Timing Failures Caused by Resource Starvation in Virtual Machines

Sune Jakobsson
NTNU, ITEM
Trondheim, Norway
e-mail: sune.jakobsson@telenor.com

***Abstract*—This paper discusses cascading effects of resource starvation in virtual machines, and how that affects end-user experiences in certain cases. The paper presents the occurring issues on an N-tier server system, and the way the starvation causes unexpected delays in a service for an end-user. The initial observations were on unexpected communication delays, and by using a simplified test system, this behaviour can be confirmed. The delays can traced back to memory management in the individual servers, causing the timing failures.**

Keywords- Java virtual machines; garbage collection; application servers; resource starvation.

I. INTRODUCTION

This paper addresses cascading effects of resource starvation, in particular where a server side system is built using multiple tiers, and they call each other in a serial fashion to a depth of N. This effect is observed by end-users intermittently, where their service fails once, and by reloading their browser or application the service is restored. The definition of timing failures is from [3] and is defined as: “The time of arrival or the duration of the information delivered at the service interface (i.e., the timing of service delivery) deviates from implementing the system function.” A Cascading Effect is an unforeseen chain of events due to an act affecting a system [9].

A typical service would consist of a client running in a terminal, accessing a server frontend exposed on the internet. The underlying system is often a multi-tier system consisting of one or more application servers and one or more databases. The frontend, does load balancing, and then passes the request to a HTTP server, which in turn forwards the request to a Servlet container. The Servlet processes the requests, and in turn will contact other servers on the Internet, databases, etc. Once all the information is returned the Servlet builds the response page to the requesting user, and the information is returned.

Each server uses a dynamic amount of memory for their task, and with modern programming languages the memory is allocated when needed and freed when the virtual machine is running low on the free memory pool, or is idle and decides to clean up its memory pool [6]. This process is referred to as garbage collection, and there are many strategies for this mechanism [7]. If one observes the amount of free memory on a virtual machine over time, the waveform is an inverse saw tooth form with a maximum value matching the total amount of free memory, and the minimum values when the garbage collection mechanism is run.

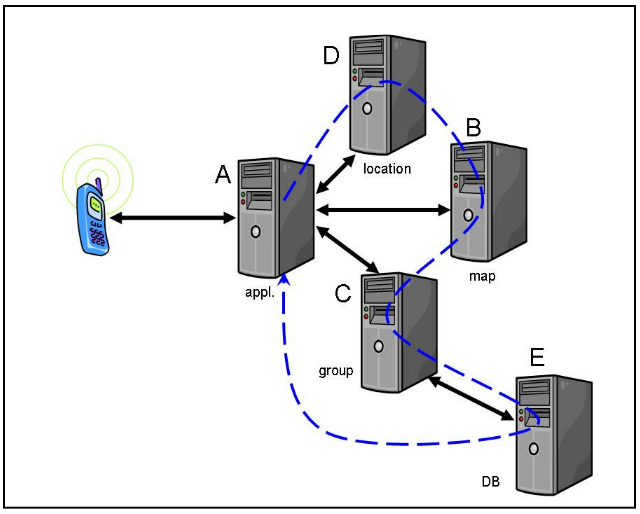

Figure 1. A server system.

Fig. 1 shows how a mobile client interacts with an application server, marked “A”, and how this server in turn interacts with the other servers, the blue dotted line is the path used when checking the availability. The figure also gives an indication on how the servers are deployed, but they might belong to different administrative domains on different networks. Fig. 2 is a sequence diagram, showing how the HTTP invocations in the test system are chained together, when they are called form each server to the next server, starting from server A, until they reach the depth of N servers, where the result is returned. It is assumed that the servers are instances of application servers like Tomcat [5].

II. MEMORY ALLOCATION

Applications need memory for their task and the communication requires buffers to store data. When new objects are allocated they are taken from the memory pool. When the available memory runs low, a garbage collector inspects and frees objects that are no longer in use, and if one observes the available free memory on a typical virtual machine this shows an inverse staircase pattern. Fig. 3 shows a set of available free-memory patterns, showing how they are allocated and garbage collected over time. The amount of free memory available is collected from the JVM, using system calls on each application server. The figures are normalized, so that they can be compared, and the x-axis shows a cycle time of approximately 9 minutes for the servers in blue and brown, whereas one of the servers shown in purple has a cycle time of 23 minutes. The frequencies of the cycles depend on the load of each individual virtual machine. The load impacts the memory usage, and when observing the graphs they show a frequency modulated inverse saw tooth shape.

