Dynamic Parameter Substitution for solution of Layered Queueing Networks with Timeout Decisions

Lianhua Li
Department of Systems and Computer Engineering
Carleton University
Ottawa, ON Canada
lianhua@sce.carleton.ca

Greg Franks
Department of Systems and Computer Engineering
Carleton University
Ottawa, ON Canada
greg@sce.carleton.ca

ABSTRACT

Solving performance models using queueing networks poses a challenge because parameters such as service times and routing are fixed and must be known prior to solution. Models which involve *decisions* based on performance quantities must therefore often be solved using a state-based model, through simulation, or a hybrid combination of these approaches. However, state-based approaches suffer from state space explosion for even moderately sized models, while simulation can be time consuming. This paper broadens Layered Queueing Networks (LQN) to handle a subset of models with state-based behaviour, namely systems with timeouts and aborts. The approach, called dynamic parameter substitution (DPS), updates the parameters of the underlying queueing networks of a LQN model as the model is being solved. This approach in this paper is both fast and highly scalable, compared to simulations (LQSIM and CSIM) and a hybrid solution.

CCS Concepts

•Computing methodologies → Modeling methodologies; •Software and its engineering → Software performance; System modeling languages; Model-driven software engineering; *Layered systems;*

Keywords

dynamic parameter substitutions; timeout; abort decisions; layered queueing network; performance modelling

1. INTRODUCTION

In the field of Software Performance Analysis, analytic performance modeling has many advantages. It can be done at any stage of the development process, the time required for the analysis is small, and it is of relative low cost compared to other methods [9, 18]. Analytic solutions to performance models can be found using a wide variety of methods, though queuing models were found to be the preferred approach [2, §4.1.1]. Queueing models scale well, solve quickly and have sufficient accuracy [18, §3.2].

The downside of queueing models is their inability to model arbitrary dynamic data-dependent behaviour. One important form of behaviour that fits this category is exception handling. Exceptions are defined as *"Of the conditions detected while attempting to perform some operation, exception conditions are those brought to the attention of the operation's invoker. The invoker is then permitted (or required) to respond to the condition."* [8]. From the standpoint of a performance model exceptions can be raised as a result of trying to acquire a resource.

The exception pattern considered in this paper is the *timeout* pattern, shown in the sequence diagram in Figure 1. This pattern arises when the response time for a service request exceeds some value, T_{timeout}, then simply gives up. Note that there may be some additional costs in giving up, so this action can also affect the performance of the system. The abort decision [20], is considered a special case of a *timeout* decision when the threshold of timeout, $T_{timeout} = 0$.

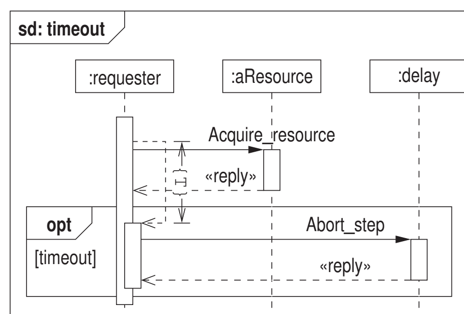


Figure 1: Timeout decision pattern.

The LQN *Timeout* decision pattern was defined in the previous work [11], and combines both timeout and abort decision logic. This generalized timeout decision was implemented in the LQN simulation tool, *LQSIM*. The results produced by LQSIM were verified by the results of the Petri Net solver, GreatSPN [4, 1] using the equivalent Generalized Stochastic Petri Nets (*GSPN*) [12] models. In this paper, LQSIM is used to verified the results of the analytic solution of DPS.

