A Computational Complexity-Aware Model for Performance Analysis of Software Servers

Vipul Mathur   Varsha Apte
Department of Computer Science and Engineering
Indian Institute of Technology - Bombay
Powai, Mumbai, Maharashtra 400 076, India
{*vipul,varsha*}*@cse.iitb.ac.in*

Abstract

Queueing models are routinely used to analyze the performance of software systems. However, contrary to common assumptions, the time that a software server takes to complete jobs may depend on the total number of active sessions in the server. In this paper, we present a queueing model that explicitly takes into account the time, taken by algorithms in the server, that varies with the user population. The model analytically predicts response time and "saturation number" of such systems. We validate our model with simulation and further demonstrate its usefulness by suggesting a heuristic technique to "discover" the complexity of algorithms in server software, solely from response time measurement. We applied the discovery technique to a Web-server test-bed, and found that we can identify the asymptotic behavior of processing time as a function of the user population with a fair amount of accuracy. The results show that this promises to be one of the many "black-box analysis" techniques, often found necessary in the real world.

1. Introduction

The phenomenal growth of the Internet and the ubiquitous availability of the Web-browser on users' workstations has made Web-servers the overwhelming choice for the front-end "presentation server" for on-line services. Over the last decade not only were innovative services offered through the Web, but many already existing services became "Web-enabled". Thus a Web-server became their service access point, and a Web-browser, the client at the user end. Everything from telephone directories, e-mail, to airline reservations, shopping, and financial transactions are now done "on the Web".

Since software servers (with Web-servers as a front-end) are now accessed directly by an end-customer, the performance of such servers has assumed increased importance. User-perceived measures such as *response time* must meet customer expectation. Performance analysis techniques of measurement and/or modeling therefore need to be applied to the software systems [1]. Hence queueing models are often used to model software performance [8].

Queueing theory, was however, developed in the context of telecommunications and was later widely employed for modeling data networks. For it to be used in the context of software, however, several challenges had to be addressed. E.g., a major challenge of modeling the interdependencies between software as well as hardware resources has been addressed thoroughly using layered approaches [5, 9].

In this paper, we address a specific behavior of software servers that is different from telecommunication links: the fact that service time (i.e. request processing time) depends on the *algorithms* that are used in the software. Software servers often build data structures, whose size changes with the number of active sessions. As the size of the data structure increases, so does the request processing time. We illustrate this with an example below.

1.1. Session-oriented Web-services

In a typical Web-based *transaction* system, customers visit a website and establish their identity with the system by "logging in" (authentication). After authentication, the customer is able to do transactions in the current session without explicitly authenticating every time. Since HTTP (Hyper Text Transfer Protocol) is inherently a stateless protocol, to enable this, state information is explicitly maintained by the server. Thus a client, after being authenticated, gets a "session identifier" and then passes it to the server with each subsequent request.This enables the server to identify the customer who is making the request and provide results accordingly.

After a client has received a reply to its request, the client processes the reply at its own end; i.e. "thinks". The server maintains the session identifiers and associated data

for the clients even during the thinking phase.This session-information is removed only when a client is logged out explicitly or the session expires. Each time that the server receives a request, the application has to search for the session identifier in the list of users logged into the system. This search may involve a linear ($O(N)$, where N is the number of users logged in) or logarithmic ($O(\log N)$) search through the data structure used to hold session information.

Apart from this search for authentication, a server may have an even larger representation of a "state" for each customer. E.g., there could be a large number of session-related objects that are instantiated on a "login" and are therefore proportional to the number of users who are logged in. Processing a query would then surely involve some computation that grows as the size of such data structures grows.

In short, some part of the computation done in the server is a function of the number of users with active sessions (which is not the same as the number of requests queued at the server). Thus the service time is a function of the *population* of customers in this closed system.

Such a behavior can also be found in Java-based systems where the size of the heap [12] can grow with the number of users, and processing of each request would require accessing the heap several times.

We believe that this form of "load-dependent behavior" of a queue has not been studied in existing literature. Most existing work on load-dependent servers looks at the number of requests at the server and queue, and not the entire population [4, 7, 10]. In this paper, we extend the basic client-server closed queueing network model [6, 11] to one that takes into account the complexity of algorithms (in terms of the population N) that constitute the processing of a request. This "population dependent" model leads to better prediction of response times.

