ECE6102 Dependable Distribute Systems, Fall2010

EWeb: Highly Scalable Client Transparent Fault Tolerant System for Cloud based Web Applications

Deepal Jayasinghe , Hyojun Kim, Mohammad M. Hossain, Ali Payani

Introduction

Cloud computing services built around virtualization are revolutionizing the computing landscapes by making unprecedented levels of computing power cheaply available to millions of users. However, this scalability cannot be leveraged unless the application does scale well, hence understanding scalability issues and providing architectural solutions to better utilize computing cloud are challenging tasks. In addition, shared computing clouds (e.g., Amazon EC2) consist of higher degree of communication and node failures compare to conventional datacenters, and thus, new tools and approaches are needed to build reliable and robust systems. In our semester project, we design a new architecture to better utilize associated scalability of computing cloud and to provide client transparent fault tolerant system for Web applications.

In our project, we design and implement EWeb system with following four major goals: (1) high scalability, (2) dynamic load balancing, (3) fault tolerance, and (4) low overhead. One of the major challenges of better utilizing associated benefits of computing clouds is difficulty of finding necessary software applications. For example, most web applications are configured to use Apache HTTPd as the web frontend and commercial or free applications servers (e.g., Apache Tomcat) at the application tier. To best of our knowledge, Apache HTTPd does not support dynamic reconfiguration (adding and removing application servers), which is major limitation achieving infinite scalability in clouds. In our approach, we design and implement a generic load balancing components for HTTP based communication systems. In particularly, our architectures follow Multi-Master-Replication, where load balancer (LB) acts as a proxy, which forwards the message to only one replica and sends the response to the client. We use multicasting mechanism to notify the LB when adding or removing replicas, and thus, LB will smoothly tolerate to system changes, and act accordingly. Figure 1 shows the overall architecture of proposing system with its components as well as the connections among the components.

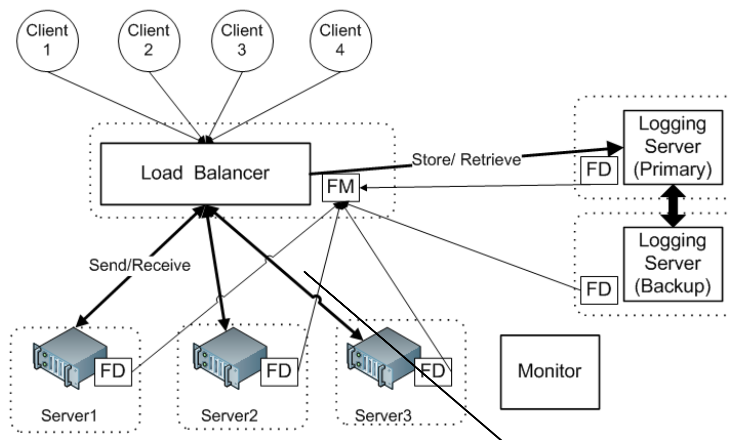

Figure 1. EWeb System Architecture

Design and Implementation

Load balancer

The load balancer is the frontend facing components in our system. External users only see the load balancer, and send to or receive from messages from the load balancer. Existence of application servers and other fault tolerance mechanisms are transparent to the users. Our load balancer differs from other traditional load balancers in following points: first, it gathers current system monitoring data, and forwards the messages to the

servers with low workload. Second, it can detect newly added servers automatically. Third, it can detect server failures (hardware or software), and fourth provides client transparent fault tolerance. Finally, our load balancer can be easily migrated to another server in the presence of system failures.

Operations of the load balancer

Figure 2 shows the sequence of operations for both normal and failures cases. The figure on the left hand side shows the sequence of a normal execution, and the right hand side figure shows the operation sequence when there is an application server failure.