Queueing models are not suitable for modeling exceptions

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
VALUETOOLS 2016, October 25-28, Taormina, Italy
Copyright © 2016 EAI 978-1-63190-141-6
DOI 10.4108/eai.25-10-2016.2267111

because the likelihood of an exception being raised must be known prior to solving the model. However, exceptions are often raised because of results the performance model produces. For example, excessive response times can lead to timeouts and excessive utilization can lead to lock request failures. One approach to solve this problem in a scalable and timely fashion is through hybrid modeling, where a queueing model is used to solve the bulk of the model, and another model is used to solve the decision, for example the Hybrid Performance Modeling Methodology (HPMM) [20]. However, in general this approach suffers from the need for two or more different models and the need for data interchange between them all.

This paper takes another approach: feed the output of the performance model back as input. This approach is called Dynamic Parameter Substitution (DPS). The advantage of DPS is that iterations between disjoint solvers do not have to be performed. The feasibility of this approach is demonstrated by using the Layered Queueing network eXperiment control language (LQX) [14] to perform the feedback and iteration using a LQN. DPS is developed using LQX to solve a system with a timeout/abort. The results from this model are compared to simulation.

The remainder of the paper is organized as follows. Section 2 describes layered queueing networks, their solution and the LQX language. Section 3 explains how to solving timeout decision using dynamic parameter substitutions and compares the results of DPS to simulation. Section 4 applies the DPS solution to solve a web-base database model (from [20]) with a timeout and abort decision. Finally, Section 5 concludes the paper.

2. LQN MODELS

A layered queueing network is a type of extended queueing network that is particularly suitable for solving performance models of software using remote procedure calls. With this type of software system, clients of servers block awaiting replies to requests. These servers may in turn make requests to other servers. Layering arises by following the call chain from the top-most client all the way down to the lowest level server.

Figure 2 shows an example of a LQN. This model, adapted from [20], represents client web-browsers making requests to a web server, which in turn makes requests to a database. The elements in the model are:

Processors: represented using circles in the figure. Processors are used to consume time in the model. If the processor is a *multi-server*, it is shown using a stacked icon, and the number of copies, n, are shown using a label with brackets '`{n}`'. The output result for a processor is its utilization, U.

Tasks: shown using the large parallelograms in the figure. Tasks are used to model actual tasks running on a processor (e.g. dbWorker), customers to the model (e.g. users), logical resources such as database locks (e.g, dblock), and other physical resources which are not processors such as disks. Tasks may be multi- or infinitely-threaded, in which case they are shown using a stacked icon and labelled accordingly. The primary output result for a task is its throughput, λ.

Entries: shown using small parallelograms within a task. Entries are used to differentiate the service provided by a task. Entries have *service-time* parameter, s, shown using square brackets '`[s]`' which is consumed on the associated processor. The primary output result for an entry is it's residence time, x, which is the sum of the response times (queuing plus service) to all the entries it calls plus the response time at its processor.

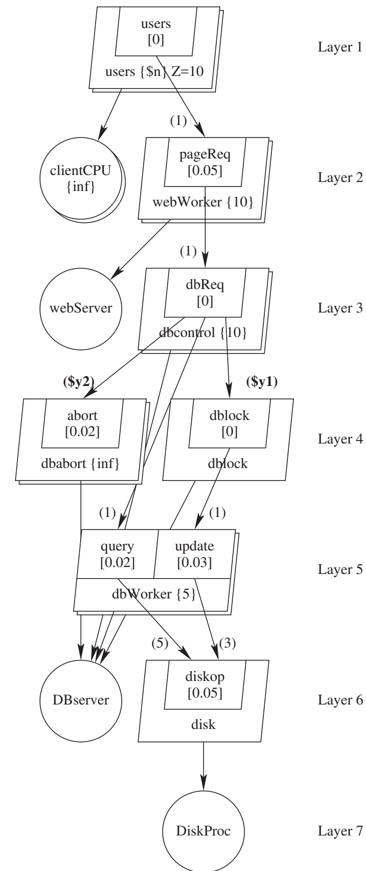


Figure 2: LQN model of a Web Server adapted from Figures 5-15 and 5-24 from [20] for the Abort Pattern.

