently distribute a large volume of requests and fully utilize the horizontally scaling infrastructure cloud. However, as we pointed out, it is not trivial to implement a scalable load balancer due to both the cloud’s commodity business model and the limited infrastructure control allowed by cloud providers.

In this study, we propose the Client-side Load Balancing (CLB) architecture using a scalable cloud storage service such as Amazon S3. Through S3, CLB directly delivers static contents while allowing a client to choose a corresponding back-end web server for dynamic contents. A client makes the load balancing decision based on the list of back-end web servers and their load information. CLB

uses JavaScript to implement the load balancing algorithm, which not only makes the load balancing mechanism transparent to users, but also gets around browsers' cross-domain security limitation. Our evaluation shows that the proposed architecture is scalable while introducing a small latency overhead.

## 6. REFERENCES

- [1] Accoria. Rock web server and load balancer. <http://www.accoria.com>.
- [2] Amazon Web Services. Amazon Web Services (AWS). <http://aws.amazon.com>.
- [3] V. Cardellini, M. Colajanni, and P. S. Yu. Dynamic load balancing on web-server systems. *IEEE Internet Computing*, 3(3):28–39, 1999.
- [4] L. Cherkasova. FLEX: Load Balancing and Management Strategy for Scalable Web Hosting Service. *IEEE Symposium on Computers and Communications*, 0:8, 2000.
- [5] F5 Networks. F5 Networks. <http://www.f5.com>.
- [6] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – http/1.1. In *IETF RFC 2616*, 1999.
- [7] Google Inc. Google App Engine. <http://code.google.com/appengine/>.
- [8] HaProxy. HaProxy load balancer. <http://haproxy.1wt.eu/>.
- [9] G. Hunt, E. Nahum, and J. Tracey. Enabling content-based load distribution for scalable services. Technical report, 1997.
- [10] E. Katz, M. Butler, and R. McGrath. A scalable HTTP server: The NCSA prototype. In *Proc. First International Conference on the World Wide Web*, Apr. 1994.
- [11] H. Liu and S. Wee. Web Server Farm in the Cloud: Performance Evaluation and Dynamic Architecture. In *Proc. of the 1st International Conference on Cloud Computing (CloudCom 2009)*, Dec 2009.
- [12] Nginx. Nginx web server and load balancer. <http://nginx.net/>.
- [13] Z. Research. The need for speed II. [http://www.keynote.com/downloads/Zona\\_Need\\_For\\_Speed.pdf](http://www.keynote.com/downloads/Zona_Need_For_Speed.pdf).
- [14] SPEC. SPECweb2005 benchmark. <http://spec.org/web2005/>.
- [15] A. Vahdat, T. Anderson, M. Dahlin, E. Belani, D. Culler, P. Eastham, and C. Yoshikawa. WebOS: Operating System Services for Wide Area Applications. In *Proc. of the 7th IEEE International Symposium on High Performance Distributed Computing*, pages 52–63, 1998.
- [16] A. Vahdat, M. Dahlin, P. Eastham, C. Yoshikawa, T. Anderson, and D. Culler. WebOS: Software Support for Scalable Web Services. In *Proc. the Sixth Workshop on Hot Topics in Operating Systems*, 1997.
- [17] Wbox HTTP testing tool. <http://www.hping.org/wbox/>.
- [18] C. Yoshikawa, B. Chun, P. Eastham, A. Vahdat, T. Anderson, and D. Culler. Using Smart Clients to Build Scalable Services. In *Proc. USENIX 1997 Annual Technical Conference*, Jan 1997.