Cost Efficiency versus Quality of Service in Auto-scaling Web Services

Jakub Zárybnický

December 6, 2017

1 Introduction

The topic that I've received, IT services, is a very wide one, with many possible interpretations. The problem that I've chosen to focus on is: "How to provide maximum availability of an auto-scaling service while being as cost-efficient as possible?" The platform that I assume is a web-service that uses a simple high-availability solution: a single load-balancer, behind which there are several replicas (or instances) of the application server. Each running replica costs money (a fixed cost per second, as is common in big cloud service providers), and we're trying to minimize our costs while providing a good user experience.

The topic of this work was inspired by my personal experiences and several interesting problems in my day job. Most of the information presented in the Analysis section were things that I've slowly accumulated over my IT career, and any references that I'd find would be retrospective. Nonetheless, I've attempted to back non-conceptual facts by references.

While creating an analytic model of this system would be possible, I'd have difficulties representing it in its entirety with my current mathematical knowledge. The 'solution' we're looking for here are several parameters of the auto-scaler algorithm: upper and lower limits on the number of active replicas and the utilization thresholds (which when crossed will prompt the auto-scaler to either start or stop a replica).

As a specific reference for this work, outside of my own previous experience, I've read several published papers, namely:

- `https://www.researchgate.net/publication/261759640_A_queuing_theory_model_for_cloud_computing` solves a similar problem, focusing on different parameters.
- `http://ieeexplore.ieee.org/document/6194446/` solves a similar problem using analytical methods.

(I apologize for the inline links, but the way I've used to compile to LaTeX doesn't currently support bibliographies.)

2 Analysis

I assume that an auto-scaling web service is composed of two parts - a load balancer, the publicly exposed part of the service which accepts requests from clients (web browsers, other web services, ...); and a number of replicas of the application itself, which use identical code, and differ only in nominal factors.

A real web service also has several other parts - a DNS server, which may choose to direct users to different load balancers (e.g. based on their geographical location, `https://www.digitalocean.com/community/tutorials/how-to-configure-dns-round-robin-load-balancing-`
a database, which is a difficult system to model; operating system factors (from socket number limits to virtualization overhead); hardware factors (network connection throughput, CPU performance), ... I've chosen to disregard these, as a complete model would be likely very complex, and to focus on my chosen issue.

The load balancer accepts any requests it receives, and forwards them to one of application replicas based on several factors (round-robin, average CPU load, ...; `https://kemptechnologies.com/load-balancer/load-balancing-algorithms-techniques/`)

An application either accepts a request directly if it has a free processing slot, or it stays queued, from where it may be either accepted by the application at a later time, or time out if it doesn't get accepted in time.

The application likely implements some variant of concurrency to handle multiple requests simultaneously - from having several OS worker processes (Nginx, `https://www.nginx.com/blog/thread-pools-boost-performance-9x`) to using purely select/kqueue/epoll (Node.js) to green threads (Warp, `http://www.aosabook.org/en/posa/warp.html`).

A replica (application instance) takes some time to start before it is ready to process requests from the load balancer. This number varies greatly depending on the type of replica - spinning up a server may take 45s (`https://serverfault.com/questions/179714/fastest-time-to-get-an-ec2-instance-running`), starting a simple Docker container takes less then 0.5s on my machine (as the kernel the container uses is already running), and starting a worker process is nearly instantaneous. A replica, while being shut down, is disconnected from the load balancer immediately, as it cannot process requests while shutting down.

2.1 Technologies

The implementation environment used is the SIMLIB library presented in demonstration lectures, as it was the most accessible one to me, given the limits on implementation languages (C/C++).

However, given the specific requirements of my solution, I've needed to slightly edit the official SIMLIB library - namely the Store class, as I've encountered several limitations of its official SetCapacity, Enter, and Leave methods. I've included the patched SIMLIB library in its entirety (excluding the fuzzy sub-module), and it is compiled alongside my model.

The main structure of the solution was inspired by SIMLIB example programs and specific bits that were directly reused are marked in the source code.