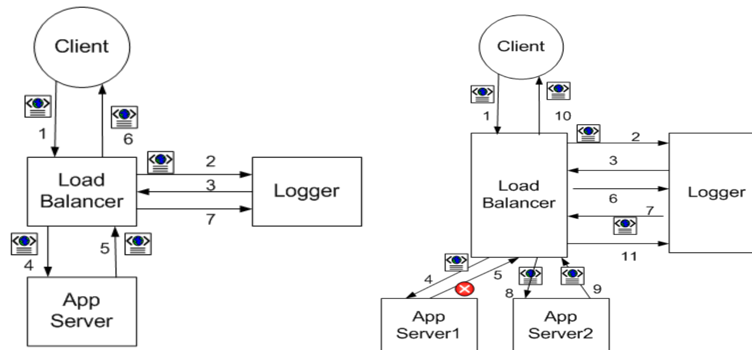

Figure 2. Operations of the load balancer

- Normal execution
  1. Load balancer receives a message from a client.
  2. Then, it sends the client message to the logging server (LogStore).
  3. LogStore returns the generated message key for the client message.
  4. Load balancer picks an application server, and forwards the message to the server.
  5. Application server processes the client request, and sends back the generated response.
  6. Load balancer forwards back the response message to the client.
  7. Finally, load balancer asks LogStore to remove the client message.

When load balancer detects an application server fault either by fault detector module or response timeout from the application server, load balancer runs following protocol.

- Operation sequence when there is an application server failure (steps 1 to 4 are same as above)
  5. Application server fails to reply, and load balancer detects server failure.
  6. Load balancer asks LogStore to return the saved client message with the given key.
  7. LogStore returns the stored message.
  8. Load balancer picks another application server, and forwards the message to the server.
  9. Application server processes the client request, and sends back the generated response.
  10. Load balancer forwards back the response message to the client.
  11. Message is removed from LogStore.

LogStore

Load Balancer needs to put client requests in a safer place until while the requests are being processed. If the web server fails, Load Balancer retrieves saved request, and redirects to other available server. We have designed Logging Server, named LogStore. It is connected via TCP/IP network with the load balancer. LogStore

has no mechanism for fault tolerance, but LogStore itself can be duplicated in EWeb system (active replication). According to the required fault tolerant level, a client request can be sent to multiple logging servers, and can safely be retrieved from one of the logging servers.

Architecture and Collaboration

LogStore consists of two modules: Interface and storage module. Interface module is in charge of TCP/IP communication with Load Balancer and storage module provides the functionality of simple key-value storage. Interface module opens a socket, and spawns a thread for every incoming connection request. Storage module uses a simple hash table to search data from a key.

LogStore Protocol

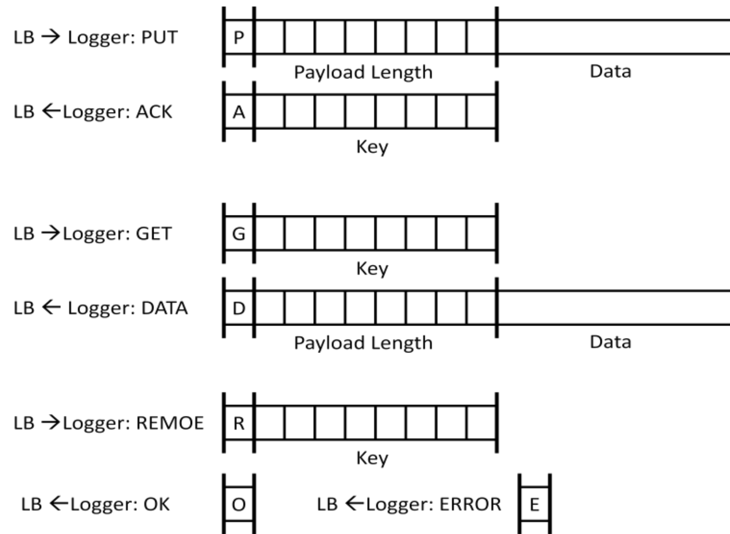

Figure 3. LogStore Protocol, LogStore provides three operations: PUT, GET, and REMOVE

Fault Detector and Fault Manager

The fault detection and management component of EWeb is based on a push monitoring system. In the push monitoring model, A Fault Detector (FD) component on each application server or LogStore repeatedly reports to the Fault Manager (FM) to confirm its aliveness. The keep-alive packet encapsulates following resource statistics of each live node, which is transmitted over UDP socket to the FM.

node type	cpu idle	memory used	memory free

- *Node Type* is any of the following four types of node in the system:
  WEB_SERVER, LOG_SERVER, LOAD_BALANCER, GUI
- *CPU Idle* value represents instantaneous cpu utilization in a range 0 – 1000
- Memory Used represents instantaneous memory usage (in KB).
- Memory Free represents instantaneous available memory (in KB).