Calls: shown using directed arcs (directed arcs from tasks to processors represent the call for demand on a processor). Calls have a *rate* parameter, y, shown using parenthesis '`(y)`', representing the average number of times the source entry makes a request to its destination entry. The output result for a call is the queuing time, q, at the destination task.

There are many more features of the model which are not described here; refer to [5] for a more complete description of the model. Additional explanations of the basic concepts behind layered queueing can be found in [17, 19].

2.1 Solving a LQN Model

To solve a LQN, the LQNS analytic solver, used here, creates a set of ordinary queueing network "layer submodels", each of which is solved using Mean Value Analysis (MVA) [16]. Other solvers for layered resources that use a similar ap-

proach include [5, 10, 13, 15, 17, 19]. The solution process has six steps:

1. Load the model.
2. Construct LQN model objects with parameters.
3. Construct layer submodels by finding all clients making calls to servers in layer $i, i \geq 2$. Note that a task may be used as a *server* in one submodel, but may also be used as a *client* in multiple submodels. Processors are never clients and client tasks representing users are never servers. Figure 3 shows all of the submodels from Figure 2 that involve the task dbcontrol.
4. Initialize layer submodels. Service times, visits and populations for each of the queueing models are initialized based on the input values.
5. Iterate to solve layer submodels to convergence.
6. Write output file.

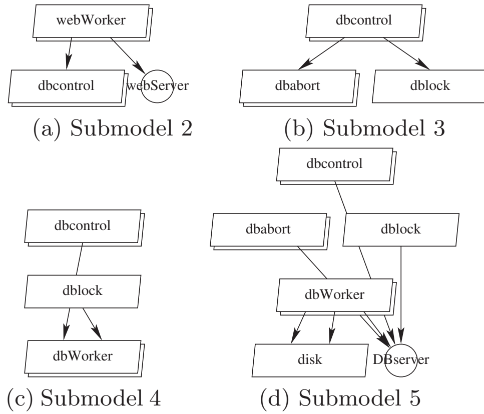


Figure 3: Submodels containing task dbcontrol

2.2 Iterative Solution

LQNS uses four levels of iteration in step 5 to solve a model:

1. The experiment control interpreter, LQX (described in §2.3) calls `solve()`.
2. `Solve()` calls Linearizer [3] over all queueing submodels until convergence.
3. Linearizer calls Bard-Schweitzer at populations $\mathbf{N}$ and with one customer removed from each chain k, $\mathbf{N} - \mathbf{1}_k$, for a submodel.
4. Bard-Schweitzer calls `core()` to find the residence times at the servers until the residence times converge. Response times, w, are found at each server, m, using the function `wait(m)`.

After performing Step 4 above, response times w in a submodel are found from each *client* to its *servers* and are stored in a vector, $\mathbf{W}_i$, indexed by *submodel i*. This vector is then used to set the demands for all of the processors and tasks in the model, regardless of whether the object is acting as a client or a server. When solving submodel i when an object m is acting as a *client*, all but item i in $\mathbf{W}$ are summed to find the service time demand, s:

$$s_{mi} = \sum_{\forall j, j \neq i} W_{mj} \tag{1}$$

This result is the waiting time caused by calls to servers not in the current submodel and is a surrogate delay in the queueing model. When solving submodel i when an object is acting as a *server*, all of the items in $\mathbf{W}$ are totalled:

$$s_m = \sum_{\forall j} W_{mj} \tag{2}$$

After solving submodel i, update $\mathbf{W}$ at index i for all *clients* in the submodel and save the throughput, λ, and utilization, U, for all servers. For example, the waiting time of dbcontrol are set each time queueing submodels 3, 4 and 5 have been solved.

The think time for chain, k, when a task is being used as a *client* is calculated using:

$$Z_m = \frac{(N_k - U_m)}{\lambda_m} \tag{3}$$

where N_k is the population in the chain and U and λ are the utilization and throughput of the task when is was acting as a server in a submodel. For dbcontrol, these values are set after queueing submodel 2 has been solved.