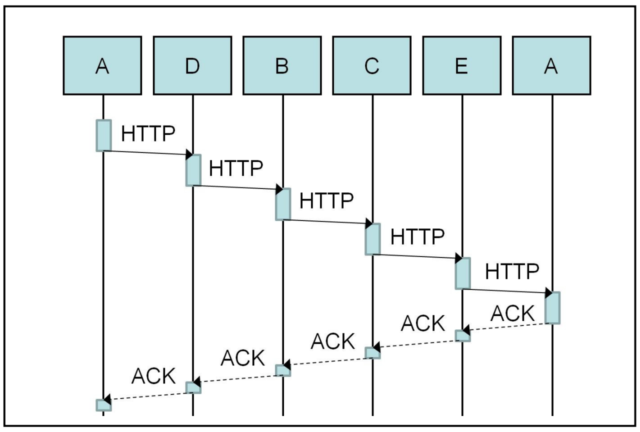

Figure 2. Cascaded HTTP invocations.

From a dependability point of view, the interesting point in time is when the garbage collection is running and hence the requests for free memory are stalled. Since the frequency this occurs at does not happened at regular intervals, and one single run of garbage collection does not impact the overall service execution time. The cascading effect occurs when multiple garbage collectors run right after each other effectively stalling the ability to reserve resources in the virtual machines involved. This condition is a late timing failure as shown in Fig. 8 in [3]. For each virtual machine the probability that the garbage collector runs is p_x. If we number the probabilities p_1 to p_n the interesting scenario occurs when multiple garbage collectors run after each other in a close sequence, effectively stalling the response back to the end-user. For all garbage collectors to run the probability is the product of the individual probability, and for all but one, the probability is sum of the individual probabilities but one, and so on. Given that the number N is small, one can construct a formula and find the N with the biggest likelihood for performance failure.

III. A TEST SYSTEM

To model a real service that would exercise as many parts as possible of an N-tier system, a simple test system has been programmed, which uses the involved virtual machines, and invokes each other into N levels with using a fixed amount of memory in each level. This test system therefore mimics the behaviour of a hypothetical service including a number of servers interacting, communicating over the Internet to provide a composite service. The core issue of application availability is trying to establish a method that ensures that the individual nodes are able to communicate, and that their application servers are functional and available, when they interact as shown in Fig. 2. A typical service would run in an application, and retrieve miscellaneous data from other servers connected to the internet, like group information, location, and maps.

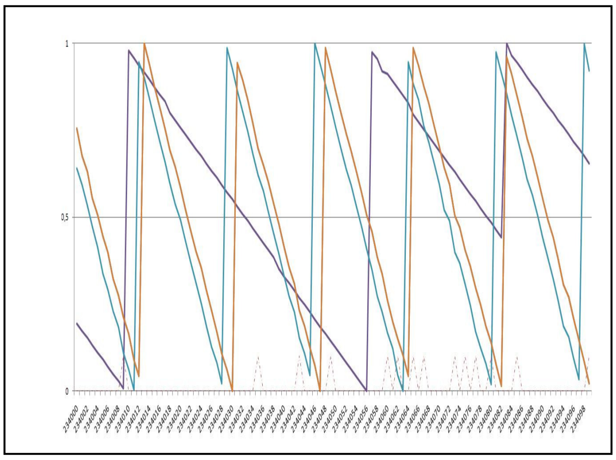

Figure 3. Plotted set of free memory samples (normalized).

Each node communicates with the next node defined by a data object consisting of a fixed list of URL's that is passed between all servers. As part of the local logging process of each node they also collect the amount of free memory available in the virtual machine, and the data object also contains timestamps for later correlation of the results. The list is continually passed around at fixed intervals (30 seconds), and when there are delays or processing issues this impacts the amount of free available memory. This logged data on each node can then be post-analyzed and the real cause can be determined, for the failure of the hypothetical service. The interesting part here is the strong correlation on how many buffers are occupied due to delays or errors in the transmission between the participating nodes. When there are HTTP messages that are not acknowledged, they use up the common memory of the system, and this can be measured with the available free memory. The amount of free memory can be obtained directly from a Java virtual machine if the application is implemented in the Java programming language or from the operations system when other programming languages are used. The elegance of this approach is that the measurements are non-intrusive to the application, and eliminates other hooks into the communication channel or their respective drivers.

