Automatic Scaling of Cloud-Based Web Applications

Walter Blaurock (wblauroc@cs.brown.edu)

Undergraduate Capstone - Spring 2011

Abstract

There exist many cloud-computing platforms, such as Amazon's EC2, which allow a developer to quickly add and remove server instances on an hourly basis. Despite this capability, no widespread solution exists for automating the scaling process; instead, periodic monitoring and human intervention is required to add or remove instances. I propose a solution that responds to an increase or decrease in traffic by following user-configurable rules, avoiding the need for human intervention beyond the initial setup. The result is the potential for significant savings, both of time and money, while ensuring a good user experience.

1 Introduction

Cloud computing is quickly becoming the default solution for modern web applications. It promises to be highly scalable and has an à la carte pricing model in which the developer pays just for the compute instances in use. While these claims are true, human intervention is still necessary in order to determine the optimal number of instances. Additionally, the performance of the system must be monitored over time to ensure the best performance-to-cost ratio. If an application increases in popularity, it may make sense to add some new instances to increase performance; if an application decreases in popularity, it would beneficial to reduce the number of instances in order to save money.

The situation described up to this point could be solved by daily or even weekly monitoring of the performance of the application, manually adding and removing instances whenever necessary. However, nearly all of the modern cloud computing solutions allow for much finer granularity. They are capable of bringing new instances online within a matter of minutes, and they charge for them on an hourly basis. This opens up the door to significant savings in cost and resources if the application could respond to increases and decreases in traffic immediately. Take for example a website that uses Extra Large instances from Amazon's EC2, priced at $0.68 per hour [1]. The highest level in traffic is reached at noon, requiring the application to use 10 instances to stay reasonably responsive. The cost to run this application is then $163.20 per day. However, traffic drops significantly between 6pm and 6am, requiring only 5 instances to maintain the same level of responsiveness. If the application could drop 5 instances every day during this time, $40.80 could be saved per day in hosting costs.

The issue with the above example is that users are not this predictable. Dropping from 10 to 5 instances may be reasonable at times, but cannot be relied upon. In order to ensure an optimal balance between performance and cost, constant monitoring of the system is necessary to avoid relying upon predictions, instead allowing the system to respond in real time to actual changes in traffic.

2 Solution

In order to solve the problem of responding to changes in traffic autonomously, I propose and implement a cluster monitoring application, aptly named ClusterMonitor [2], to both monitor the application and to add

or remove instances when necessary. It should be noted that ClusterMonitor is a proof of concept used as a basis for the experiments that follow; therefore, it is lacking many features that would commonly be found in a production application, including fault tolerance. However, it is built with flexibility in mind, based on the following goals. Cluster monitor should be:

- platform agnostic: it should be adaptable to any distributed system and make no assumptions about the underlying architecture.
- metric agnostic: it should make no assumptions about available metrics and rely completely upon the developer for acquisition of these metrics from the individual instances, avoiding any reliance upon a particular monitoring solution.
- rule agnostic: a set of rules provided by the developer will use the available metrics in order to determine when an instance should be added to or removed from the cluster.
- able to support multiple clusters: if a developer has both an application cluster and a database cluster, there should be no need to run two instances of ClusterMonitor.

2.1 ClusterMonitor API

In order to implement the aforementioned goals, ClusterMonitor both exposes an API and requires the developer to provide—for each instance—a reference capable of communication with the provider and physical instance. A reference to all available instances must be registered with ClusterMonitor, even if they are not currently running; this limits ClusterMonitor to using at most the number of instances registered, and is a safeguard against large monthly bills. These references must implement the interface `PhysicalHandle`, the methods of which are described below, grouped by goal.

It should be noted that in its current state, ClusterMonitor does not provide load balancing. In the experiment below, a number of shell scripts were written to communicate with the Apache module mod_proxy_balancer through its web interface. I made this choice to allow ClusterMonitor to be used with any application, not just a web server. However, because I have already written these scripts, it would be possible to bundle compatibility with certain load balancers (such as Apache's) with ClusterMonitor, to lessen the work on behalf of the developer.