The FM depends on a user-defined timeout to detect a node failure. When the first message arrives at FM from a node (either application server or LogStore), it enters the information (IP addres, resource statistics etc.) of that node in the active node-list, and sets up a timer for that node. If a node gets failed and does not send a keep-alive packet within the specified timeout interval, the timeout happens for the timer allocated to that node. The signal handler routine then removes that node from active node-list. In the normal operation, when a node is alive, a

successive message arrives before the timeout occurs. Then, the information for the node is updated, and a new timeout interval is set up for that node.

The above fault detection mechanism naturally imposes that a slow or congested node may be in late to respond within the threshold timeout value, and could be detected as "faulty" by the FM. To cope up with this scenario, we could incorporate a complex mechanism to handle the timeout interval parameter. We could dynamically adjust the timeout intervals for each node. In such design, each node's timeout interval would be a function of inter-arrival latency of keep-alive messages.

The FM component communicates with the load balancer (LB) and the GUI component over UDP to disseminate a snapshot of resource utilization statistics of the active nodes. To send a request to the FM, the LB or the GUI sends a UDP packet to the FM repeatedly. The format of this request packet:

node type	,	request number

On response to a request, the FM encapsulates information of all live nodes in the distributed system, along with their statistics. The response packet keeps node information in decreasing order of cpu idle value. The response packet format:

node count(n)	-	Statics for node 1	…	Statistics for node n	newline

Each node statistics portion of the response packet encapsulates following resource utilization information for each active node:

addr	type	cpu_idle	mem_used	mem_free	request count	last request time	---

FD / FM Implementation

The code is developed in C++ under Linux kernel 2.6.18. G++ compiler is used to run the programs. There is a limitation of using maximum three system timers at a time in Linux. However, the POSIX library provides a mechanism to support multiple timers in software.

Web Monitoring GUI

This is the user interface of the system that can be used to monitor the system functionality and performance. The monitoring software connects to the Fault Manager component of the system using UDP and periodically obtains a list of currently working web servers as well as the Load Balancer itself. As you can see in Figure 4, the GUI consists of 5 different parts;

- **System View:** shows that the current status of the systems, whether it is working or not, and the latest time the data had been updated. Using the start Recoding button, we can record all the data for each node in a text file for further analyses, the log data will be saved for each of the servers separately in files with names corresponding to the node's IP.
- **Node List :** shows all currently working web servers in the system, for each node it shows the CPU load for that node as well as used and free memory , request count and the time of the last request for that node
- **Event Log :** This view shows each of event in the system like event corresponding to connecting and disconnecting to Fault Manager , also it shows when a new node join the system or when it leaves.
- **Graph:** In this part of the view the CPU usage for each of the nodes in the system is plotted continuously so we can monitor the workload for each server.
- **Properties:** In this part of the view, the information about the load balancer as well as the total number of servers and the request counts will be showed.

Design and Implementation

This user interface program is written for windows platform using C++ and MS MFC 8.0 (Visual Studio 2008). All the views have been created to be dock able and resizable so user can arrange the screen the way it is more convenient for example we can maximize the graph so to see the details of the graph easier . Each of the views can be minimized or docked as well. All the socket programming has been implemented using Winsock API 2.2. All the drawing is also done using MFC drawing classes.