Waiting times propagate up one level while think times propagate down one level for each iteration of Step 2. Step 2 identified above terminates after the root mean square of the differences in the utilizations reaches some small value.

2.3 LQX

Model solution and parameter iteration is controlled using the Layering Queuing eXperiment control language (LQX) [14]. LQX can be written explicitly as part of XML model input (e.g., Listings 1). LQX is high-level, dynamically typed language with a syntax that resembles C. The choice of syntax was made because of pervasiveness of C and it's derivatives; as such it supports many of the language constructs of C. LQX also brings in more modern concepts such as associative arrays and `foreach` loops for iterating over them. LQX supports many common mathematical functions plus functions for generating random variables with various distributions. Finally, it includes I/O operations for reading and writing model data, and for reading LQX programs from external sources.

LQX supports four types: *numeric*, *boolean*, *string* and *object*. Types are automatically inferred. All numeric types are represented internally as double-precision floating point numbers. LQX is strongly typed so mixed type expressions are not permitted without explicit type conversion. The *object* type is used to access objects in the input model and result attributes such as throughput and utilization, namely *processor*, *task*, *activity*, *entry*, *phase*, and *call*.

Input values are set by using `$`-variables in the input specification and then writing to these variables using a LQX program. A `$`-variable may occur more than once in the input model, so one variable can be used to update multiple parameters simultaneously. There is no support for changing the *structure* of a model through LQX. Rather, separate model files must be created, though they can all share the

same LQX program through the use of the LQX import facility.

The LQX function `solve()` is used to solve the model and can be invoked once all of the `$`-variables have been set. If this function is omitted in an LQX program, `solve()` is invoked implicitly after the program runs.

3. SOLUTION OF DYNAMIC PARAMETER SUBSTITUTION

Dynamic parameter substitution extends the core idea of layered queueing, that is, modifying the parameters of one submodel based on results found from other submodels. Layered queueing typically modifies service and think times of the underlying queueing models. Dynamic parameters extend this concept to allow any input parameter of the underlying queueing network to be modified based on one or more output parameters. For this work, updates to parameters are done using user-defined LQX expressions at the outermost iteration of the solver.

3.1 Dynamic Parameter Substitution

The principle idea behind dynamic parameter substitution is to extend layered queueing networks to handle decisions. The request rates, y, of the execution paths of a decision are set using dynamic parameters, which change as the model is being solved similar to the way the service times are changed over time. A `for` loop is used to call `solve()` repeatedly until the difference in input parameters becomes sufficiently small. This is equivalent to hybrid performance modelling since this approach solves the model to completion each time `solve()` is invoked. However, there is no requirement to solve a model to completion before changing parameters, as this is the essence of layered queueing as described earlier. In [6], all of the inner iterations (Steps 3 and 4 in §2.2) were removed leaving only the outer iteration to execute. For fixed rate and infinite servers, this approach produced the same results as the more expensive technique. Because of this observation, subsequent versions of the solver will move the update process into the inner iterations.

One other necessary step is to find suitable expressions for mapping output parameters such as throughput and utilization to input parameters. These expressions for timeout patterns are found below. Because the DPS solution of timeout decisions is using the DPS solution for abort decision, it is necessary to introduce the DPS solution for abort decision first.

3.2 Abort Decision

The decision on whether to take the success or abort path is based on the probability of finding a customer queued for resource. These probabilities are Pr{success} and Pr{abort} respectively and are used to set the call rates `$y1` and `$y2` in Figure 2. The probability of going to the abort path Pr{abort} is the probability of finding requests in the queue, but not in service, in the success path to dblock, i.e.:

$$\Pr\{\text{abort}\} = \frac{q_{\text{resource}}}{s_{\text{resource}} + q_{\text{resource}}} \quad (4)$$

$$\Pr\{\text{success}\} = 1 - \Pr\{\text{abort}\} \quad (5)$$

where s is the residence time (including requests to lower-level servers) at resource and q_{resource} is the time a request is waiting in the queue but not in service.