This exercise also brings forth an interesting application of such models–that of "discovering" the complexity of algorithms in a software server. We suggest a heuristic technique to discover the complexity of algorithms in software, whose internals are not known, solely from user-perceived measures such as response time and throughput.

It can be argued that the population-dependent component of request processing time may be insignificant compared to the total processing time. Although this may be true in most cases, the existence of this model and of the discovery technique does help in providing a sanity check for the performance of software servers. Such "black-box" analysis techniques have been proposed and used in the real world before [2]. We envisage the discovery technique to be used in situations where server software is being built for a customer organization, by an outsourced vendor. The discovery technique could then be used in the phase of "acceptance testing" of the software by the customer. If the software has design flaws, or bugs, that make its service time increase abnormally with the number of logged-in users, our technique would "expose" this problem.

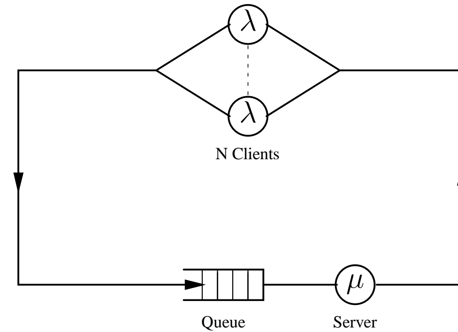

Figure 1. Basic client-server system model

The rest of the paper is as follows: We start by taking a look at the current approach taken for response time estimation based on a basic client-server model. We then extend this model to cases where the mean service time is dependent on the population of the system. This modified model is validated with results from simulations of population dependent service. In the next part of the paper, we develop a heuristic technique to "discover" the service time complexity of algorithms in a software server, purely based on user-perceived measures. This technique is validated–first with results from a simulation of sequential and binary searches, and then by doing measurements on an actual Web-based software system hosted on a typical server. This demonstrates the usefulness of the model and techniques developed in the present work. Some concluding remarks are offered and directions for future work are explored at the end.

2. Current approach

The performance prediction of a Web-based service (or any client-server application) typically involves a mix of measurement and modeling. The steps involved in this approach would normally be:

- Measure system performance (throughput, response time) by generating synthetic load with say N virtual users emulating "sessions".
- Estimate service time by measuring utilization U and throughput T of the server and setting service time τ to $\tau = U/T$.
- Use the well-known basic client-server model [6, 11] (Fig. 1) to predict response times.

Attention is usually not given to the fact that average service time τ could actually vary with changes in N.

This basic model assumes a single server with a FIFO queue and exponentially distributed service times with

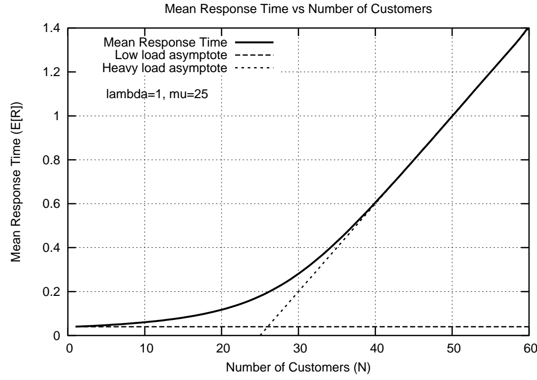

Figure 2. Response time vs. population

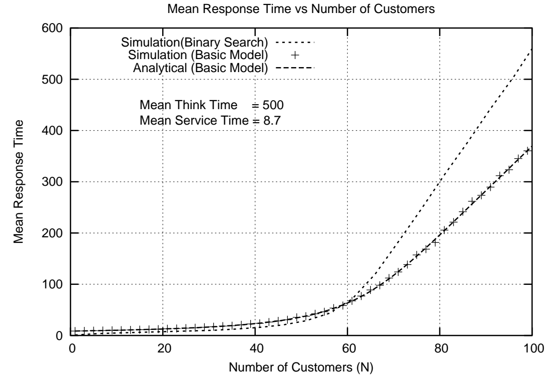

Figure 3. Prediction with current approach

mean $\tau = 1/\mu$. Each client "thinks" for some time before issuing a request at the server. The think times are assumed to be exponentially distributed with mean $1/\lambda$. After completion of service the client goes back into the thinking state. Thus the clients circulate endlessly in the closed queueing network.

