Measuring and Modelling the Effect of Application Server Tuning Parameters on Performance

Gábor Imre, Ágnes Bogárdi-Mészöly, Hassan Charaf

Department of Automation and Applied Informatics, Budapest University of Technology and Economics
Goldmann György tér 3, H-1111 Budapest, Hungary
gabor@aut.bme.hu, agi@aut.bme.hu, hassan@aut.bme.hu

Abstract: Several factors affect the performance of a web application. In this paper, the effects of two configurable settings of the J2EE application server are discussed: the maximum size of the thread pool and the maximum size of the connection queue. The response time, throughput and error rate of a web application are measured under different client load, while changing these settings. The results are primarily analyzed in a qualitative manner which is followed by quantitative reasoning based on a queueing model of the system. Our experiments show that both tuning parameters have a considerable impact on the performance metrics, and both of them should be taken into account when constructing a performance model of a web application.

Keywords: Performance evaluation, Web technologies, thread pool, connection queue

1 Introduction

At the early stage of the Internet, the Web was mainly used to display static content. As the Web became more and more widespread, several companies realized that web applications that are able to provide dynamic content can offer a strong support for their activities. Managing business processes, the improper performance of a web application can cause serious financial loss to a company. The performance-related requirements of an Internet application are often recorded in a Service Level Agreement (SLA). SLAs can specify an upper limit for the average response time, a lower limit for availability, while the application guarantees a certain throughput level.

These performance metrics depend on several factors, such as hardware, software, network, and client workload. This paper focuses on the settings of the application server software that serves the HTTP requests of the browsers. More precisely, the performance of a test web application is measured under different client load with different values of two parameters of the application server. These tuning parameters are the maximum size of the thread pool, and the maximum size of the HTTP connection queue. To understand the meaning of these parameters, consider Figure 1.

In the application server, accepted HTTP connections are placed into a connection queue. The size of the connection queue is limited by an adjustable parameter of the given application server. When this limit is reached, it is denied to serve the request. The threads in the thread pool take connections from the queue and serve the requests. The server can decide to create more threads (i. e. increase the size of the thread pool), but cannot exceed a certain configurable maximum. When the maximum thread pool size is reached, however, the requests are *not* dropped, as long as they find free space in the connection queue. The policy for adding new threads is typically based on the state of the connection queue. For more information refer to [1].

The limit for the size of the thread pool is necessary for controlling the memory usage of the applications. When the memory requirements of serving a request is known, the maximum memory usage of a web application can be set to fit in the physical memory in order to prevent thrashing because of the size limit of the thread pool. It is important to mention that not all the application servers allow manipulating both of the settings.

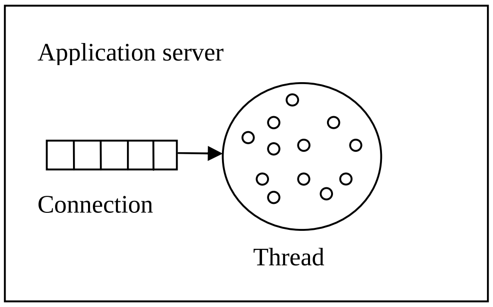

Figure 1
The connection queue and the thread pool

The rest of this paper is organized as follows. Section 2 reviews the related work. Section 3 contains the process of the performance measurement and discusses the results, including our proposed performance model. Finally, we draw conclusions.

2 Related Work

Several papers and research projects are engaged in studying how the various configurable parameters affect the performance of web applications. Two approaches for evaluating the impact of these parameters are presented in [2] and in [3]. They use statistical methods, hypothesis testing in order to retrieve the software parameters that influence the performance. [3] investigates the average response time only, while [2] also takes the throughput and the probability of rejecting a request into consideration.

Some industry-standard benchmarks address standardizing the evaluation of application servers. In the field of Java 2 Enterprise Edition, ECPerf [4] and its successor, SPECjAppServer [5] are the most popular benchmarks. TPC-W [6] is a benchmark that is not tied to any particular implementation technology.