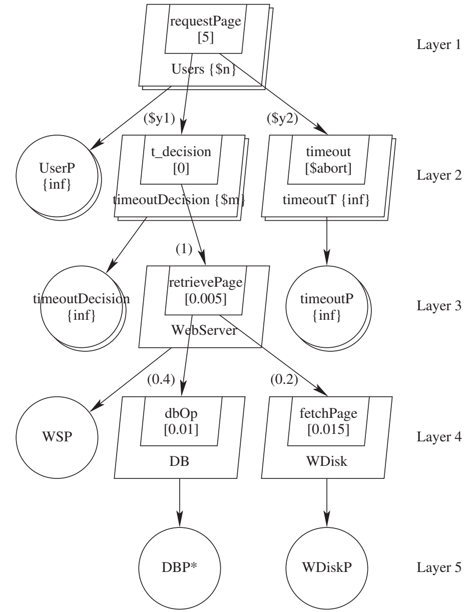


Figure 4: A web server system with timeout.

Increasing the value of `$y2` removes requests that are aborted and also reduces the traffic seen at a server which in turn makes the likelihood of an abort smaller. The next iteration of the solution will then add some traffic back until convergence, then `$y2` reaches the desired value of the abort probability.

3.3 Solving Timeout Decisions

Modeling a timeout directly using feedback may have problems when the timeout probabilities are small. The solution here is to construct an abort decision to solve a timeout decision by adding the auxilliary pseudo task. Figure 4 shows an example of the use of this technique where requests from Users to WebServer can timeout. After adding the auxilliary pseudo task timeoutDecision into this model, the reference task requestPage, the pseudo task timeoutDecision, and the delay task timeoutT form an abort decision. The task timeoutDecision is a pseudo task added to the model to find the response time (queueing plus service) at WebServer to determine whether a request will or will not timeout. The multiplicity of this task, m, is set to

$$m = max\left((m_{\text{WebServer}} + 1), \left\lceil \frac{T_{\text{timeout}}}{x_{\text{WebServer}}} \times m_{\text{WebServer}} \right\rceil\right) \quad (6)$$

where x is the response time at the task WebServer (found using (2)), and $m_{\text{Webserver}}$ is the number of threads of the task WebServer. Each copy of the multiserver timeoutDecision which is busy (i.e., the utilization of the webserver) represents requests which do not time out. Conversely, the queue at timeoutDecision represents requests that do timeout. Note that the multiserver parameter m may change during solution due to delays propagating up to WebServer as the model is solved. m has a minimum value of $(m_{\text{WebServer}}+1)$, which ensures that some requests queue at the resource.

The second part of the solution is to find the probability

of a timeout, which is used to set `$y1` and `$y2` in Figure 4 where `$y2` is the probability of a timeout, represented using y_2 below. Two other variables are needed, the *upper*-bounds, U, and *lower*-bounds, L, of y_2 which are initialized to 1 and 0 respectively and converge towards each other as the solution progresses.

$$L^{(i+1)} = \max(y_2^{(i)}, L^{(i)}) \quad q_1^{(i)} > 0 \tag{7}$$

$$U^{(i+1)} = \min(y_2^{(i)}, U^{(i)}) \quad q_1^{(i)} = 0 \tag{8}$$

where q_i is the mean queue length at timeoutDecision and i is the iteration number. The probability of a timeout is calculated as the waiting time, q_1, over the residence time at timeoutDecision, x.