For a given N, let $R(N)$ denote the mean response time. Then for this model,

$$R(N) = \frac{N\tau}{U(N)} - \frac{1}{\lambda} \quad (1)$$

where $U(N)$ is the utilization of the server, and can be calculated from the corresponding Markov chain [11].

Fig. 2 shows the graph of response time vs. population N. At light loads ($N \rightarrow 1$) the expected mean response time approaches the service time τ. Also as $N \rightarrow \infty$, we have $U \rightarrow 1$ and $R(N) \rightarrow (N\tau - (1/\lambda))$. Thus the asymptotic behavior of $R(N)$ is linear in N. These light and heavy load asymptotes are depicted in Fig. 2. The point where these two meet is termed the "saturation number" [6, 11].

$$N^* = \frac{\tau + \frac{1}{\lambda}}{\tau} = 1 + \frac{\mu}{\lambda} \quad (2)$$

Fig. 3 shows the results of applying the basic model to a software server which does a binary search, through a list of size equal to the population, for every request that comes in. The analytical results are generated using (1), with τ equal to the time taken by the server when the population is 50. The simulation, on the other hand, explicitly performs a binary search on a data structure whose size is equal to the population. It is easy to see that as number of users increase, the response time predicted by the basic equation is an underestimate of the real response time. (The figure also shows the plot from a simulation of the basic model, which clearly matches the analytical model.)

3. Population dependent model

The basic model assumes that the mean service time is constant. We now model the mean service time as a function of N, the population of the system. Thus $\tau(N)$ is the mean service time when the user population is N.

Since we are still interested in the steady-state measures of the system with a fixed number of users, (1) still holds with the modification of service time τ also reflecting its dependence on N.

$$R(N) = \frac{N\tau(N)}{U(N)} - \frac{1}{\lambda} \quad (3)$$

Let us take a look at the common forms that $\tau(N)$ can take. In the case of a search of the session ID in some data structure as discussed in the introduction, common service time complexities would be $O(N)$, and $O(\log N)$. Other forms like $O(N \log N)$ and $O(N^2)$ can also be present in cases where a sorting operation is needed to serve every request. In most well-designed applications, the use of hashing should make complexity of searches $O(1)$.

Equation (3) shows that the asymptotic behavior of $R(N)$ may no longer be linear. Table 1 shows the asymptotic behavior of $R(N)$ corresponding to service times that are different functions of N.

To predict $R(N)$ accurately, however, the "order of complexity" of the service time is not enough. The value of

$\tau(N)$	$\lim_{U\rightarrow 1} R(N)$
$O(1)$	$O(N)$
$O(N)$	$O(N^2)$
$O(\log N)$	$O(N \ \log N)$

Table 1. Some common forms of $\tau(N)$ and the corresponding forms of $R(N)$

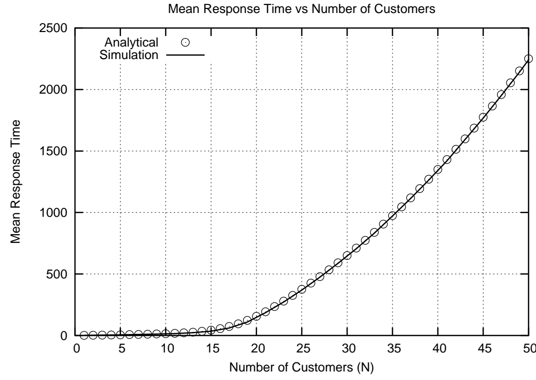

Figure 4. Model validation: $\tau(N) = O(N)$

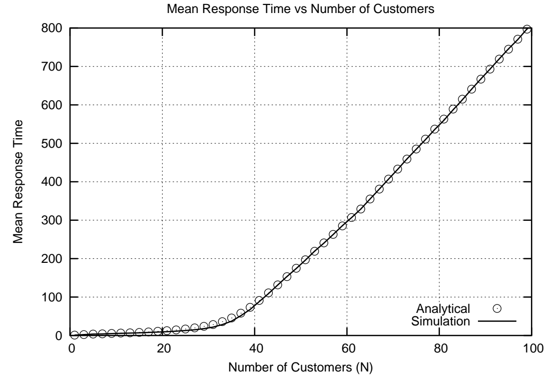

Figure 5. Model validation: $\tau(N) = O(\log N)$

$\tau(N)$, i.e., the mean service time as a function of N must also be known for calculating $R(N)$. We show $\tau(N)$ here for two cases–sequential search and binary search.