Performance measurements can serve as the basis for performance modeling and prediction. In the past few years several techniques and methods have been proposed to address this goal. A group of them are based on queueing networks [7], or extended or layered versions of queueing networks. These methods establish a queueing network model of the system. By solving this model with analytical methods or simulation, the prediction of performance metrics is possible. Some of the proposed methods generate a queueing network model of the system based on its UML model [8][9]. In [10], a queueing model for multi-tier internet applications is presented, where queues represent different tiers of the application. The model faithfully captures several aspects of web applications, like caching and concurrency limits at the tiers. The maximum size of the connection queue, as presented earlier, can be considered as a concurrency limit of the web tier in this model, but it cannot handle the maximum size of the thread pool.

Another group of performance modeling techniques uses Petri-nets or generalized stochastic Petri nets, such as [11]. Petri nets can represent blocking and synchronization aspects much more than queueing networks, which are more suitable for modeling resource contention and scheduling strategies. A powerful combination of the queueing network and the Petri net formalism is presented in [12]. Using queueing Petri nets, the authors successfully model the performance of a web application, considering the maximum size of thread pools. Their model, however, does not take the maximum size of the connection queue into account.

This paper shows that the limits configured both for the connection queue and the thread pool have a considerable effect on the performance.

3 Contributions

3.1 The Performance Measurement

The test web application is intentionally designed to be very simple, such that no factors other than the settings of the application server can influence the performance. It can serve one type of HTTP requests, and is implemented with a Java servlet. On processing a request, it performs double precision computations, and periodically inserts *sleep()* calls. This emulates typical web applications, which use the processor of the machine that hosts the web container and calls services on other machines (e.g. a database server) in a synchronous way, which blocks the caller thread. The number of computations and the total sleep time can be defined as parameters of the request. After processing a request, the web application generates a small html file (of approximately 10 kilobytes) as a response.

The application server (Sun Java System Application Server Enterprise Edition 8.1) runs on a PC with Windows XP, and a 3 GHz Pentium 4 HyperThreading processor and 1 GB memory. The emulation of the browsing clients is performed by an open source load tester, JMeter, which runs on another PC, with similar hardware. The two machines are connected via a 100 Mbit/s LAN.

Each test takes 20 minutes, during which the virtual clients send their requests to the server. Each virtual client inserts an exponentially distributed thinking time between its requests with mean 4 seconds. The virtual clients are started gradually, in a 40 seconds interval. The system reaches a steady state after 2 minutes in terms of average response time and throughput. Between two test runs, the application server is restarted because the new settings have to be reloaded. The values of the two investigated tuning parameters, and the numbers of the emulated clients during the individual measurements are summarized in Table 1. With these values, we expect that both the saturation of the thread pool and of the connection queue can be observed.

Maximum size of the thread pool	30, 100
Maximum size of the connection queue	50, 200
Number of emulated clients	10, 20, 30, 40, 50, 60, 70, 80, 90, 100

Table 1
Number of clients and values of the tuning parameters during the measurements

3.2 The Results

During each measurement, the following metrics were registered:

- The average response time of the requests, measured at the clients. This includes the network time of sending the request and receiving the response, but this time it is negligible, since these files are small, moreover, the client and the server are on the same local network. This is verified by comparing the client side response time to the response time measured at the server side. The two values differ less than 5 percent, thus, the time of the server side processing dominates the client side response time in our experiments.
- The throughput of the system, which is the number of served requests in a second.
- The rate of requests that are dropped by the server.
- The processor and memory usage of the server and the machine running the clients. We found that the memory usage does not reach the 80% of the available physical memory on either of the machines. The processor usage on the client machine never reaches 60 percent, therefore, it cannot be a bottleneck in our measurements.