$$\Pr\{\text{timeout}\} = \frac{q_1^{(i)}}{q_1^{(i)} + x^{(i)}} \tag{9}$$

Finally, the new value for `$y2` is found using:

$$y_2^{(i+1)} = \begin{cases} \Pr\{\text{timeout}\} & q_1^{(i)} > 0, L \leq \Pr\{\text{timeout}\} \leq U \\ \frac{L^{(i+1)}+U^{(i)}}{2} & q_1^{(i)} > 0, \Pr\{\text{timeout}\} < L \\ \frac{L^{(i+1)}+U^{(i)}}{2} & q_1^{(i)} > 0, U < \Pr\{\text{timeout}\} \\ \frac{L^{(i)}+U^{(i+1)}}{2} & q_1^{(i)} = 0 \end{cases} \tag{10}$$

The LQX code for solving the models is in Listing 1. Line 18 is used to find m, the number of copies at timeoutDecision, using (6). Lines 20 and 28 are used to find the lower (7) and upper bounds (8) respectively. Finally, lines 23, 25, and 30 are used to find the new value of `$y2` using (10).

The solution for timeout decisions can be easily extended to handle abort decision by setting the dynamic parameter m equal to $m_{\text{Webserver}}$, when $T_{timeout} = 0$ at line 15 in Listing 1.

However, there is a special case needs to be considered. The first condition of this case is the probability of timeout is small, more precisely, when the mean queueing delay at the resource is smaller than $T_{timeout}$. The second condition of this case is the dynamic parameter m is also small, for example, less than five. In this case, $Pr\{timeout\}$ may be very sensitive to the mean response time at the resource at some points. This problem can severely affect the results of timeout probabilities and also may lead to the convergence failure if this problem occurs during solution iterations. The experiments show that small changes upon the response time of a resource server may cause a large difference in the timeout probabilities. For example, the model used in the case study, given $T_{timeout}$ is 0.6 seconds, when the mean response time of the resource (the writer lock) are 0.29 and 0.31 seconds (around 15 customers in the system), the corresponding timeout probabilities are 0.22 and 0.4. The reason is that the value of m is changed from three to two, which results the mean queueing time (q_1) at the decision is increased from 0.2 to 0.4. Consequently, the corresponding timeout probabilities are significantly larger than the simulation results. To address the problem of small timeout probability, the $Pr\{timeout\}$ in (9) is adjusted by a ratio using the queueing time at the resource (q_2), which

Listing 1: LQX program for solving timeout

```
1  lowerbound = 0.001;
2  upperbound = 0.999;
3  T_timeout = 1.0;
4  $abort = T_abort+T_timeout;
5  $m_res = 1;
6  for ($n = 100; $n <= 1000; $n += 100) {
7    $m = $n; i = 1;
8    upper = upperbound; $y1 = upper;
9    lower = lowerbound; $y2 = lower;
10   for (delta = 2; delta > 0.000005 || i <= 2;
         delta = (last_y2-$y2)**2) {
11     last_y2 = $y2;
12     solve();
13     s_succ = phase(entry("retrievepage"),1).
           service_time;
14     w_succ = call(phase(entry("requestPage"),
           1),"t_decision").waiting;
15     if (T_timeout == 0.)
16       $m = $m_res;
17     else
18      $m=max(($m_res+1),ceil(T_timeout/s_succ));
19     if (w_succ > 0.) {
20       lower = max($y2,lower);
21       if (lower <= $y2 && $y2 <= upper) {
22         prob = w_succ/(w_succ+phase(entry(
               "t_decision"),1).service_time);
23         $y2 = ($y2+2.0*prob)/3.0;
24         if ($y2 <= lower || upper <= $y2)
25           $y2 = (upper+lower)/2.0;
26       }
27     } else if (i > 1) {
28       upper = min($y2,upper);
29       if ($y2 > lowerbound)
30         $y2 = (upper+lower)/2.0;
31     }
32     $y1 = 1 - $y2;
33     i += 1;
34   }
35 }
```

is shown in (11).