IV. OPERATION

Each node, when invoked, obtains a time reference and the amount of free memory, and when the communication is done with the next node, these values are written to the standard output file of the application server. At the sending end there is a standalone program issuing the requests at fixed intervals as shown in Fig. 1, from server marked “A”.

If the transmitted URL list makes it all the way to the destination the sender receives an acknowledgment in the form of HTTP status (200) OK, but there are plenty of observations where the list is delayed or its acknowledgment is delayed, but not lost. If the list is lost that is an obvious case, and also indicated in the HTTP status from the underlying TCP implementation in the operating system, but there is a group of cases where the list is not lost and makes it all the way to the last node, and the acknowledgment is returned, and all seems fine, yet the time seen from the initiating node is unacceptable long, and we have a case of timing failure. The term “unacceptable long” varies from case to case, but if a human does the interaction a delay for a second on a response from a service might be unacceptable.

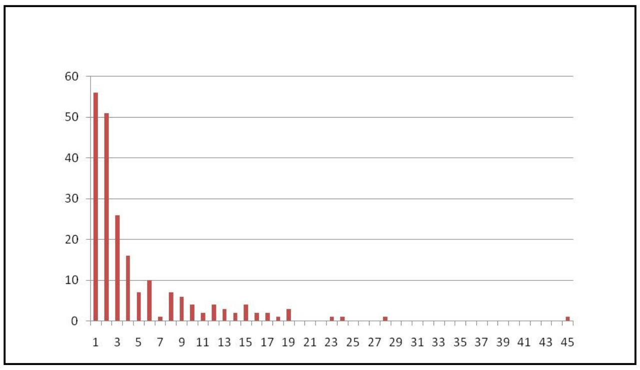

Figure 4. Time between timing failures.

By post-processing the log files from each node, more details about the cause can be obtained. The graph in Fig. 4 shows the time between failures for a period of a month, where only a few other errors occurs, where the data object fails on its route, and in this period only timing failures occurred. The steps on the x-axis are 3000 seconds each. The number samples with timing failures is 212 out of totally 80624 data objects dispatched. In this period there are also 15 cases of halt failures in the data set. The shape of the graph indicates that it resembles a Poisson distribution, with a λ of approximately 0.955. This fits well with the assumption that the events occur continuously and independently at a constant average rate.

V. State of the Art

In closed and well monitored systems, with probes and other means of surveillance there are many commercial available solutions to detect timing failures and other failures and faults. However when it comes to distributed systems across different domains where there is no common administration, there is little material available. Several authors have studied causes of catastrophic failures in Web Applications [8] and the failures impacts on operation and how the failure has impacted the companies’ respective brand. Porter defines a system called X-Trace [1], to collect trace data to figure out what went wrong in an Internet scale system. This is done by adding extensions to HTTP headers in the requests, in order to be able to trace or locate them afterwards. His approach has some scaling issues, and also requires insertion of monitoring nodes or additional SW on the servers.

In the approach we propose, one would add the proposed minimalistic test application on each node one has control over and call the other nodes with some dummy data, in order to decide if the communication and the application servers are indeed available.

VI. Conclusion

This paper showed the cascading effect of the individual memory allocation processes, and how they do affect the performance and availability for a service using multiple serves loosely connected over the Internet. The data collected also supports the assumption that the occurrence of timing failures occur continuously and independently at a constant rate. This paper outlines one of the issues observed on practical data collected in my research work, and will be further validated and modelled in my thesis work.

Acknowledgment

I would like to thank Professors Rolv Bræk and Bjarne Helvik at Department of Telecommunication at NTNU, for their advice and guidance in my research work.

References

[1] G. Porter, “Improving Distributed Application Reliability with End-to-End Datapath Tracing”, PhD at Electrical Engineering and Computer Sciences, University of California at Berkeley Technical Report No. UCB/EECS-2008-68 http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-68.html (last seen Jul. 2011)

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

[3] 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.

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

[5] Tomcat application server. http://tomcat.apache.org/ (last seen Jul. 2011)

[6] Java virtual machine. http://java.sun.com/javase/ (last seen Jul. 2011)

[7] R. Jones, “The Garbage Collection Page”, http://www.cs.kent.ac.uk/people/staff/rej/gc.html (last seen Jul. 2011)

[8] 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 2011)

[9] Cascading effect: http://en.wikipedia.org/wiki/Cascade_effect (last seen 2011)

Article in conference proceedings:

[10] 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