Estimation of Performance and Availability of Cloud Application Servers through External Clients

Sune Jakobsson

Department of Telematics
NTNU

Trondheim, Norway

sune.jakobsson@comoyo.com

***Abstract*— This paper investigates two aspects of the QoS offered by some cloud providers on the Internet, the availability and the dedicated capacity in terms of how well a user process is isolated from other users of the same application server instance. By using standard components and software utilities, a small external measurement client is able to gather the necessary information about the cloud application servers in question, and hence, addresses the measured availability and capacity over time. In summary, I show that the biggest cloud providers indeed demonstrate good isolation and availability.**

Keywords-component; QoS; Java virtual machines; garbage collection; application servers; availability

I. INTRODUCTION

More and more of our computing needs are being moved to the "cloud", but what does this really mean with respect to dedicated capacity, availability, and reliability? The terms used in this paper are defined in [6]. This paper describes a limited experiment with cloud providers on the Internet that commercially provides application server instances, with the objective to investigate the quality of service one can observe from these providers with a client connected to multiple providers as shown in Figure 1. The focus in this paper is on application availability, and to some extent dedicated capacity. Some major providers offer application server instances for free for a limited period of time and others provide them at a reasonable cost. None of the providers provide detail regarding their availability or their internal structure. The best one can find ahead of signing up are the relative up-time during the last period often without stating what the period is. So how can one assess their offer in reasonable time and at a reasonable cost?

Several papers point out failures on servers present on the Internet, [2, 10]. There are several papers addressing the isolation among virtual machines, [11], but in these particular cases there is no prior knowledge available of the underlying system, and hence, a different approach is needed. The approach chosen here is to probe live application servers, and collect externally available data from their operation. This paper presents the approach, and discusses what is possible to observe from the outside of the application server providers.

By analyzing just a few isolated parameters, one may yield significant conclusions regarding their behavior. In other papers related to this subject [1, 13], it is shown that it is sufficient to observe the response time and the amount of free memory in the application container to experimentally predict their long term behavior.

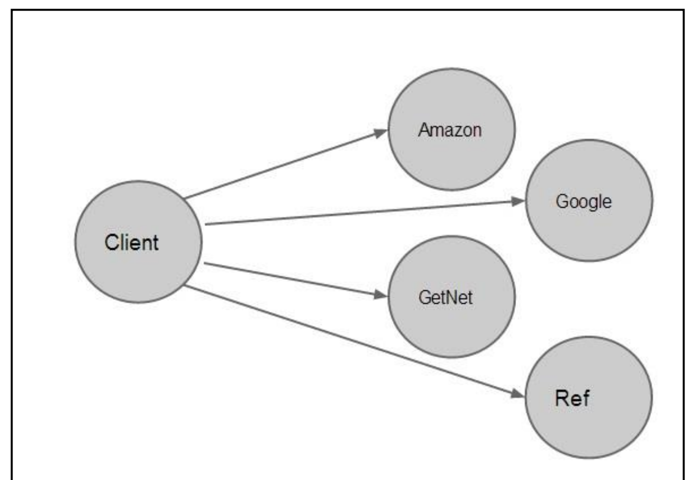

Figure 1. The measurement system.

For the current experiment, a simple client and server instance is developed and run for more than a month. The external client, in turn, invokes four application instances at different nodes, i.e., virtual machines provided as cloud services. All virtual machines are running with the same server software. An application instance is defined as an application container running the application software in a virtualized environment. The client measures the response time of the invocation and logs the amount of free memory reported by the respective application servers. The operating systems provide a millisecond clock on the client side for time stamps, and by choice the trial was run over a period of one month, sampling the unavailability at 2.5 second intervals to obtain enough samples to validate the claims. The amount of free memory from the servers is reported in bytes, and varies from instance to instance, but their overall behavior is similar. In Section II the measurement system is described. Section III gives some background on memory allocation. The experiment and the kind of results obtained are presented in Section IV. Finally, the results are presented with a discussion related to the offerings from the vendors in Sections V.

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-301-8

II. THE MEASUREMENT SYSTEM

One key element was to investigate cloud based servers and to see if the application server instances the providers provide are indeed isolated from other usage or not. Since Amazon EC2 [4] and Google App-server [3] are two big players in this domain, they were selected. To have some comparison and base line, a smaller vendor and a personal reference server connected to the Internet was part of the experiment as shown in Figure 1. All invocations were done from the same place, to avoid or filter out close and near network issues. With the chosen invocation rate of 2.5 seconds, unavailability of less than 2.5 seconds is not detected, see Figure 2. The client logs the result code of the invocation, the invocation duration time, and the reported server side free memory.