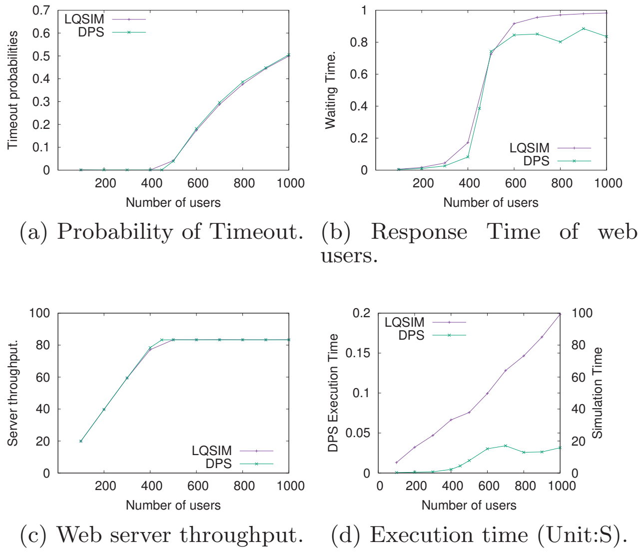
$$\Pr\{\text{timeout}\} = \frac{q_1^{(i)}}{q_1^{(i)} + x^{(i)}} \times \frac{q_2^{(i)}}{q_1^{(i)} + q_2^{(i)}} \tag{11}$$

The rationale for this adjustment is that when small values of m result in larger values for q_1, they also result in smaller values for q_2. Therefore, the smaller q_2 can stop the increase of $Pr\{timeout\}$. If a model has a larger probability of timeout or a larger value of m, this problem can be eliminated. This fact is demonstrated by the results in the next section, by using the web server model in Figure 4.

3.4 Results

The model shown in Figure 4 is a simple web server model with timeouts that was used to evaluate DPS algorithm in Listing 1. Figures 5a through 5c show the results from solving this model using DPS and from simulation. Simulations were run with confidence intervals of 95%. Simulation results were found using a version of lqsim [7] modified to handle timeouts for requests directly in [11]. These results show that DPS compares favourably to simulation in calculating the server throughput and the probability of a timeout with an average error of under 1%. DPS is less accurate at calculating the waiting time from requestPage to t_decision.

Figure 6 shows the probabilities of timeouts when the



(a) Probability of Timeout. (b) Response Time of web users.

(c) Web server throughput. (d) Execution time (Unit:S).

Figure 5: Results from solving the model timeout shown in Figure 4 with lqsim and DPS.

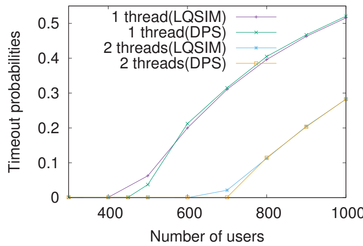


Figure 6: The probabilities of timeouts for the web server model with one thread and two threads resource server.

web server has one and two threads, while $T_{timeout}$ is set to 0.8 second. The results of DPS in both cases are compare favourably to simulation. Lastly, Figure 5d shows the execution time of DPS and simulation time when using a 3GHz Intel Core 2 Duo processor. When the system has 1000 customers, DPS only took 0.025 seconds to solve the model, while the simulations took around 100 seconds on average 95% of the time. Therefore, DPS is significantly efficient comparing to simulation.

4. CASE STUDY

The model used in this case study, shown in Figure 2, was adapted from the model used by Wu [20] to develop a hybrid solution for exception handling in the context of a web application that accesses a database. This model emphasizes the handling of decisions made at the write lock. Update requests from users arrive at dbcontrol, at which point a lock request is made to the underlying database dbWorker. If the lock is available the request follows the call from dbcontrol to dblock. Otherwise, this request will wait for the lock until the specified period expires. This request may acquire the lock or be rejected by routing it to dbabort.