3 Model

Two main parameters that the model must represent is quality-of-service and cost of the web service, as those are the two factors we're trying to optimize.

I've chosen to represent quality-of-service as the time spent in the queue while waiting before a request starts being processed. This disregards a number of factors - the time spent processing a request, the time spent waiting for the database, ... These factors will be included as a fixed number, but to simplify the model, I won't specify them in detail - as simulating e.g. database congestion, processor utilization or network throughput is a large topic, and I'll focus on replica scaling.

The auto-scaler is represented as a periodic event that starts and stops replicas based on the current utilization of the system, constrained by the upper and lower limit on the number of replicas. Utilization is calculated as the sum of the number of requests being processed and requests in queue divided by the number of available replicas.

The auto-scaler representation includes two other assumptions or simplifications. One is the queue of requests - the model I'm using contains a single queue shared among the available replicas, whereas in a real-world system, each instance would have its own queue. The other is the nature of an application instance - in this model, a replica is represented by a synchronous process that takes requests in order - this most closely resembles the processing model of PHP, or worker processes of Nginx. On the other hand, a replica takes a fixed amount of time to start, and shuts down instantaneously - that resembles starting and stopping a virtual machine or a container. I could model a replica as one SIMLIB Store, but that would require more bookkeeping and more patching of SIMLIB itself.

The auto-scaler algorithm itself - while necessarily simplified, due to the absence of CPU utilization - works on the same principles as real ones. After a system utilization threshold has been crossed (too high or too low), an instance is either started or stopped. However, due to the scales used here, I've had to allow starting several instances simultaneously, as the system was often too slow to respond to sudden bursts of requests - given that a single replica usually represents a machine instance which can handle hundreds of requests per second at least, and here a replica handles its queued requests sequentially and synchronously.

The set of replicas is then represented as a single Store with variable capacity - that is what has required me to patch SIMLIB, as the built-in Store implementation didn't support lowering of Store capacity.

I also assume that the number of replicas that we can potentially use is unlimited - in practice, the number will be very high, but limited by e.g. the provider's hardware - but this fact is unimportant, as I expect that in most simulations, we'll have a fixed cap on the number of replicas.

The load-balancer and replicas are then assumed to be ideal, without any failures

(hardware or software), though this wouldn't affect simulation results much, given the short time-frame of a single run and the reliability guaranteed cloud providers which usually ranges from 98% to 99.999% (e.g. `https://www.forpsicloud.cz/vseobecne-podminky.aspx`).

As for requests themselves, they have two parameters: their processing time, which I'll model as fixed, and their timeout - most web browsers use very large timeouts (on the order of minutes, `https://stackoverflow.com/questions/5798707/browser-timeouts`), so apart from a few simulation runs, I won't be using timeouts.

4 Implementation

The model itself consists of three parts: one Store representing the set of running replicas (as described in the previous section), one auto-scaler process (a periodic event), which manipulates the replica Store's capacity and calculates current run-time costs, and a request generator (or several of them).

I've included several request generator types - steady generator, which generates requests spaced by fixed delay, a simple random generator, which generates requests along an exponential distribution; burst generator, which generates bursts of requests along an exponential distribution - one burst is an evenly spaced sequence of requests; varying-intensity generator uses an exponential distribution as well, but the mean time between requests oscillates between two values (a fixed-size step after every request).

As mentioned in the 'Technologies' section, I've had to patch the SIMLIB library itself, due to limitations on the Store, namely on setting its capacity (and I was unable to extend it, as my extended class wouldn't be a 'friend' of Entity.

5 Experimentation

The goal of the following experiments is to find out the ideal values of the upper and lower limits on the number of replicas, and the 'scale up' and 'scale down' thresholds of the auto-scaler.

In the course of experimentation, I've had to gradually edit the auto-scaler algorithm as described in the 'Model' section, as the original transcription of `https://docs.wso2.com/display/ELB210/Demonstrating+the+Auto-Scaling+Algorithm` didn't perform very well in my test runs, mainly due to the simplifications described above.