The client and the servers running on the virtual machines were all programmed in Java [7].

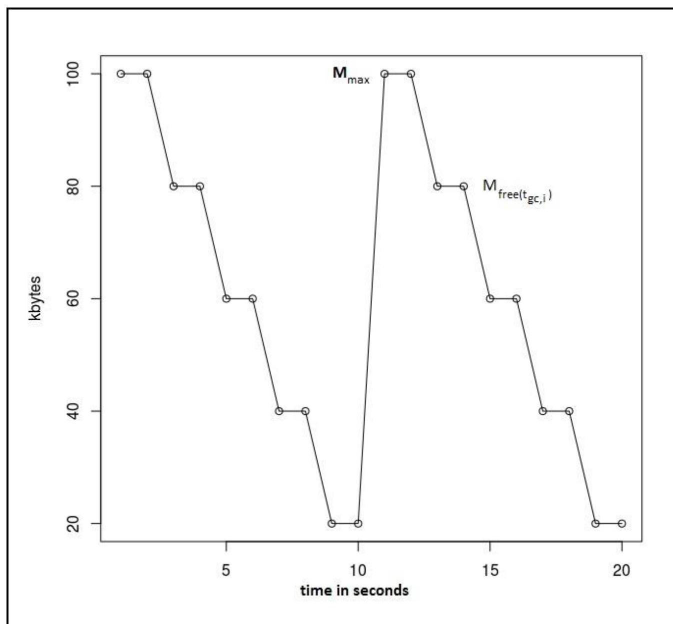

Figure 2. Amount of free memory.

Some of the cloud providers use intrusion detection systems, and to avoid having a client black listed when the server is unavailable, a back-off mechanism is included in the client. This is implemented so that if there are more than a fixed number of non-finished requests, the client waits until the server responds or the network connection times out.

III. DEDICATED CAPACITY

Each server allocates an amount of memory to process the request and in this context the amount is fixed and only dependent on the server side implementation. In this experiment the amount of free memory is logged by the client each time a server is invoked, typically resulting in steps of approximately 10k bytes as shown in Figure 2. In an idealized instance the amount of free memory would then produce a downward-stepping shaped curve until the garbage collector runs and restores the curve to the “max” value, or what memory it is able to free up, as shown in Figure 2. This curve can be modeled as follows. Let us assume the garbage collector runs at instances$t_{gc,1}$,…, $t_{gc,i}$,… and let M_{max} be the memory available to the virtual machine immediately after the garbage collection is finished. The free memory available to the virtual machine can then be expressed as

$$M_{free(t_{gc,i})} = M_{max} \quad \text{and} \tag{1}$$

$$M_{free}(t_{gc,i}) = M_{max} - m \int_{t_{gc,i}}^{t} p(t)dt, t_{gc,1} < t \leq t_{gc,i+1} \tag{2}$$

Where p(t) is the instant load on the virtual machine at time t. For equidistant and constant invocations, the result is an almost linear behavior as shown in Figure 2.

In a server that is little used, the period between garbage collections is long compared to the invocation time, and when plotted on a curve it will show a downward stepping function until the minimum memory point is reached and the garbage collector recovers the memory at which point it starts at the “M_{max}” point again. If one removes the garbage collection steps and only looks at the slope of the steps there is a correlation between how often the server is invoked and how much memory is used for each invocation. This correlation is then proportional to the load that the server process processes. If the load is constant then the slope is only dependent on the invocation frequency and if the slope is constant this implies that there are no others invoking the server in that same time period. Assuming that there is little amount of processing in the request, the requests will be memory intensive and show the amount of available memory at any time.