Hereafter the measured response times are presented first. The JMeter tool can log the average response time of all requests (R_{all}), and the average response of the unsuccessful requests (R_{error}). The average response time of the successful requests (R_{succ}) can be derived as follows. Let *e* denote the rate of unsuccessful requests, *n* the total number of requests during a measurement. Then equation (1) holds.

$$n * R_{all} = n * e * R_{error} + n * (1 - e) * R_{succ} \qquad (1)$$

This suggests that

$$R_{succ} = \frac{R_{all} - e * R_{error}}{1 - e} \qquad (2)$$

The values computed with Equation (2) are depicted in Figure 2. As one can see from the two overlapping lines, the maximum size of the thread pool does not influence the response time when 200 connections are allowed. With maximum 50 connections, however, the maximum size of the thread pool has a significant effect on the response time: the configuration with 100 threads performs better than the one with 30 threads. Our other observation is that for more than 60 clients, if we limit the connection queue to 50, it is possible to achieve better response time than that with 200 connections allowed. The reason of this phenomenon can be explained considering the rate of the unsuccessful requests in Figure 3.

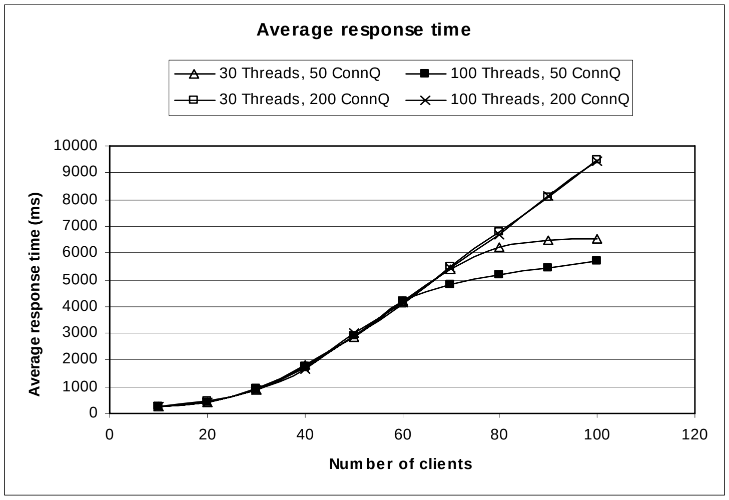

Figure 2
The average response time in the various configurations

It can be seen that with maximum connection queue size of 200, all the requests are successfully served. This is the expected behavior, since all the clients wait for the response to their current request before sending a new one. In this way, it is impossible not to serve a request due to a full connection queue, since the number of clients never exceeds the maximum connection queue size of 200. With the connection queue sized at 50, however, dropping requests is expected, when the number of clients exceeds 50. Figure 3 confirms this expectation: the two configurations with the maximum connection queue of 50 have a non-zero rate of unserved requests. Because a considerable part of the requests does not consume the resources of the server, a better performance can be guaranteed for the rest of the requests. It is notable that the error rate is higher for 100 threads than for 30 threads, supposedly due to the increased contention between the threads.

Figure 4 illustrates the throughput measured during the experiments. The curves are virtually identical in the different configurations and they saturate around 7 requests/sec. This means, that with 50 connections, although dropping a notable part of the requests (see Figure 3), the improvement in the average response time of the served requests reaches a degree that compensates the negative effect of the increased error rate.