2.1.1 Platform Agnostic

The following methods of `PhysicalHandle` pertain to access to the cloud computing provider and are called on the provided reference:

- startServer: the instance should be brought online and enabled (see enableServer). This operation should block.
- stopServer: the instance should be disabled (see disableServer) and brought offline. This operation is not required to block.
- enableServer: the instance should be confirmed running, brought online if not (see startServer), and then added to the load balancer group. This operation should block.
- disableServer: the instance should be removed from the load balancer group. This operation is not required to block.
- isRunning: returns whether or not the instance is currently running. Used for getting the initial state of the cluster and for error checking.

- isEnabled: returns whether or not the instance is a member of the load balancer group. Used for getting the initial state of the cluster and for error checking.

2.1.2 Metric Agnostic

The following methods of `PhysicalHandle` pertain to access to the individual physical instances and are called on the provided reference:

- getPerformanceMetrics: CloudMonitor will periodically request the current value of all metrics referenced in the rules (see Section 2.1.3). The reference is expected to return all metrics, but is permitted to leave blank any metrics unobtainable at this time.

2.1.3 Rule Agnostic

A call to `ClusterMonitor.addRuleForCluster` is made for each rule governing the addition and removal of instances. As an example, the following rules were used for the experiments run below:

metric	comparison	threshold	duration	action
load	>	10.0	15000	add
load	>	5.0	30000	add
load	>	2.0	300000	add
load	<	0.5	30000	rem

Table 1: ClusterMonitor rules.

Translated into English, the first rule states that if the load is greater than a value of 10.0 for more than 15,000 consecutive milliseconds, an instance should be added to the cluster. Load here is a system metric (as seen in the standard unix program `top`) and could easily be replaced with or augmented by other metrics such as a free memory, network usage, or disk throughput. Only one rule is allowed to evaluate to true at a time, and a configurable refractory period is imposed between successive evaluations of rules to allow the system to equilibrate if possible, avoiding the potential for rapid increase or decrease in cluster size.

2.1.4 Multiple Clusters

In order to support multiple clusters, each instance and rule is associated with a unique cluster identifier provided by the developer. Because of this, any number of clusters can be monitored at the same time.

3 Experiment

To an end user, the only important metric is response time. It doesn't matter what goes on behind the scenes, or how many machines are involved in crafting a response. According to a study by Akamai [9], if the user doesn't see a page within approximately 4 seconds, they are going to leave and are significantly less likely to return. Therefore, the following tests will all focus on the metric of response time as seen from the perspective of a client. To accomplish this, clients are simulated using a variety of tools, and collected data are correlated with system performance metrics acquired from the server instances; through this, it is possible to calculate the expected response time without the need to actively monitor it. Tests begin with a single-instance cluster—the simplest possible setup—to ensure that a basic understanding is achieved before complicating things with more instances. Once the behavior of a single-instance cluster is well understood, rules can be crafted and the performance of ClusterMonitor tested to ensure that it does automatically scale

the size of a cluster to handle changes in traffic in a cost-effective manner and with minimal interruption to clients using the application.

3.1 Setup

In order to run these experiments, a test system is necessary. WordPress (3.1.1), an open-source blog implementation [3], was chosen as the test application, as it is a reasonable approximation of an average modern web application. To run WordPress, Apache (2.2.16), PHP (5.3.3), and MySQL (5.1.49) were used. The OS powering all of the test instances was Ubuntu (10.10, 32-bit server).