Searching consists of a series of successive *compare* and *forward* operations. Let C_C and C_F denote the operational cost of a *compare* and a *forward* respectively. In our model, a search is always successful as it is the same users that keep circulating in the closed queueing network.

Let γ denote the average number of *compares*. This average for a successful sequential search is given by

$$\gamma_{\text{seq}} = (N+1)/2. \tag{4}$$

For a binary search, on a balanced binary tree or a sorted array [3], the average is given by

$$\gamma_{\text{bin}} = (N - (2^{h-1}))h + \sum_{i=2}^{h} 2^{h-i}\,(h-i+1) \tag{5}$$

where $h = \lfloor \log_2 N \rfloor + 1$. For a successful search, in both sequential and binary search methods, the number of *compares* is one more than the number of *forwards*. Thus for both sequential and binary search we have

$$\tau(N) = \gamma C_C + (\gamma - 1)C_F. \tag{6}$$

Using these values of $\tau(N)$ in the appropriate cases, $R(N)$ can be calculated for varying values of N.

In a more general case, one can have a combination of such different types of algorithms in the software server. For instance, a constant time computation may follow after the initial search. In such cases the expected mean response time $R(N)$ as seen by a request would be a combination of the different service types. Thus if $\tau(N) = B + C.N + D.\log N$ then

$$\lim_{U \to 1} R(N) = A + B.N + C.N^2 + D.N\log N \tag{7}$$

where A, B, C, D are constant coefficients. Equation (2) for the saturation number (N^*) of the system can be re-written for this general case as

$$(N^* - 1).\tau(N^*) = \frac{1}{\lambda}. \tag{8}$$

This is obtained by finding the meeting point of the low-load asymptote, which is $\tau(N)$ in our model, and the high-load asymptote $(N\tau(N) - (1/\lambda))$.

For a sequential search, the expression for N^* can be obtained from (4), (6), (8):

$$N^* = \frac{C_F \pm \sqrt{{C_C}^2 + 2\left(\frac{C_C + C_F}{\lambda}\right)}}{C_C + C_F} \tag{9}$$

4. Model validation

A simulation of a population dependent server was carried out in order to validate the population dependent analytical model. Linear and logarithmic types of service time requirement were simulated by explicitly performing sequential search ($O(N)$) and binary search ($O(\log N)$). The analytical and simulation results were compared for these cases with the parameters: $\lambda = 0.004$ and $C_C = C_F = 1$. Fig. 4 and Fig. 5 clearly show that the analytical and simulation results match very well.

The analytical value of saturation number for sequential search was calculated by solving for N^* using the above parameters in (9). For binary search, a numerical solution of (8) was obtained, referring back to (5) and (6). Solving the equations we get: $N^* \approx 16$ for sequential search, and $N^* \approx 34$ for binary search. These values correspond very well to the plots depicted in Fig. 4 and Fig. 5.

5. Discovering service complexity

In the discussion till now, the composition of service time was known to us. Thus we were in a so called “white-box” scenario with a view of what is going on inside the

software server. However, a useful application of our model is that of "discovering" the complexity of algorithms running in a server that may be a "black-box", i.e. we neither have prior knowledge nor a direct view of the internal working of the application. Thus the only measures available are user-perceived measures, i.e. response time and throughput. In this section, we propose a way to achieve the dual goals of such discovery and subsequently, the prediction of response time of the system.

Such a discovery process may be useful in an application hosting environment where the operator is required to meet all the customers' service level agreements. Before hosting a new application, the operator may want to characterize the performance of the application and discover any hidden scalability bottlenecks. A similar need is also present when an organization integrates software developed by an outside vendor into its IT infrastructure.