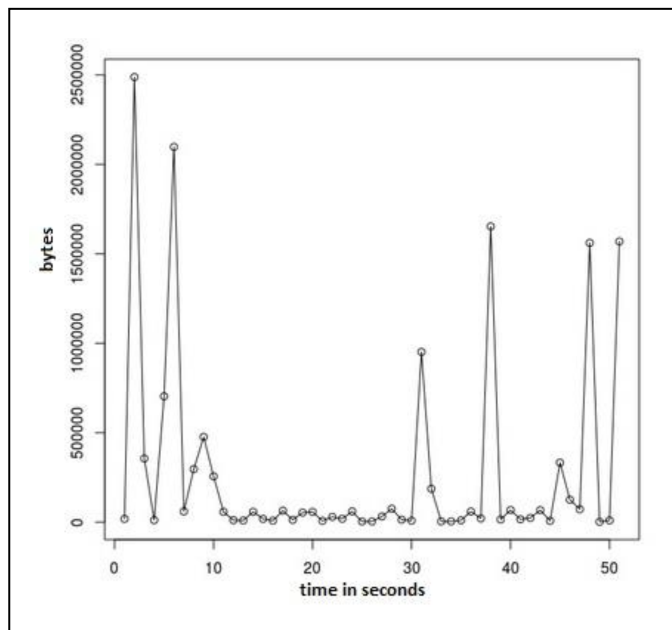

Figure 3. Distorted delta free memory

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-301-8

The time between when the garbage collector is run then depends on the amount of initial memory and the load on the server. The higher the load on the server, the more memory is allocated, and the more often the garbage collector runs [8]. Also if there are other users of the same application server instance the relatively modest usage of the application server is then cluttered by other invocations and the smooth downward-stepping pattern is not observed. Figure 3 shows a delta free memory plot that is showing distorted available free memory.

If one plots the memory usage over a long period, as in Figure 4, one gets a macro view of the data, where the amount of free memory shows up as a fat black line on the graph, this is due to the number of samples, but if the time axis is expanded, the figure would show a pattern as in Figure 2. In Figure 4, the invocation times are shown in green, however, the details get lost in the macro view except in the cases where the server does not respond and the free memory sample is unavailable, and hence, is zero.

The minimum invocation time illustrates the physical transport times and the clustering of these indicate that the route the requests took was the same over longer periods of time which establishes the pattern seen in the Figure 4.

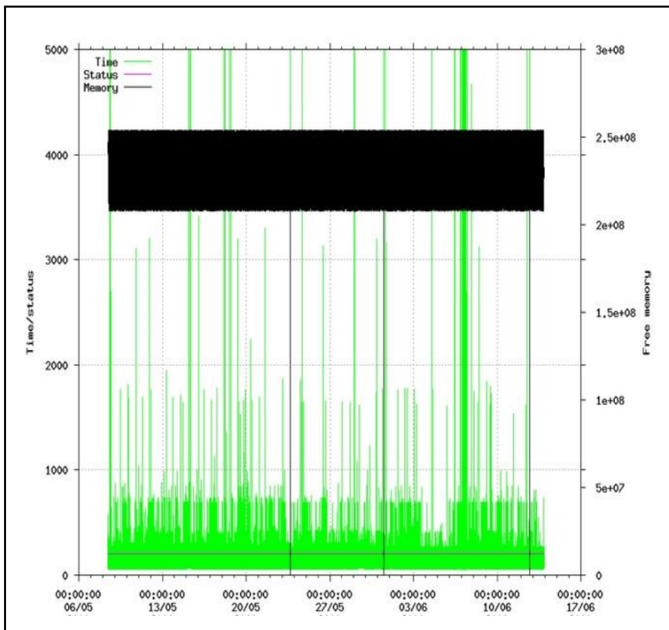

Figure 4. Macro view of the collected data from a Amazon EC2 instance.

By sorting the green invocation times in descending order and plotting them according to their value as shown in Figure 5, the different invocation times appear in better detail and the different “plateaus” seen in Figure 4 are recognized. Note the logarithmic scales. Where the red horizontal and vertical lines cross, the lower right quadrant contains the “successful” invocations meeting the requirement of a response within 5 seconds.