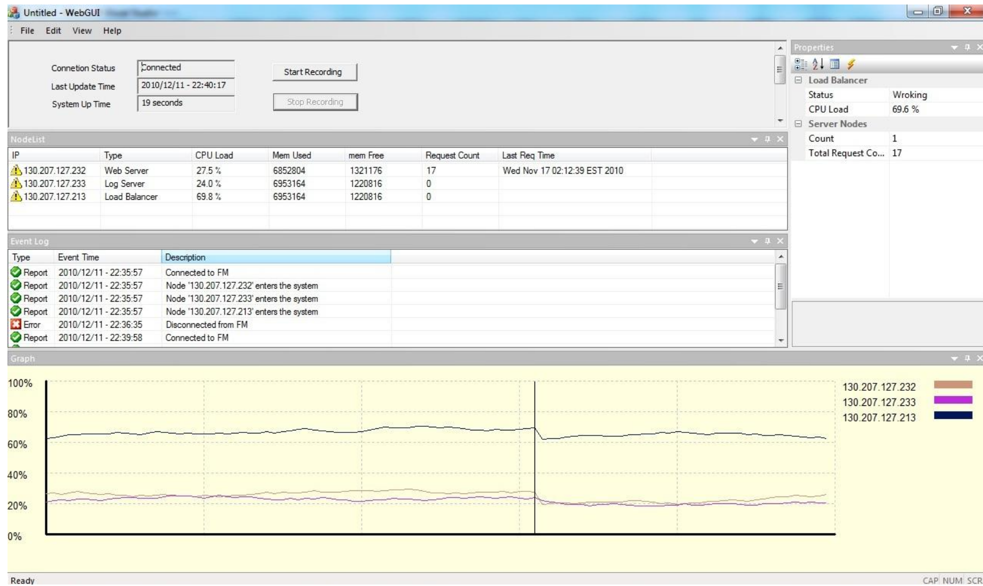

Figure 4. Screenshot of Web Monitoring GUI

Evaluation

Testing Environment

In this section we present our experiment results. We present the experiment platforms and software packages used for our experiments.

Type	Amazon EC2	Emulab
Processor	Intel(R) Xeon(R) 2.66GHz (Dual core)	Intel(R) Xeon(TM) CPU 3.00GHz
CPU speed	2659.992 MH	2992.898 MH
Memory	7 GB	2 GB
Platform	64 -bit	64-bit
Operating System	Fedora core 10	Fedora core 10

Table 1: Platform configurations

Software Package	Version
JDK	1.5.0_07
Apache Tomcat	5.5.17
Apache HTTPd	2.0.54
Apache Axis2	1.5

Table 2: Software packages and versions

In our evaluation, we used a dedicated server for each application. For example, when we have a configuration with one Apache server and three Tomcat servers, then we four servers are used in total.

Workload executor

We have developed a workload executor as a multi-thread program, and we programmed the number of threads to be configured. By increasing the number of concurrent threads, we can increase the workload. Our workload executor is a simple Web service client, which invokes a Web service using REST manner. One of the requirements we had was to generate workloads from different physical nodes, hence in our experiment, we have deployed the workload executor on five different physical machines, and used shell scripts to started and stop multiple workload executors simultaneously.

For each individual configuration, we configured ramp up time, execution time, and ramp down time to 7, 12, and 3 minutes, respectively. For each run, we stop and reset all servers by removing log files. Servers are started with fresh setup always.

Notation: We use ***EWeb*** to denote observed results when we use our approach as the load balancer, in contrast, ***Apache*** to denote the observed results when using Apache HTTPd as the frontend load balancer. In addition, we also use EWeb-ppp to differentiate Emulab and EC2. For EC2 we use ppp as ec2, and for Emulab we use ppp as em.

In the following graphs, we have only given the observed throughput and response time on a single client. However, actual system throughput should be five times the values shown in the figures. Similarly, in all the graphs we have changed the workload from 20 to 200. However, since we have five clients, actual workloads were from 100 to 1000 users.

Evaluation on Emulab testbed

In our approach, we started with two application servers, and increased the number of application servers up to four servers. For each configuration, we measure the response time and throughput. Figure 5 shows the observed response time and throughput for both using EWeb and Apache HTTPd. As shown in the figure, for lower workload our approach performs better, and shows considerably higher throughput and lower response time. In contrast, when the workload goes up, our approach shows lower throughput compared to Apache HTTPd, however response time still remains less than Apache HTTPd. One of the key reasons we have observed was *the effect of garbage collection on EWeb load balancer*.