To simulate server hardware, VMWare Fusion (3.1.2) was installed on a dual 2.26GHz Quad-Core Intel Xeon processor machine with 16GB of memory. The following test instances were created, each running on their own virtual machine (VM):

```
apache0  Apache node.           2GB memory    1 core
apache1  Apache node.           2GB memory    1 core
apache2  Apache node.           2GB memory    1 core
lb       Apache Load Balancer.  512MB memory  1 core
db0      MySQL master.          512MB memory  1 core
db1      MySQL slave.           512MB memory  1 core
```

Table 2: Server instances.

To simulate a client, VMWare Fusion was installed on 2.4GHz Intel Core 2 Duo processor machine with 4GB of memory. A separate physical machine from that running the server VMs was used to ensure that traffic was traveling over a network and to avoid any possible interference between client and server VMs. The following test instance was created:

```
stats    Client node.           512MB memory  1 core
```

Table 3: Client instances.

The WordPress application is replicated across `apache0`, `apache1`, and `apache2`. Client requests are directed not to these instances but instead to `lb`, which is running the Apache module mod_proxy_balancer. This module provides load balancing across all available Apache instances. When a new request is received, this module will pick the least-used Apache instance and obtain from it the response on behalf of the client. To ensure that client sessions are maintained across multiple requests, a cookie is set by the module so that successive requests for the same client can be made to the same Apache instance. This module makes it easy to enable and disable Apache instances in the cluster, allowing for simple testing of one-, two-, and three-instance clusters.

3.2 Single-Instance Analysis

The first goal was to demonstrate that a cluster comprised of one Apache instance could be easily and consistently overloaded. To accomplish this, autobench [4], a wrapper around httperf [5], was used. Httperf is a workload generator that is given an endpoint to which requests are made (e.g. http://www.google.com/), a number of requests to make in total, and a concurrency level specifying how many requests should be made per second. Autobench can be configured to issue requests with an increasing level of concurrency, which is useful when a quick overview of performance is desired. As seen in Figure 1, a cluster of one Apache instance is capable of handling between 1 and 4 concurrent requests with equivalent response times; however, after a rate of 5 requests per second is reached, the response time begins to increase.

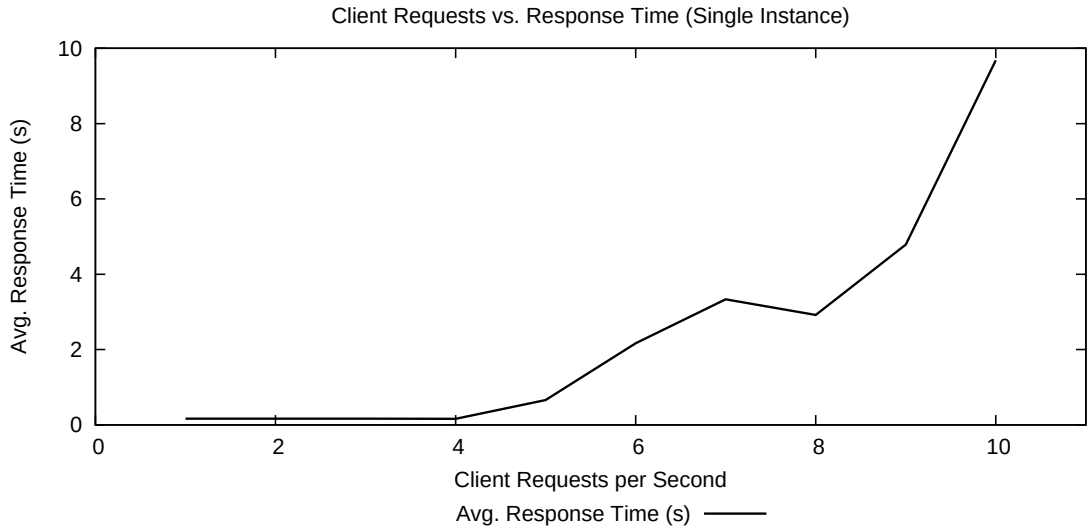


Fig. 1: autobench is used to issue an increasing number of concurrent requests to a cluster comprised of a single Apache instance. The level of concurrency is plotted against the response time.