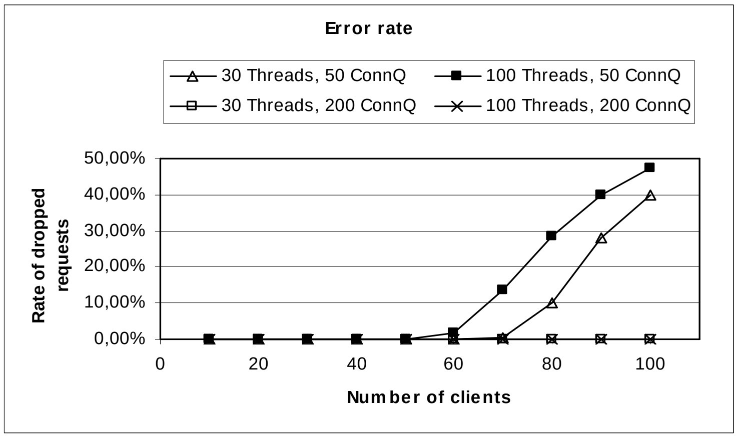

Figure 3
The rate of unsuccessful requests in the various configurations

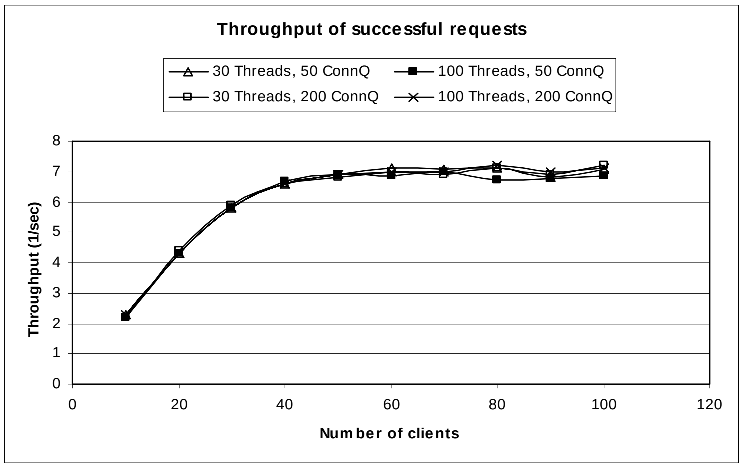

Figure 4
Throughput of successful requests in the various configurations

3.3 Performance Reasoning

To meet the performance requirements of Service Level Agreements, the performance metrics of a system under given conditions can be obtained in two ways. The first method is load testing which was presented in the previous sections. The other method is to establish an analytical performance model of the system which can provide an estimation of the relevant performance metrics.

In this section we apply a queueing model of the performance and validate it against the results of the load testing. To specify a queueing system, it is necessary to identify following parameters. The distribution of the *interarrival time* (the think time introduced by the virtual clients) is exponential with mean 4 seconds (*Z*). The exact distribution of the *service time* is unknown, but to keep our model simple, we will assume an exponential distribution with mean 0.14 seconds (*S*) based on measurements using one virtual client. The service rate μ is defined as *1/S*. The *number of servers* is one, the *popoulation size* (*K*) is equal to the number of virtual clients. The *system capacity (B)*, i.e. the maximum number of requests in the system is equal to the maximum size of the connection queue, because a request in the connection queue remains there until the request is served.

Using these parameters, our queueing system can be described using Kendall notation: M/M/1/B/K, where M stands for *memoryless*, a well-known property of the exponential distribution. To solve this system, we should first calculate the probability that there are *k* requests at the server (p_k), using (3) and (4). A derivation of these formulas can be found in [13].

$$p_k = p_0 \frac{K!}{(K-k)!(\mu Z)^k} \qquad k = 1,\ldots,B \tag{3}$$

$$p_0 = \left[\sum_{k=0}^{B} \frac{K!}{(K-k)!(\mu Z)^k} \right]^{-1} \tag{4}$$

Based on these results, all other performance metrics are easy to compute, using some basic results of queueing theory. The requests are dropped, when the system reached its capacity *B*, hence, the error rate $e=p_B$. The throughput of the system (*X*) can be calculated based on the Utilization Law which states *X=U/S*, where *U* is the utilization of the server i.e. the probability that the server is busy, so $U=1-p_0$. And finally, we obtain the average response time from Little’s Law: *R=N/X*, where *N* is the average number of request in the system, that is:

$$N = \sum_{k=0}^{B} k * p_k \tag{5}$$

The performance metrics obtained from this model are compared to the measured values in the figures below. The throughput values obtained from the model are virtually the same for 50 and 200 connection queue size, thus, they are presented together in Figure 5. The model slightly overestimates the measured values, but the fit is sufficiently accurate.