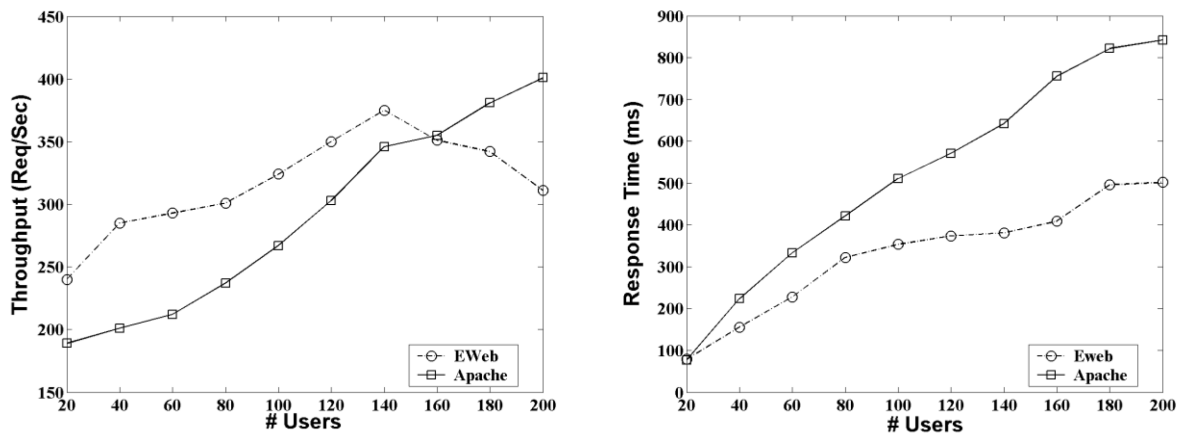

Figure 5: Throughput and response time with two Tomcat servers in Emulab

We then increase the number of application servers by one, and perform the same set of workload against the new configurations. The observed results are shown in Figure 6. As shown in the figure, for lower workload

EWeb shows better performance compared to that of Apache. However, for the higher workload Apache shows higher throughputs.

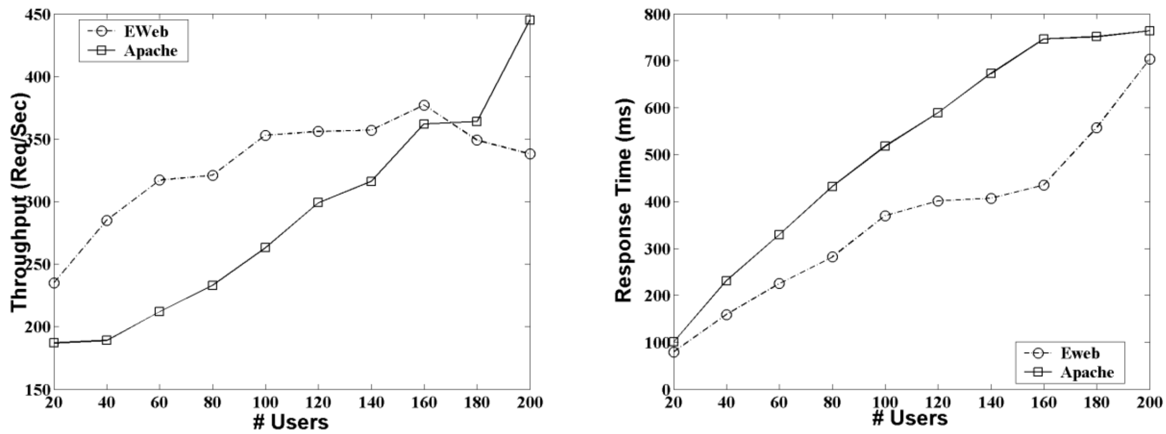

Figure 6: Throughput and response time with three Tomcat servers in Emulab

Similarly, we increased the number of application servers to four, performed a set of experiments against the new configuration, and the results are shown in Figure 7. As shown in the figure, the throughputs of Apache and EWeb are similar when workloads are low, and Apache performs better than EWeb for the higher workload. However, EWeb shows much shorter response time than Apache in all the configurations.