Following these results, a small change to the configuration on `lb` allows two Apache instances to share incoming requests. It should be expected that this change will allow the cluster to sustain higher levels of concurrency than in the single-instance test. As Figure 2 verifies, the cluster of two instances is now able to serve 9 requests per second before an increase in response time is observed.

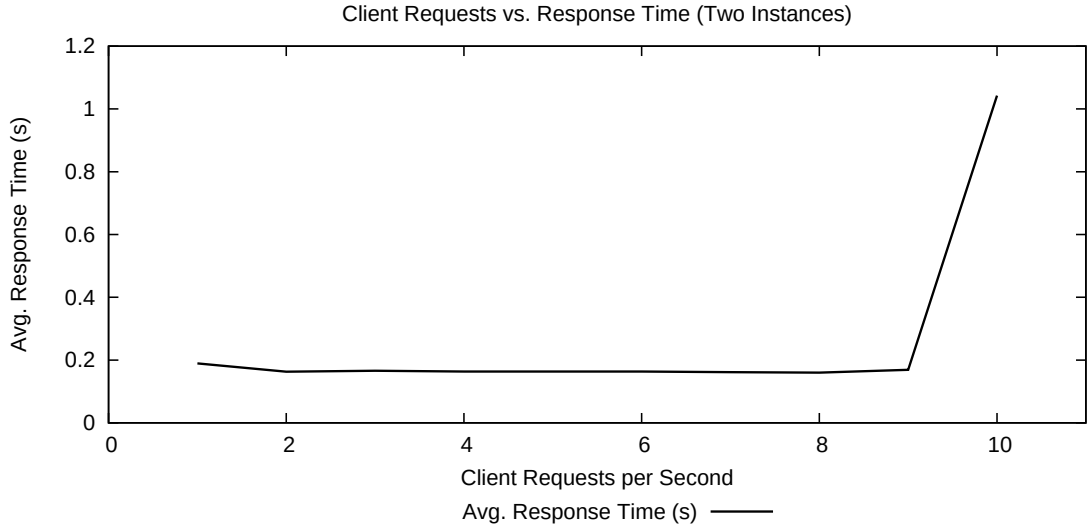


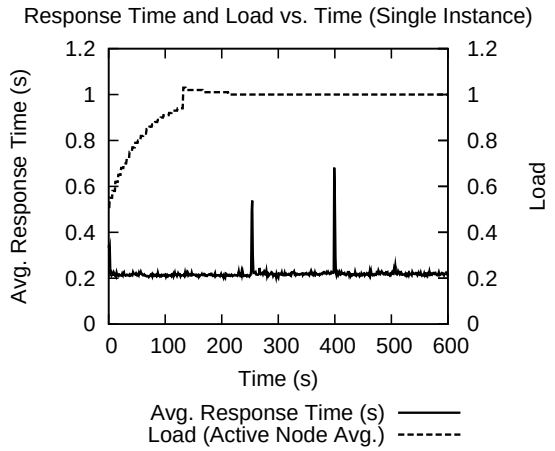
Fig. 2: autobench is used to issue an increasing number of concurrent requests to a cluster comprised of two Apache instances. The level of concurrency is plotted against the response time.

While this verifies that we can overload a cluster comprised of a single Apache instance, and that adding a second instance to the cluster approximately doubles its capacity, it does not tell us what is causing the increase in response time. In order to explore this further, collectd [6], a daemon which collects various system performance metrics (e.g. load, free memory), was installed on the server instances. Collectd periodically records the value of all system metrics for which it is configured and logs them to a file.