Our technique for discovering the service complexity of a black-box server starts with measurements of response time and throughput of the server. To discover the characteristics of the "dependence" of R on N, we heuristically try to fit some common forms, that can be taken by $R(N)$ (Table 1), one by one, to the measured data. The "goodness of fit" in this process is measured by comparing the deviation between the measured and fitted data. This is depicted in "error profile graphs". In the end, a likely estimate of the composition of the service happening in the server is shown in the form of "service profile graphs". These are an estimate of the actual proportion of each kind of service steps happening inside the server. It is worth noting that our technique can give good results with measurements at just 8 to 10 points. The technique is presented below.

5.1. Complexity discovery: A heuristic technique

The steps involved in discovering the complexity of algorithms running in a software server are as follows:

1. Obtain response time and throughput data by varying the number of users between 1 and some N_{max}.
2. Plot the response time data obtained, and from the plot for throughput find the number of users N' where the throughput reaches its peak.
3. Remove the data for the range $[1, N']$ or obtain more data for $[N', N_{max}]$. This is necessary since it is the nature of variation after saturation that brings out the difference between many types of complexities.
4. After pruning the response time data, fit curves representing the commonly expected time-complexities to the observed response time. Do this individually for each service type. We used least-squares curve fitting. This initial curve fitting gives us two kinds of data. A set of fitted values of $R(N)$ for the given values of N, and values of the constants $A, B, C, D, \ldots$ in (7).
5. From the fitted data and the measured data, calculate the sum of square of errors (deviations) in the data, individually for each service type.
6. Now plot an "error profile graph" (e.g. Fig. 10), which is a plot of the error-terms calculated above for each type of service complexity. The service types having the least (or lower) error-terms give an indication of the type of algorithms running in the server.

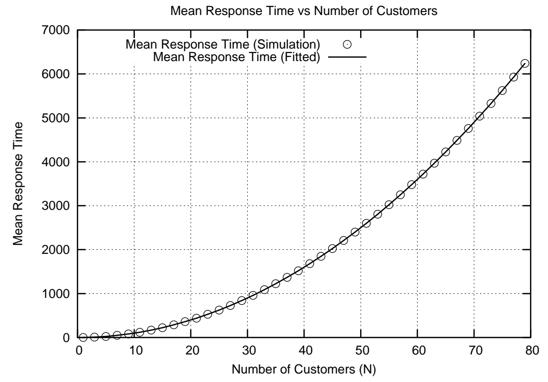

Figure 6. Curve fitting for linear search

A case where the actual service is a combination of multiple service-types will show up as multiple relatively low error-terms in the error profile graphs. In this case, another cycle of curve fitting and error-term evaluation should be done. This time, however, the fitting is done only with a combination of the components having lower error-terms in the previous step. This gives a better estimate of the values of the coefficients for the fitted components. The fitted curve thus obtained, is a fairly good representation of the actual mean response time R with varying population N.

The above method was used on a linear search simulation and another simulation that had a linear search as well as an exponentially distributed random time component with constant mean 0.5. The parameters were $\lambda = 1$ and $C_C = C_F = 1$. Fig. 6 and Fig. 7 show the fitted curves and Fig. 8 shows the corresponding service profile graphs. The graph on the left in Fig. 8 (for the first simulation) shows C as the dominant coefficient (with value $\approx$ 1), thus correctly suggesting a linear-time algorithm. In the graph on the right (for the second simulation), dominant coefficients B and C correctly indicate a combination of $O(1)$ and $O(N)$ service types, with a good match on the values as well.