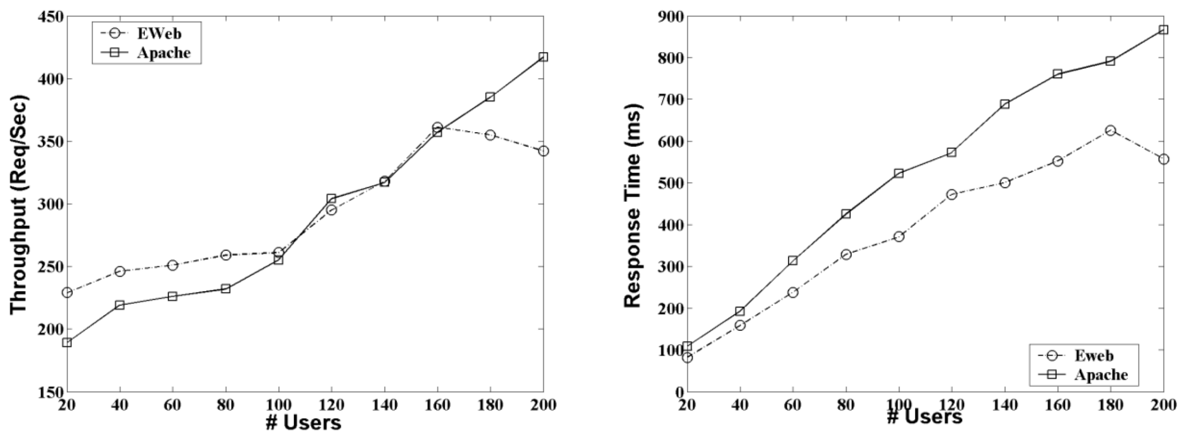

Figure 7: Throughput and response time with four Tomcat servers in Emulab

Evaluation on Amazon EC2

Perhaps the best known example of commercial compute clouds is Amazon's Elastic Compute Cloud (Amazon EC2), which enables users to flexibly rent computational resources for use by their applications. Amazon EC2 provides resizable computing capacity in the cloud, including the number of instances (horizontal scalability) and different types of nodes (vertical scalability, with the choices such as Small, Large, Extra Large, High-Memory Extra Large). Amazon EC2 also provides provisioning control of their virtual computing resources (compute nodes). However, direct privileged access to the physical hardware is not exposed to users, which additionally complicates system analysis.

In our experiment, we selected large instances, and performed similar set of experiments as in Emulab. Notably, here in Amazon EC2, we only perform experiment using our approach, and compared the performance on EC2 and Emulab. This was mainly due to the associated cost. Our experiments show similar results in Emulab and Amazon EC2. However, response time of EC2 experiments is higher than that of Emulab experiments.