I'll be using several combinations of parameters and request generators. The input parameters are: request timeout, request processing time and replica start time. The input parameters to be optimized are the auto-scaler parameters: upper and lower limit on the number of replicas, scale-up and scale-down thresholds. These will change as well, but I'll observe their relationship with the QoS and accumulated cost of the system.

Some expectations: I expect that given a steady non-zero request stream, the number of replicas will stay stable and the utilization will be inside the required bounds. For a burst of requests, the number of replicas will jump to a high enough number that after the start-up period, the replica set will be able to efficiently process the rest of the burst,

and then will be scaled down. For a steadily changing request generator, the average queue time won't jump very high and the replica set will adjust up and down as required.

In all simulations, the cost of running a single replica for one simulation tick will be equal to 1. Having a replica number equal to 0 means unlimited, whether upper or lower.

The target value 'efficiency' represents the quality-of-service and cost combined. While I'm sure I could find better metrics, one that has worked out well in preliminary tests was a simple product of the cost and average queue time divided by 1000, so that is the one I'll use in the presented results (given my time constraints). For this metric - same as for QoS and cost - lower means better.

What follows is a tabular summary of input parameters and the average queue time and accumulated cost. The following parameters were fixed: timeout = 0, processing time = 10, start-up time = 100.

5.1 Steady request generator (spacing = 1)

Lower	Upper	Scale-down	Scale-up	Queue time	Cost	Efficiency
0	0	0.1	0.9	0.47	48411	22.75
1	0	0.1	0.9	0.22	27379	6.02
2	0	0.1	0.9	0.46	27071	12.45
0	0	0.5	0.6	0.08	33148	2.65
0	0	0.6	0.7	0.13	29320	3.81
0	0	0.7	0.8	0.22	25470	5.6
0	0	0.8	0.9	0.38	25360	9.64
0	0	0.9	1.0	0.38	23512	8.93
0	0	1.0	1.1	0.66	19904	13.14
0	12	0.9	1.0	0.39	23490	9.16
0	11	0.9	1.0	0.51	21598	11.01
0	10	0.9	1.0	7.76	19684	152.75
0	9	0.9	1.0	55.36	17748	982.53
0	8	0.9	1.0	102.86	15812	1626.42

The cost differences in the first three rows are due to the large difference in scale-up and -down thresholds - the auto-scaler cannot accurately capture the utilization of the system, and it is mostly start-up conditions that affect the number of replicas that run after the system achieves a steady state.

I was unable to set the utilization target higher than 100% due to the way the auto-scaler works - this is an implementation problem and not a fundamental one. In any case, I expect the queue times would rise drastically.

Most of the avg. queue time differences are due to start-up conditions, as requests back up at the start, before the auto-scaler ćan start enough replicas to correctly process all incoming requests as they come.

I've experimentally determined that the number of replicas that can optimally process one request per tick is ten (obvious in retrospective, given that the number of ticks to process one request is ten) - in fact, if the starting number of replicas was 10, then the average queue time would be zero!

Two results can be gained from this set of runs - for a known size of the request stream, it pays off to set up your parameters precisely. This rarely happens in real life - at least for public services - but being aware of this can save money.

5.2 Random request generator (mean spacing = 1)

Lower	Upper	Scale-down	Scale-up	Queue time	Cost	Efficiency
0	0	0.4	0.5	0.06	37720	2.26
0	0	0.4	0.6	0.10	35894	3.59
0	0	0.4	0.7	0.14	34530	4.83
0	0	0.4	0.8	0.14	32506	4.55
0	0	0.4	0.9	0.31	31274	9.69
0	0	0.4	1.0	0.36	29030	10.45
0	0	0.5	0.6	0.21	31186	6.55
0	0	0.5	0.7	0.22	30504	6.71
0	0	0.5	0.8	0.27	29360	7.93
0	0	0.5	0.9	0.54	27974	15.11
0	0	0.5	1.0	0.60	26610	15.97
0	0	0.6	0.7	0.39	27776	10.83
0	0	0.6	0.8	0.41	27116	11.12
0	0	0.6	0.9	0.63	26610	16.76
0	0	0.6	1.0	0.87	24938	21.7
0	0	0.7	0.8	0.62	25048	15.53
0	0	0.7	0.9	0.91	24542	22.33
0	0	0.7	1.0	0.95	23948	22.75
0	0	0.8	0.9	1.20	23090	27.71
0	0	0.8	1.0	1.41	22562	31.81
0	0	0.9	1.0	1.77	21638	38.3