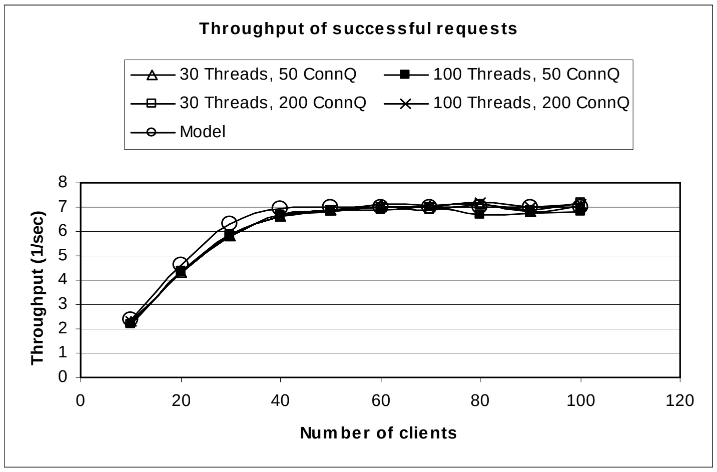

Figure 5
Comparing the model with the measured data – throughput

Since with a maximum connection queue size of 200, the error rate is zero both for the measured configurations and in the model, this case is not depicted. When the connection queue is limited to 50, the model provides values that are between the measured values for maximum 30 and 100 threads (Figure 6). This is due to the fact that our model does not model the thread limits.

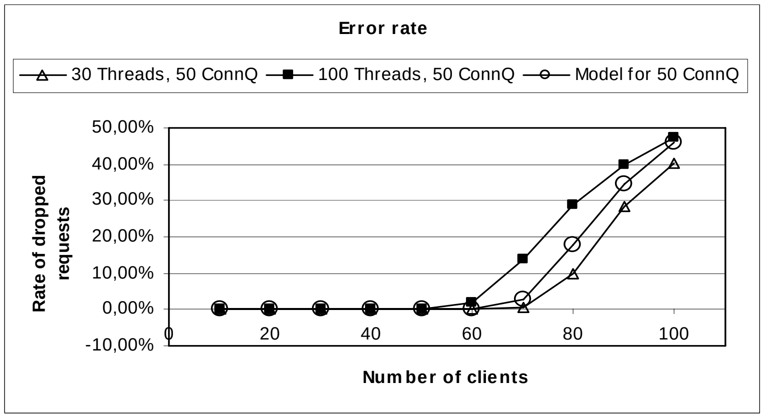

Figure 6
Comparing the model with the measured data – error rate

The curves for the average response times are quite different depending on the connection queue size, and they are considered separately in Figure 7 and 8. In both cases the model overestimates the response time, but the nature of the curve is captured for both cases. With maximum 50 connections, the effect of the thread pool which is not included in our model becomes considerable which explains the larger deflection from the measured values.

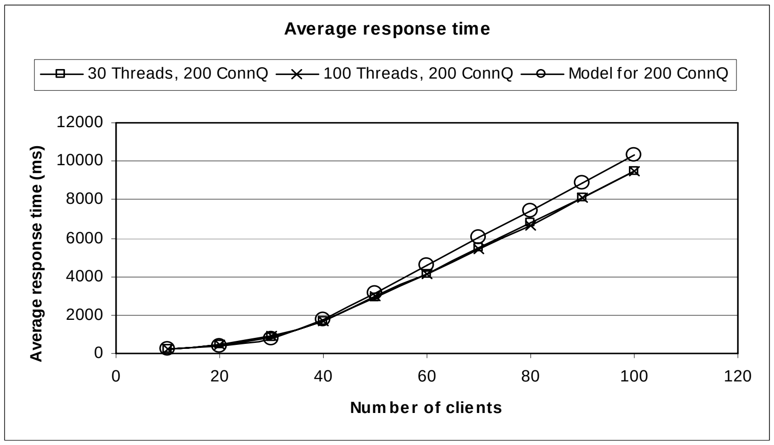

Figure 7

Comparing the model with the measured data – average response time with 200 ConnQ