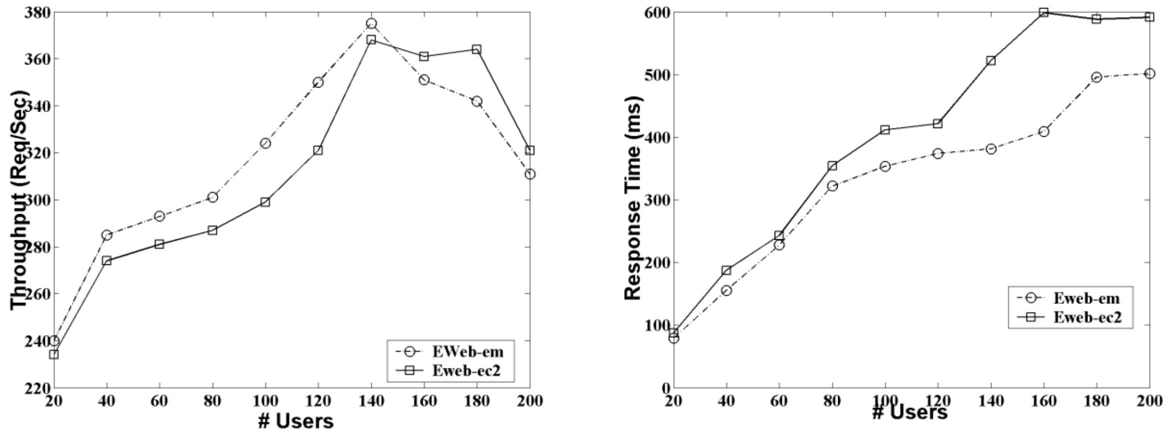

Figure 8: Throughput and response time with two Tomcat servers in EC2

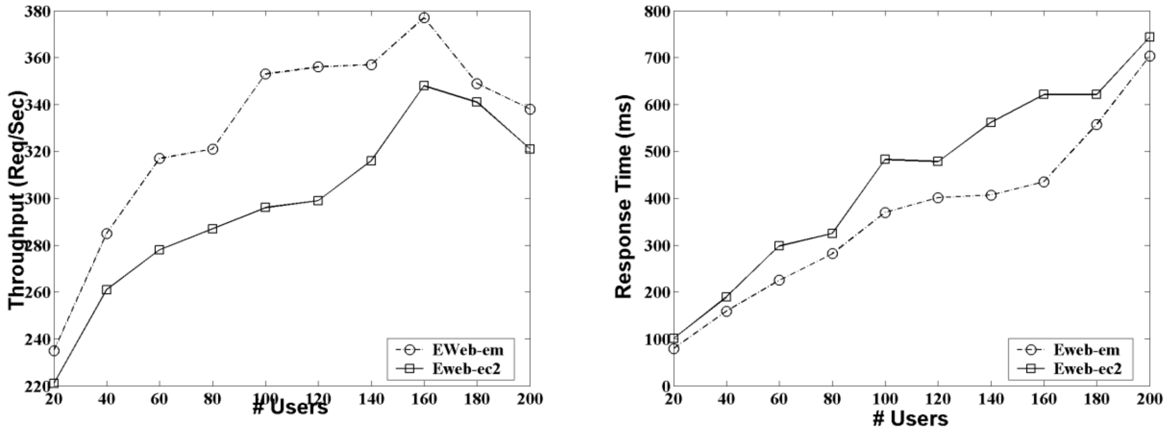

Figure 9: Throughput and response time with three Tomcat servers in EC2

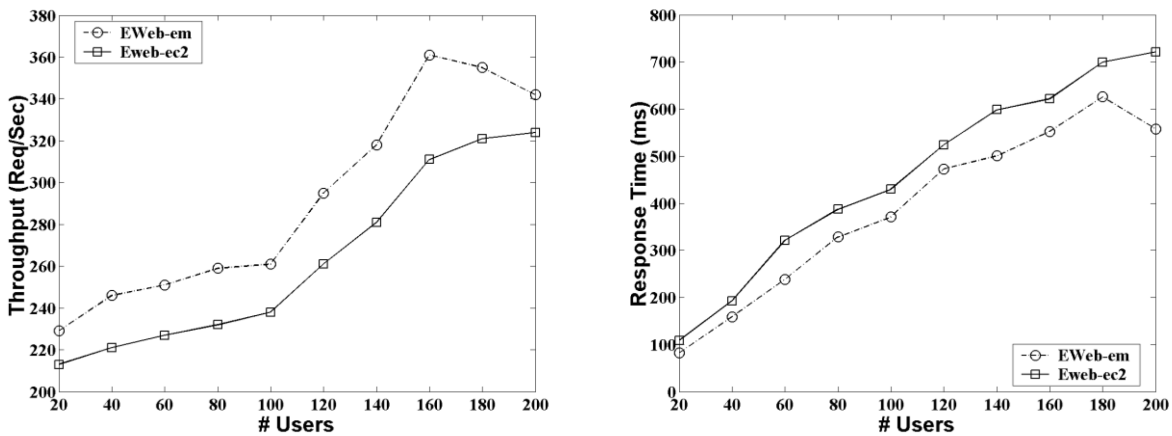

Figure 10: Throughput and response time with four Tomcat servers in EC2

Discussion and Limitation

In our system, all messages go through load balancer. Therefore, the log balancer may have high load, and the availability of the whole system is depends on the availability of the load balancer. In other words, the load balancer can be the single-point-of-failure. To resolve this issue, we have designed all system components to be modular, and each system component can be easily added or removed, dynamically. We can easily replace the fault manger, LogStore, and even the load balancer with new servers.

Conclusion

In this project, we have designed and implemented EWeb system, which is highly scalable and client transparent fault tolerant for cloud based web applications. Five major components have been implemented: 1) load balancer, 2) LogStore, 3) fault detector and manager, and 4) application monitor. We have performed extensive evaluation studies on two real cloud computing systems: Emulab and Amazon EC2. According to our experimental results, EWeb system shows higher throughputs and shorter response time when workloads are low. When workloads are heavy, the throughput of EWeb system becomes lower, but response time is always shorter than Apache HTTPd.

Goals and Achievements

Goal	Achievements
High Scalability	**Achieved.** In EWeb, system is constantly monitored and determined to add additional servers or to remove existing servers to adapt to the requested demand.
Dynamic Load Balancing	**Achieved.** FD module monitors system utilization, and reports it to FM. Load balancer schedules user tasks based on collected information.
Fault Tolerant	**Achieved.** EWeb provides client transparent fault tolerant model by combining node and link status monitoring and message logging mechanisms.
Low Overhead	**Achieved.** Our evaluation results are showing that EWeb system has been implemented to have very low overhead.

Work Breakdown

Deepal Jayasinghe	Overall system design Load balancer Implementation and documentation Whole system integration Performance evaluation and documentation
Hyojun Kim	LosStore protocol design LogStore implementation and documentation Final report integration
Mohammad M. Hossain	Fault Manager /Detector protocol design Fault Manager/Detector implementation and documentation
Ali Payani	Application Monitor (GUI) design and implementation GUI documentation