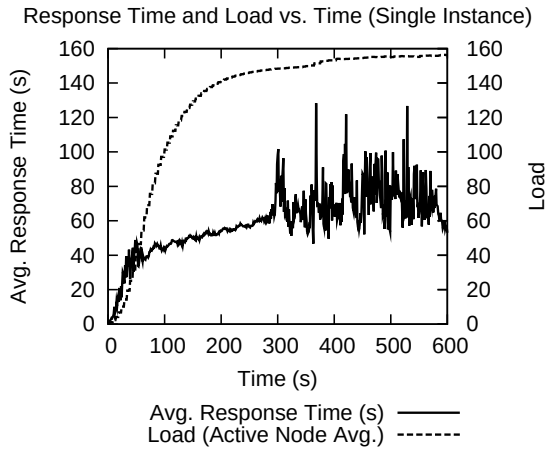
With this new data, it is possible to correlate system metrics with response time. To begin, two common bottlenecks were chosen: load and memory. Autobench was again used to generate requests to a single-instance cluster. The response times output by autobench were then correlated with the values for load and free memory. Though not shown here, it was obvious that memory was not a bottleneck with WordPress—regardless of how overloaded the instance was, memory usage would rarely exceed 30%. Load, on the other hand, correlated strongly with response time: response time began to increase when load exceeded a value of 1. This makes sense, as load is roughly defined as CPU utilization. Once the CPU utilization exceeds 1, the CPU is no longer able to process all runnable tasks, instead forcing some to wait; the response time will necessarily increase.

Before continuing, it should be noted that load will be the focus of the remainder of this paper only because it was found to be the best predictor of response time for WordPress running on this particular hardware. It is possible for another web application to be bound by other resources, such as memory, disk throughput, or network capacity. In those cases, the metric(s) in question (e.g. free memory, data read per second, or data sent per second) would likely be used in place of load. Because ClusterMonitor was designed to be metric-agnostic, this is not important; simply swap load for any other metric appropriate for the application, and everything that follows would still hold. It is also possible that some combination of metrics (e.g. a rule using load and a rule using memory usage) would best predict the response time of the application. Again, this was not done here because only load was found to be a reliable predictor of response when running WordPress.

While autobench is great for obtaining an overview of performance, httperf by itself is necessary to examine how the cluster behaves in the long run when given a constant number of requests per second. However, httperf only summarizes the data it collects—there is no way to get a real-time stream of response times as responses are received. Luckily, thanks to it being open source, a quick modification resulted in a log file of all response times, perfect for more detailed analysis.



(a) 4 requests per second. Response time is steady and load hovers around 1.



(b) 5 requests per second. Response time increases to unusable levels and load peaks at 160.

Fig. 3: httperf is used to generate requests to a cluster comprised of one Apache instance, and response time and load are plotted over time. Note the vast difference in scale between (a) and (b).

Figure 3 shows two plots of response time and load plotted over time and is the result of two 10 minute tests run on a cluster containing a single Apache instance. These plots give an explanation, measurable on the server, for the behavior seen in Figure 1. Given 4 requests per second, an instance is able to handle each request upon receipt, evidenced by the fact that load reaches but does not surpass 1. However, when the same

instance is given 5 requests per second, the load quickly increases beyond 1 to a value of nearly 160, taking response time with it; the response time averages nearly 80 seconds by the end of the second test. Upon more detailed analysis, the threshold after which load increases beyond 1 was found to be approximately 4.92 requests per second. Any number of requests per second below this threshold could be handled for at least 10 minutes, while any number of requests above this threshold resulted in rapidly increasing response times, up to the same value of 80 seconds. This limit of 80 seconds appears to be attributed to the fact that Apache can only have so many sockets open at once; new requests will simply be ignored, and as a result will not slow down Apache further.

3.3 Cluster Analysis

I have shown that a single instance can be predictably overloaded (i.e. more than 4.92 requests per second will result in unacceptably large response times), and we have found load to be a reliable server-side predictor of response time. With this accomplished, it is now possible to test the effectiveness of ClusterMonitor. However, before running the application, rules have to be decided—at least one rule for the addition of an instance and one for the removal of an instance. The rules chosen were already presented above but are repeated once more to avoid unnecessary cross-referencing.

metric	comparison	threshold	duration	action
load	>	10.0	15000	add
load	>	5.0	30000	add
load	>	2.0	300000	add
load	<	0.5	30000	rem

Table 4: ClusterMonitor rules.