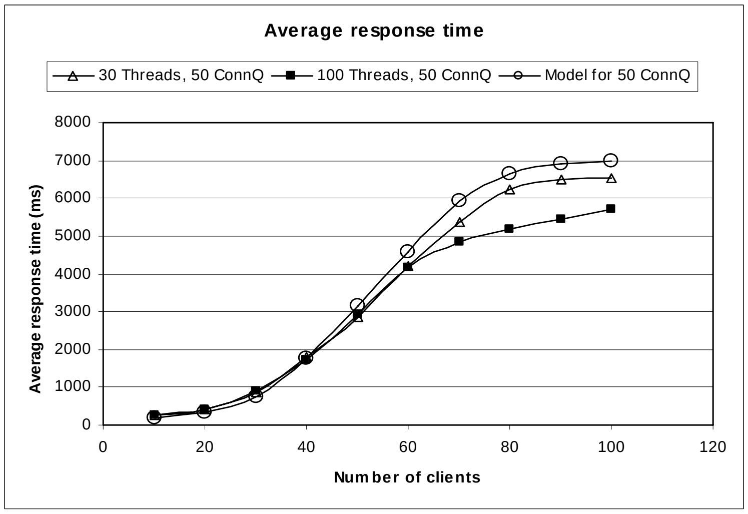

Figure 8

Comparing the model with the measured data – average response time with 50 ConnQ

Conclusions

This paper presented the results of a performance measurement that focused on two settings of the application server. We state that by limiting the maximum size of the connection queue, better response time can be achieved, but with higher error rate, while maintaining the same throughput. We have also shown that increasing the limit of the thread pool size results in an increased error rate and decreased response time when the number of clients exceeds the maximum size of the connection queue. The simple queueing model proposed in this paper is able to quantitatively capture the effect of the maximum size of the connection pool. Extending the model with the thread pool is subject of future work, similarly to testing the relevance of the model with more complicated web applications.

References

[1] Sun Java™ System Application Server Enterprise Edition 8.1 Performance Tuning Guide, Sun Microsystems, Inc., 2005

[2] M. Sopitkamol, D. A. Menascé: A Method for Evaluating the Impact of Software Configuration Parameters on E-Commerce Sites, in Proceedings of ACM 5th International Workshop on Software and Performance, Palma, Illes Balears, Spain, 2005, pp. 53-64

[3] Á. Bogárdi-Mészöly, G. Imre and H. Charaf: Investigating Factors Influencing the Response Time in J2EE Web Applications, WSEAS Transactions on Computers, 4(2), 2005, pp. 179-183

[4] ECPerf project homepage can be found at - http://ecperf.theserverside.com/ecperf/index.jsp

[5] Standard Performance Evaluation Corporation (SPEC), SPECjAppServer website - http://www.spec.org/osg/jAppServer/

[6] Transaction Processing Performance Council, TPC-W - http://www.tpc.org/tpcw/

[7] L. Kleinrock: Queueing systems, Volume 1: Theory (John Wiley and Sons, Inc., 1975)

[8] V. Cortellessa, A. D'Ambrogio and G. Iazeolla: Automatic Derivation of Software Performance Models from CASE Documents, Performance Evaluation, 45(2), 2001, pp. 81-105

[9] V. Cortellessa and R. Mirandola: Deriving Queueing Network Based Performance Model from UML Diagrams, in Proceedings of ACM 2nd International Workshop on Software and Performance, Ottawa, Ontario, Canada, 2000, pp. 58-70

[10] B. Urgaonkar, G. Pacifici, P. Shenoy, M. Spreitzer and A. Tantawi: An Analytical Model for Multi-Tier Internet Services and Its Applications,

ACM SIGMETRICS Performance Evaluation Review, 33(1), 2005, pp. 291-302

[11] S. Bernardi, S. Donatelli and J. Merseguer: From UML Sequence Diagrams and Statecharts to Analysable Petri Net Models, in Proceedings of ACM 3rd International Workshop on Software and Performance, Rome, Italy, 2002, pp. 35-45

[12] S. Kounev, A. Buchmann: Performance Modelling of Distributed E-Business Applications Using Queueing Petri Nets, in Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software, Austin, Texas, 2003, pp. 143-155.

[13] D. A. Menascé, V. A. F. Almeida: Capacity Planning for Web Services, Prentice-Hall, Upper Saddle River, 2002, pp. 325-337