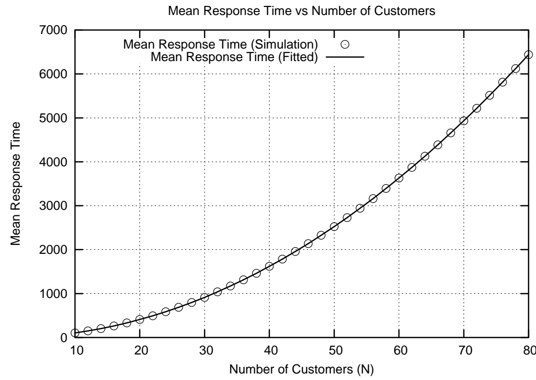

Figure 7. Curve fitting for linear search + a constant mean random component

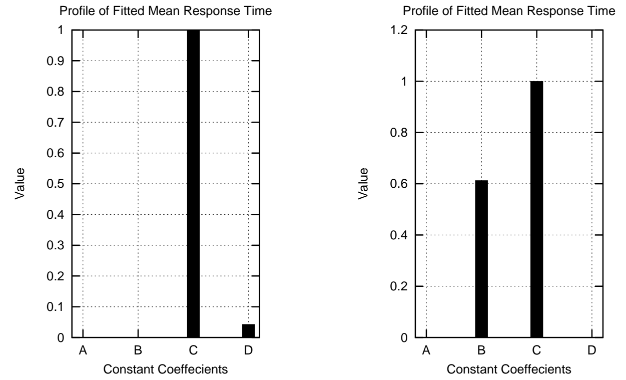

Figure 8. Service profile plots for Figs. 6 & 7. Dominant coefficients are clearly visible.

6. Measurement based validation

In the previous section we have described a technique for discovering the service complexity of an algorithm running in a software server. This technique was validated with tests on response-time data from simulations of various service time complexities. The next step is to move on to measurement based validation and see if the technique works on data gathered from actual applications. In this manner a program based on this technique may be used as a *performance debugging* tool to check the behavior of software.

This technique is meant to be applied in a controlled test environment. Thus the infrastructure that the test is running on will be known. That would help in making sure that factors other than the application under test are not influencing the test. This would minimize/control factors like variable communication delays and network congestion, thus avoiding spurious results. The ideal option is to run these tests on an isolated high-speed LAN. Under such a controlled environment, requests reaching the server are only from load generated as part of the discovery technique. Thus the assumption that the actual number of users are known and fixed during one measurement interval is a valid one.

6.1. Test system

The test system used for measurements was a typical Web-server, hosting static and dynamic Web pages. The Apache Web-server (ver. 1.3.29), using PHP (ver. 4.3.4) as the server-side scripting language, was used to serve Web pages. The host OS was Debian GNU/Linux Sid (updated till June 15, 2004). MySQL (ver. 4.0.18) was used as the back-end database to store session-data and provide database services to the applications under test. The server hardware was an Intel P-IV 2.4 GHz based system with 256MB of RAM and a 40GB hard disk. Client systems used for load generation had similar configurations. The above configuration (LAMP: Linux-Apache-MySQL-PHP) is one of the most preferred software combinations in use today for hosting of dynamic Web-based applications.

Some specific changes were made in the configuration of Apache to ensure that server parameters remain constant throughout the test. `KeepAlive Off` is necessary to make sure that each request makes a fresh contention for the server, irrespective of who made the previous request. `MaxClients 1` is required to make Apache adhere to our model of a single server by using only one "worker" thread. `MaxRequestsPerChild 0 #unlimited` is needed to prevent periodic "re-spawning" of the server.

6.2. Software under test

In order to test the session-management overheads, a custom session management system, which stores the session data in a database, was written. Both the standard and the customized session management systems were tested to be able to distinguish an efficient system from one that runs slower. The standard session management system in PHP is designed to be highly efficient. The custom system stores session-related information in a MySQL database table. A search is done through this table to locate the current session and read the session data.

6.3. Load generation

A session-aware load generator was used to emulate multiple users issuing requests to the Web-server. The load generator was made to follow the model of exponentially distributed think time between successive service requests from a client.

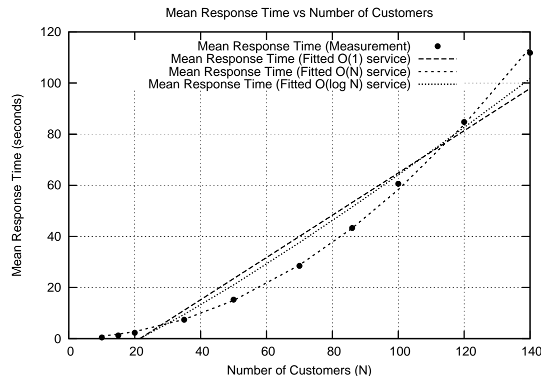

Figure 9. Successive fitting to measured response time (slow linear search)