Since ClusterMonitor supports the use of multiple rules, three different rules for `add` were included. The first `add` rule is in place to catch large spikes in load that happen nearly instantaneously. The second `add` rule is used to catch a smaller spike in load that happens quickly. Finally, the third `add` rule is used to catch a slow increase in load that could result from a small but significant increase in traffic. This final rule is the one most likely to evaluate to true, but the other two can remain in as safeguards. These rules have durations significantly less than durations that might be in place in a production system—this is due only to the desire to obtain results relatively quickly, allowing for more comprehensive testing.

In order to demonstrate the effectiveness of ClusterMonitor in a real-world setting, an approximation of diurnal traffic patterns or those from a spike in traffic from a website such as reddit.com is desired. While the traffic pattern seen in Figure 4 is not entirely realistic, it does include a rising and falling section and was straightforward to implement with a threaded wrapper around httperf. (Threading was necessary to ensure that each successive call to httperf is made at precise intervals, and not upon completion of the previous call.) The test lasted for just over an hour, remaining at each step for 180 seconds. The test was run with 3 Apache instances: 1 active instance along with 2 additional instances ready to be brought online by ClusterMonitor as dictated by the rules in Table 4.

In addition to the traffic pattern, Figure 4 also contains a graph of cluster size over time, showing that ClusterMonitor was executing as planned. Shortly after 5 requests per second was reached, one of the rules from Table 4 evaluated to true and a second instance was brought online. This was predicted by the analysis done in Figure 3, as the threshold of 4.92 requests per second was crossed. A third instance is brought online shortly after 10 requests per second was reached. Again, this was predicted by Figure 3, as each of the two active instances was receiving 5 requests per second, putting them both above the threshold of 4.92 requests per second. On the opposite end, ClusterMonitor was a bit slow to remove instances, but this is likely the fault of the (perhaps overly) cautious `rem` rule from Table 4, as a load of less than 0.5 has proven to be hard