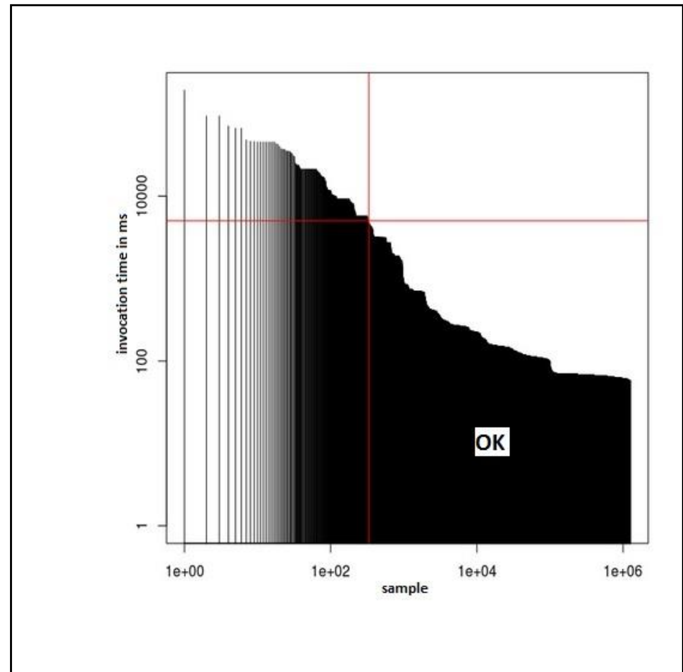

Figure 5. Sorted invocation times from a Amazon EC2 instance.

IV. INVOCATION TIME

When a server is invoked the invocation consists of multiple parts:

1. Connection set-up time
2. Request transport time
3. Request acknowledge time
4. Processing time
5. Response transport time
6. Response acknowledge time

In this experiment the client is the same running instance sending requests in parallel to all servers. The amount of data in the request and the response is small enough to fit into a single transport unit, so there is no consideration of reassembly of transport packets in this experiment. To avoid blacklisting by intrusion detection systems, a back-off mechanism was implemented, so that in the cases where a server ceases to respond, no more than 5 new requests were issued before responses started returning, hence, effectively changing the sampling interval until the server starts responding again.

Note that if the client in question was a person interacting with a graphical user interface and consuming a service and the invocation takes more than 5 seconds their interpretation of the situation would be that something may have gone wrong.

The invocation times were measured with the internal system clock with millisecond accuracy. The invocation

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-301-8

times are typically one to a few hundred milliseconds range as shown in Figure 5.

V. AVAILABILITY STATES

The invocations are logged on the client side and they are categorized using the following states depending on their outcome.

1. Invocation succeeded (***Ok***).
2. Invocation succeeded, but the invocation took more than 5 seconds (***Slow***).
3. Invocation failed, and connection was established, but a timeout occurred when reading or writing the data to the client (***Time-out***).
4. Invocation failed due to lack of connectivity, no route to server was available (***No conn.***).
5. Invocation of server was not started since there are currently more than five outstanding unanswered requests in a row (***Un-answer.***).
6. Skipped requests due to errors in a row (***Skip***).

In order to avoid the black-listing of the client, the client stops sending new requests when the server does not respond in a timely manner. These impacts the availability in state 5 by staying in that state until the issued request receives a response or the network connection times out. A consequence of this is that the server or network might have recovered earlier than the built-in retry times at the TCP transport layer. The results of the invocation per server are summarized in Table I.

TABLE I. STATES OBSERVED BY INVOCATIONS

Servers	Availability states					
	1. Ok	***2. Slow***	***3. Tim-out***	***4. No Conn.***	***5. Un-answer.***	***6. Skip***
Amazon EC2	1259371	326	7	10	170	89
Google App. Engine	1258780	1167	471	36	761	3319
GetNet	1258890	2288	380	17	651	5320
Reference system	1259191	320	25	17	350	901

VI. OPERATIONS

The client was deployed in the NTNU network and was run for a period of over a month. Initially some adjustments had to be made on the client to avoid getting blacklisted on intrusion detection systems if one issued request towards a server that was taken down for maintenance. All but one of the providers provided continuous service, while one took a restart every night in order to restore the system to a known state, also called software rejuvenation [5, 9]. This resulted in a down-period, at a fixed time every night (2AM), however, this is not mentioned anywhere in the terms or conditions for their site. The biggest players announce their target figures for the availability of services and instances. In the case of not meeting their targets, customers may get credits for future free usage. Amazon states that they will use commercially reasonable efforts to make Amazon EC2 available with an "Annual Uptime Percentage" of at least 99.95% during the Service Year. "Annual Uptime Percentage" is calculated by subtracting from 100% the percentage of 5 minute periods during the Service Year in which Amazon EC2 was in the state "Region Unavailable". The Google Apps Covered Services web interface will be operational and available to Customer at least 99.9% of the time in any calendar month. Since the experiment is run over a one month time the results show that they are indeed close to their targets, where Amazon EC2 availability is conservative, however, it is difficult to predict how much the results would differ over a longer period of time.