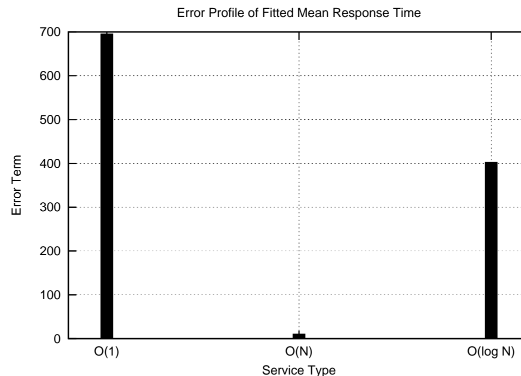

Figure 10. Error profile graph for Fig. 9 (slow linear search)

6.4. Results

Fig. 9 shows the results of measurements when the population was set to $\{1, 2, 5, 10, 15, 20, 35, 50, 70, 86, 100, 120, 140\}$ in order to reflect the irregular points of measurement that may be present in a real scenario. The application under test used a custom session management system that does a linear search through the table, with slow comparisons. The mean think time $(1/\lambda)$ was set to 200ms. Fig. 10 shows the "error profile" which clearly indicates that the error-term is lowest for $O(N)$ service.

To validate these results, service time at the server was measured directly. This was done by using `mod_benchmark` (http://www.trickytools.com/), an Apache module meant for such purposes. A plot of this measured service time vs. population (Fig. 11) clearly indicates that the service time is linear ($O(N)$).

To test our discovery technique further, we carried out measurements on an unmodified PHP-based application that stores the session-table in a database and also generates Web pages dynamically from data pulled out from the database. Thus the service time-complexity depends on the application's internal working and handling of queries by the database engine; we do not have any *a priori* knowledge about the algorithms in the software.

In this case the mean think time was set to 4s. The curve fitting results and error profiles are depicted in Fig. 12 & Fig. 13. The profile on the left in Fig. 13 is the first attempt at fitting. The clearly high error-term for $O(N)$ reveals that $O(N)$ service is not present here. This is shown, for contrast, in Fig. 12. Furthermore, error-terms of $O(1)$ and $O(\log N)$ seem to be close. The next step is to fit a combination of these two types (profile shown on right in Fig. 13). This shows that the combination has the lowest error-term, thus revealing that the service seems to comprise of $O(1)$ and $O(\log N)$ components.

Since in this case, we had access to server-side measurements of service time (from `mod_benchmark`), we used them to verify this predicted composition of service. Fig. 14 shows a plot of the measured service time and a fitted curve.

This confirms that the service is indeed a mixture of a predominant $O(1)$ component and a rather feeble $O(\log N)$ part. Though the test on this application does not reveal any great dependence of R on N, our technique itself is shown to work well towards detecting such a dependence. The coefficients obtained in the discovery process can be used to predict the response time for any given N. This illustrates the usefulness of our model and heuristic technique in performance prediction and debugging.

7. Conclusions and future work

In this work we started by asking ourselves a simple question: how would response time vs population curves look for servers that have population-dependent service times? We presented a simple model to capture this behavior, and specifically validated it for servers performing linear and binary searches with data from simulations/measurements on an actual software server.

We extended this idea further by suggesting a "complexity discovery" technique, which works only on user-perceived performance measures. This heuristic technique was applied and tested on simulation-generated data and data from measurements on real-world applications.

We believe our results and technique will be of practical value to software developers and operators of application hosting centers. Future work involves refining the discovery technique and applying it in a more general scenario where the fitting curve may not be just a linear combination of N, $N \log N$ & N^2. Another direction for future is extending the model to multi-threaded servers, and cases having multiple classes of requests, each having its own service-type.

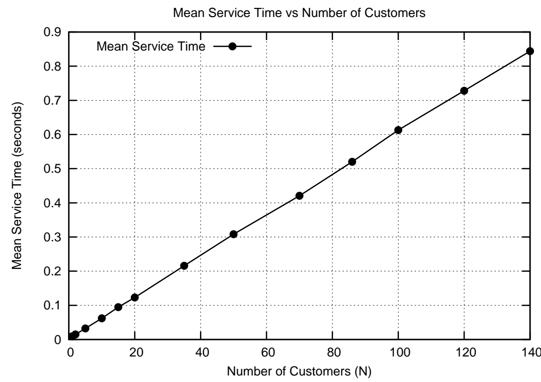

Figure 11. Measured mean service time (slow linear search)