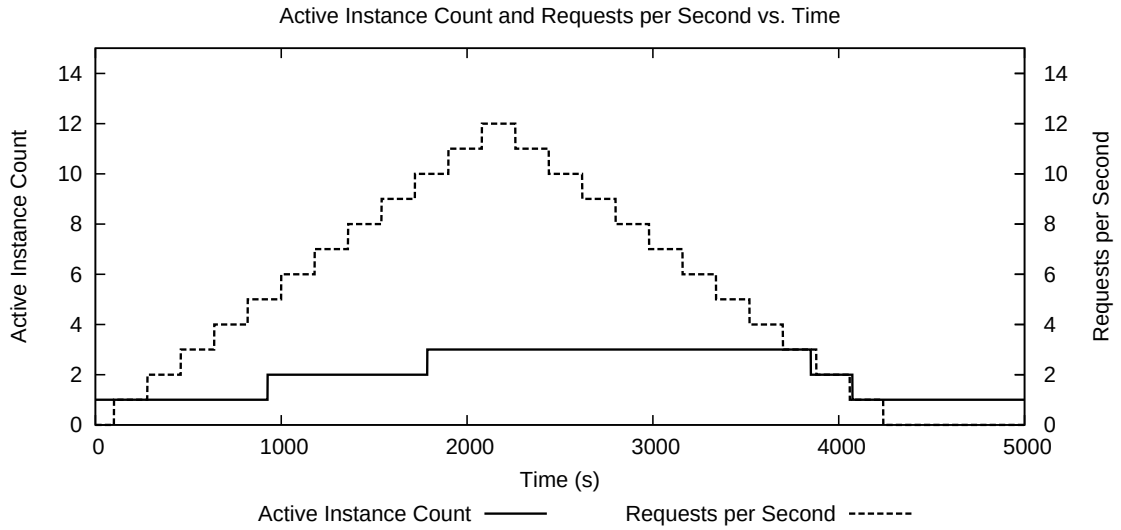
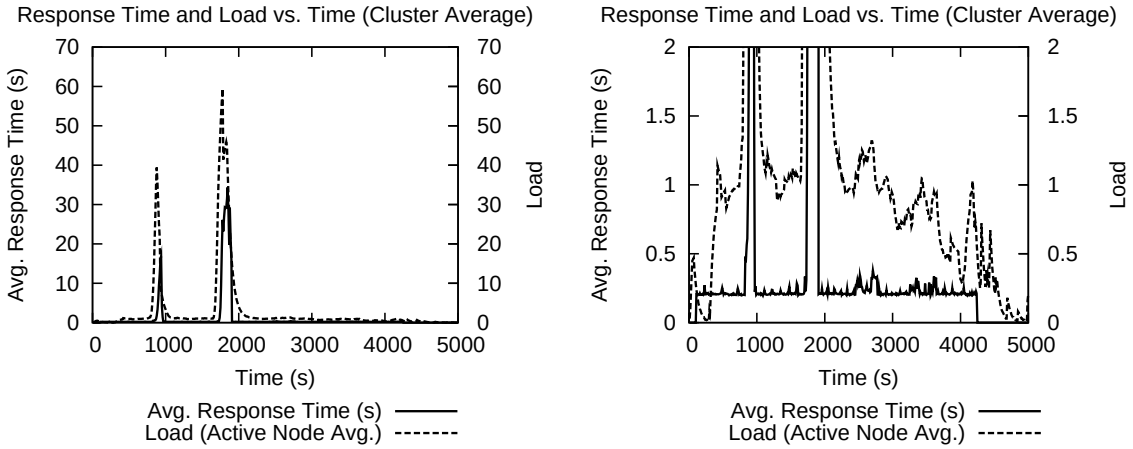


Fig. 4: as the number of requests per second increases, ClusterMonitor adds 2 additional instances to the cluster; as the number of requests per second decreases, ClusterMonitor removes 2 nonessential instances.

to obtain even when the number of requests per second drops to 1. This rule evaluated to true for the first time nearing the end of the 3-request-per-second step and was evaluated to true again approximately 180 seconds later (the default refractory period), leaving the cluster with 1 active instance.

While Figure 4 demonstrates that ClusterMonitor was effective, there is more analysis to be done. Was ClusterMonitor able to do its job with minimal interruption to clients actively using the application? Sadly, the answer is not quite. As Figure 5 shows, there were two brief periods of time during which the response time was above our acceptable maximum of 4 seconds per request. The first period lasted for 61 seconds and reached a maximum response time of 18.50 seconds and averaged 10.92 seconds, while the second period lasted for 153 seconds, reaching a maximum response time of 38.46 seconds and averaging 23.54 seconds.



(a) Full view of response time and load.

(b) Magnified view of (a).

Fig. 5: a custom wrapper around httperf is used to send an increasing and then decreasing number of requests per second (see Figure 4) to the cluster. The two large spikes are approximately where ClusterMonitor added two instances. Value for load is averaged across all active instances.

There are two plausible explanations for this behavior. The first comes from Figure 5(b): there is almost no warning of the rapid increase in load before either of these periods. Without some sort of early warning, it is nearly impossible to write robust rules to detect these spikes before they happen. The second explanation comes from an analysis of the log generated by ClusterMonitor: there was a significant delay when obtaining metrics from the active instances during periods of increased load, causing ClusterMonitor to hang while waiting for a response. A more desirable behavior would have been to implement a timeout or to collect metrics in some parallel fashion, allowing ClusterMonitor to continue despite unresponsiveness. Because a slow response from the individual instances is likely indicative of a large spike in load, ClusterMonitor could even infer possible load values, allowing for the introduction of a new instance when it is likely needed most. That being said, Figure 5 shows that for a majority ($\approx$95%) of the test, the response time hovered around 200ms.