Now we're finally getting to more realistic scenarios. Having replicas as close to 100% utilization is more cost-efficient, as we've discovered in the previous section. However now, we see that such a system can't react to changes in request intensity as well as one that has wider thresholds. Here we can see that the final 'efficiency' metric that I've chosen isn't very configurable - a more realistic approach could be to set a avg. queue time cutoff (or perhaps median - which I'm not using simply because SIMLIB doesn't calculate it by default), and sort the remaining configurations by cost.

The one takeaway from this set of runs is this: having a lower average utilization helps with handling sudden increases in request intensity (represented here by a random distribution, later investigated by burst and combination generators). However, as before, having replicas that are fully utilized drives the costs down. Looking for more patterns in the above data, it seems that having a buffer of about 20% (i.e. scale-up factor = 0.8) helps avoid the biggest impact of sudden increases - though this should be investigated further by repeating these runs, as having a single request generator means that they are quite dependent on even slight patterns in the random data.

5.3 Burst request generator (mean spacing = 50, burst length = 50, burst spacing = 2)

Lower	Upper	Scale-down	Scale-up	Queue time	Cost	Efficiency
0	0	0.5	0.8	0.16	30680	4.91
5	0	0.5	0.8	0.09	31054	2.79
10	0	0.5	0.8	0.08	33474	2.68
15	0	0.5	0.8	0.02	36708	0.73
0	30	0.5	0.8	0.17	30548	5.19
0	25	0.5	0.8	0.41	30020	12.31
0	20	0.5	0.8	0.89	28172	25.07
0	15	0.5	0.8	4.92	23552	115.88
5	30	0.5	0.8	0.09	30922	2.78
10	30	0.5	0.8	0.09	33342	3
15	30	0.5	0.8	0.02	36510	0.73

Starting with one of the more efficient auto-scaler configurations from the previous experiment, we try to find the effect of limits on the number of replicas.

In the first part, we can see that while having a higher number of stand-by replicas decreases the queue time, it increases the cost - though not as much as adding an upper limit in the second part decreases it. Having an unlimited lower limit means that at certain times, there is no replica running at all - and the effect of delaying the first requests can be seen on the queue time.

I've looked at the maximum amount of replicas running in the baseline case (which was 32), and tried to find out the effect of decreasing the upper limit. It has decreased costs quite markedly at the cost of a lot higher queue time. In the case of sudden bursts like the simulated one, a common technique used is either plain rate limiting or more sophisticated techniques that prevent (Distributed)-Denial-of-Service ((D)DoS) attacks - but this risk needs to be balanced with the possibility of a legitimate source of an increased request intensity (e.g. a marketing campaign or a product launch).

Based on these results, the best course of action seems to be setting up a small (but non-zero) lower limit, and setting up a higher limit several times higher than is the expected peak usage - and setting up a variant of (D)DoS protections.

6 Summary

From the results, we can deduce several recommendations - for a private web service, or a web service with well-known usage characteristics, setting up very tight auto-scaler thresholds that maximize replica usage is the only cost-effective way.

For public-facing web services or services without well-known or predictable usage patterns, lowering the lower auto-scaler threshold is a way to prevent small variations in request intensity from affecting QoS (namely lowering the maximum queue time).

Setting up a lower limit on the number of replicas - even a small one - prevents a very high response time for the first few requests.

Setting up some - likely quite high - upper limit on the number of replicas may guarantee some protection from excess costs in the case of (D)DoS and other attacks - depending on other available protections.