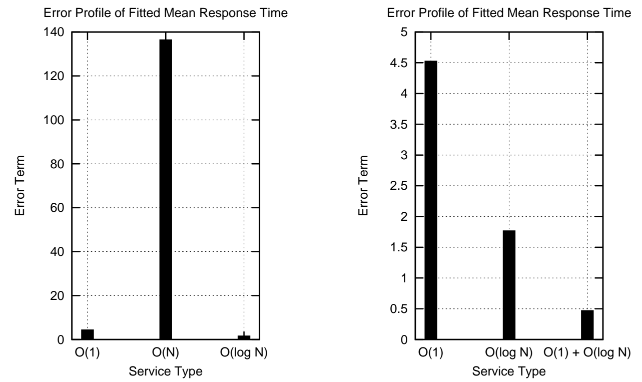

Figure 13. Error profile graphs (sessions and application data in database, two steps)

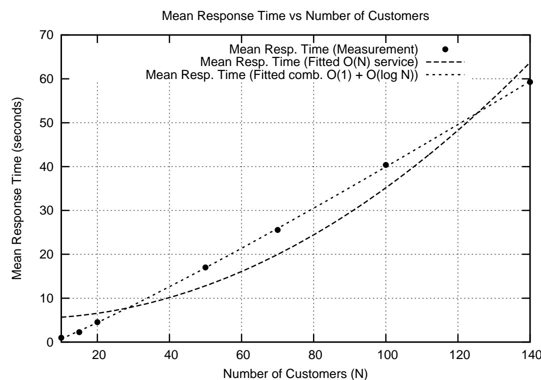

Figure 12. Discovery technique results (sessions and application data in database)

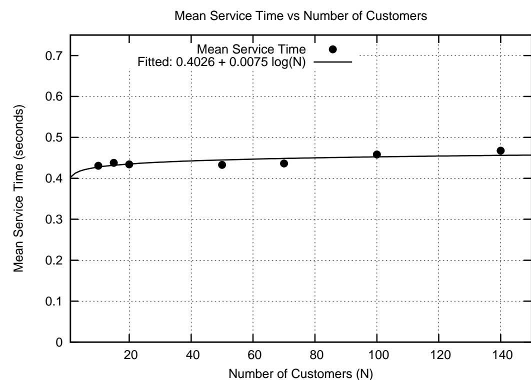

Figure 14. Measured mean service time (sessions and application data in database)

References

[1] V. Apte, T. Hansen, and P. Reeser. Performance comparison of dynamic web platforms. *Computer Communications*, 28(8):888–898, may 2003.

[2] A. Avritzer, R. Farel, K. Futamura, M. Hosseini-Nasab, A. Karasaridis, V. Mainkar, K. Meier-Hellstern, P. Reeser, P. Wirth, F. Hubner, and D. Lucantoni. Internet application performance: A signature-based empirical approach. In *International Teletraffic Congress*, 2001.

[3] T. T. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[4] M. Curiel and R. Puigjaner. Using load dependent servers to reduce the complexity of large client-server simulation models. In *Performance Engineering, State of the Art and Current Trends*, pages 131–147. Springer-Verlag, 2001.

[5] G. Franks and M. Woodside. Performance of multi-level client-server systems with parallel service operations. In *Proceedings of the First International Workshop on Software and Performance*, pages 120–130. ACM Press, 1998.

[6] L. Kleinrock. *Queueing Systems*, volume II: Computer Applications. Wiley-Interscience, New York, 1976.

[7] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik. *Quantitative System Performance*. Prentice Hall, Englewood Cliffs, New Jersey, 1984.

[8] P. Reeser and R. Hariharan. An analytic model of web servers in distributed computing environments. *Telecommunication Systems*, 21(2):283–299, 2002.

[9] J. A. Rolia and K. C. Sevcik. The method of layers. *IEEE Trans. on Soft. Engg.*, 21(8):689–700, Aug 1995.

[10] C. Sauer. Computational algorithms for state-dependent queueing networks. *ACM Trans. on Comp. Sys.*, 1(1), 1983.

[11] K. S. Trivedi. *Probability and Statistics with Reliability, Queuing and Computer Science Applications*. John Wiley and Sons Ltd., New York, 2001.

[12] B. Venners. *Inside the Java Virtual Machine*. McGraw-Hill Professional, second edition, 1999.