4 Related Work

While there are numerous solutions of this nature available today, most are proprietary systems useable only with one product. For example, Amazon has two: Amazon Auto Scaling [7] and Amazon Elastic Load Balancing [8]. The first allows a single instance to scale between different capacities, while the second allows a set of instances to work together, adding and removing instances in much the same way as ClusterMonitor. However, both of these systems work only with Amazon EC2.

Zeus also provides a solution, improving upon Amazon's offerings by not integrating with any particular service [10]. However, a developer must still pay for Zeus, and the technology is not open source.

There are numerous other examples of solutions that fall between these two, but to my knowledge no open source project exists that allows automatic scaling of cloud-based clusters in both a language- and provider-agnostic way.

5 Conclusion

ClusterMonitor was designed with four goals in mind (Section 2). It accomplishes the first goal with the use of an interface through which ClusterMonitor can talk to the cloud computing provider. In this case, a set of shell scripts was used to interact with the Apache module and the VMs. ClusterMonitor achieves the second goal through a similar interface to the underlying instances; collectd was used here, but this was an arbitrary choice. The third goal has already been discussed in depth; ClusterMonitor, through its use of rules, allows the developer to specify exactly how the cluster should respond to changes in system performance. Finally, the fourth goal is accomplished by associating every available instance and server with a cluster identifier, allowing any number of clusters to be monitored simultaneously. The end result is that ClusterMonitor is able to autonomously monitor and scale a cluster based on changes in traffic. Through this, it is able to approximate the ideal number of instances for the application at any given point in time, saving significant amounts of time and money for the developer while ensuring a good user experience.

6 Future Extensions

ClusterMonitor as it stands today relies on human-generated rules to add or remove instances from a cluster. As evidenced by Figure 5, these rules can result in a system that is good at observing its current state, but bad at predicting its future state. While certainly an issue, this is not a fundamental flaw in ClusterMonitor—given an appropriate set of rules, it is likely that ClusterMonitor could use these rules to predict with a high degree of accuracy its future state. The key is then to extend its capabilities from simply following rules to

generating them. It is easy to imagine a setup process for each application and/or cluster during which a series of client tests, not unlike those presented above, would be run in conjunction with the collection of server-side metrics, again not unlike those collected above, in order to build up a database of exactly what kind of conditions on the server side lead to an increase in response time on the client side. This process could generate an intricate set of rules not easily producible by humans. Additionally, continuous logging of system metrics on all server instances could be used to retroactively determine the cause of unpredicted spikes in response time. In this way, ClusterMonitor could evolve the set of rules to suit the changing needs of the application—for example, perhaps it is naïve to assume that the same set of rules will apply to a cluster of 3 instances and a cluster of 300. In the current implementation, rules would have to be changed manually if this were the case; in the future extension, ClusterMonitor would automatically introduce new rules and phase out old ones if and when the need arose.

References

[1] http://aws.amazon.com/ec2/pricing/.

[2] https://github.com/walterblaurock/cluster-monitor.

[3] http://wordpress.org/.

[4] http://www.xenoclast.org/autobench/.

[5] http://www.hpl.hp.com/research/linux/httperf/.

[6] http://collectd.org/.

[7] http://aws.amazon.com/autoscaling/.

[8] http://aws.amazon.com/elasticloadbalancing/.

[9] Akamai. Boosting online commerce profitability with akamai. http://www.akamai.com/dl/whitepapers/WP_ECOMM_ROI.pdf.

[10] Zeus. Zeus elastic application delivery platform. http://www.zeus.com/sites/default/files/users/documents/Zeus_Elastic_Application_Delivery_Platform.pdf.