VII. CONCLUDING REMARKS

As this simple experiment indicates, the cloud providers may provide application servers with similar or better availability than what one can obtain with a traditional non-redundant approach using standard of the shelf hardware and software. Using the results for "Ok" state in Table I, as available, the availability is calculated and the results are shown in Table II.

TABLE II. AVAILABILITY SUMMARIZED

Servers	Monthly values		
	Announced Availability	***Observed Availability***	***Accumulated obs. downtime in minutes***
Amazon EC2	0.9995	0.999522	25
Google App. Eng.	0.999	0.995432	239
GetNet	No info	0.993128	360[a]
Reference system	No info	0.998719	67

a. Nightly restarts.

The reference system in this experiment is a standard PC running Ubuntu operating system [12] and connected to the NTNU campus network.

In summary the big players announce the same numbers for availability as obtained in this experiment.

Concerning how well an instance in the "cloud" is really isolated from other instances, if no prior information exists, is possibly by the limited measurements, as described in this paper. One can observe if the instance is alone or disturbed by other usage from that provider. By comparing the macro results with the micro results it is then possible to assess the offer at hand and make qualified choices regarding the cloud providers in question.

By looking at details of invocation times and amount of free memory at both macro and micro levels, different types of information emerges, to support decisions on scaling and availability.

VIII. Further work

Given the dynamic nature of cloud computing and the possibility to both scale up and scale down by requesting more or bigger instances from the cloud providers, finding a simple means to detect when this should be done is of economic interest for the users, however, starting new instances of servers requires some startup time, and finding good predictors on when new instances are required, or when redundant instances can be stopped and shut down, are still an issue for further work.

Acknowledgment

I would like to thank Professor Rolv Bræk and Professor Bjarne E. Helvik at Department of Telecommunication at NTNU, for their advice and guidance in my research work, and my wife for proof reading my papers.

References

[1] S. Jakobsson “Timing Failures Caused by Resource Starvation in Virtual Machines ”, DEPEND 2011, ISBN: 978-1-61208-149-6

[2] Ryan K.L. Ko, Stephen S.G. Lee and Veerappa Rajan, “Understanding cloud failures”, Spectrum, IEEE , vol.49, no.12, pp.84,84, December 2012 doi: 10.1109/MSPEC.2012.6361788

[3] https://code.google.com/status/appengine/ (last seen June 2013)

[4] http://aws.amazon.com/ec2/ (last seen June 2013)

[5] W. G. Bouricius, W. C. Carter and P. R. Schneider, “Reliability Modelling Techniques for Self-Repairing Computer Systems” Proceedings 24th National Conference ACM, 1969.

[6] A. Avižienis , J. Laprie, B. Randell, and C. Landwehr, “Basic Concepts and Taxonomy of Dependable and Secure Computing,” IEEE Trans. Dependable and Secure Computing, vol. 1,no. 1,pp. 11-33, Jan.-Mar. 2004.

[7] J. Engel: “Programming for the Java Virtual Machine”, Addison-Wesley, 1999. ISBN 0-201-30972-6

[8] R. Jones, “The Garbage Collection Page”, http://www.cs.kent.ac.uk/people/staff/rej/gc.html (last seen June 2013)

[9] Huang, Yennun, et al. "Software rejuvenation: Analysis, module and applications." Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers., Twenty-Fifth International Symposium on. IEEE, 1995.

[10] S. Pertet and P. Narasimhan, “Causes of Failure in Web Applications (CMU-PDL-05-109)”. Parallel Data Laboratory. Paper 48. http://repository.cmu.edu./pdl/48 (last seen June 2013)

[11] http://en.wikipedia.org/wiki/Temporal_isolation_among_virtual_machines (last seen June 2013

[12] http://www.ubuntu.com (last seen June 2013)

[13] S. Jakobsson, “A Token Based Approach Detecting Downtime in Distributed Application Servers or Network Elements”, Networked Services and Applications - Engineering, Control and Management, 16th EUNICE/IFIP WG 6.6 Workshop, EUNICE 2010, Trondheim, Norway, June 28-30, 2010. ISBN 978-3-642-13970-3 Proceedings, pp. 209-216