To handle the timeout decision using DPS, the pseudo task dblock was added to this model. The decision on whether to take the success or timeout/abort path is based on the probability of finding a customer queued for this task. These probabilities, namely Pr{success} and Pr{abort}/Pr{timeout}, are used to set the call rates `$y1` and `$y2` respectively in Figure 2.

4.1 Case 1: Abort Decision

When the timeout threshold $T_{timeout}$ is ZERO, this timeout decision is an abort decision. In this case, the dynamic parameter $\$m$, the multiplicity of the pseudo (timeout) task, is set to one, since the writer lock (dblock) only can have one thread.

Figures 7a through 7e show the results from solving the model using DPS and LQSIM. DPS has average relative errors of less than 4% when compared to the simulation in calculating the lock probability, the holding time for the lock and the "useful" throughput (throughput at entry dblock). Figure 7 also shows the results from solving the model using CSIM and HPMM from [20]. DPS is more accurate when calculating the response time at the web server at high loads when compared with the hybrid approach HPMM.

4.2 Case 2: Timeout Decision

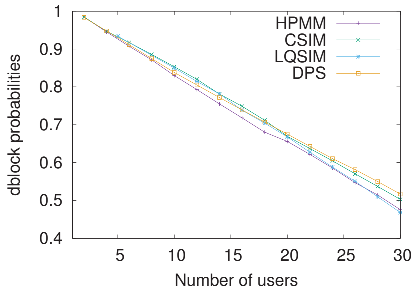
Case 2 uses the database model with a timeout decision shown in Figure 2 where $T_{timeout}$ is set to 0.6 and is an example of the special case discussed in Section 3.3 which occurs when the number of customers is less than 22. Figure 8 shows the results from the model when solved using DPS and LQSIM. The results from DPS are quite accurate in solving the probabilities of timeouts in the special case. However, there is a discontinuity of the probability of lock acquisition in Figure 8a where the calculation of $Pr\{timeout\}$ switches from using (11) to (9) because the mean queueing delays of requests are close to $T_{timeout}$. After this point, the error rates of the probabilities of lock acquisition are less than 8%. The results for the mean lock holding time, the mean response time of web users, and the utilization of the disk (the bottleneck device) are all quite accurate. The error rates of the useful throughput, the throughput of the lock, are same as the error rates of the probabilities of lock acquisition.

Finally, Figures 7f and 8f show the execution time of DPS and simulation for both cases. All the DPS executions are less than 0.8 second, while simulation times are around 30 50 seconds. However, DPS usually requires a more iterations when solving timeout decisions than solving abort decisions.

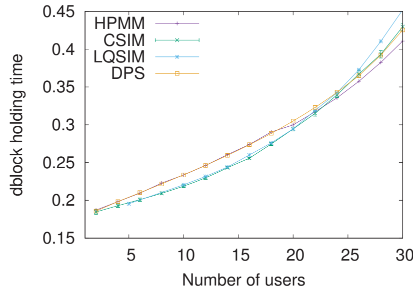
5. CONCLUSIONS

This paper has shown how the Layered Queueing Experiment Language has been used to implement Dynamic Parameter Substitution to allow the solution of performance models with state-based behaviour, which emphasizes on timeout decisions here. Pure queueing-based solutions to models of this type are not possible because all of the parameters to a queueing network must be known prior to solution. Consequently, state-based solutions, simulation, or combinations of these two techniques with a queueing model have been employed in the past.

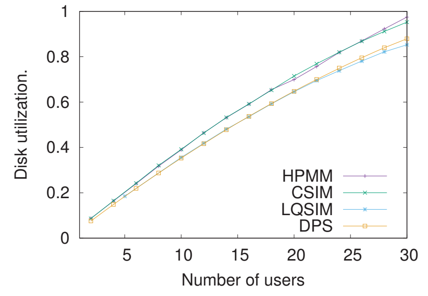
Dynamic Parameter substitution takes advantage of the very method used to typically solve a layered queueing network analytically: iterative solution among multiple submodels. With this approach, the results of the solution of a



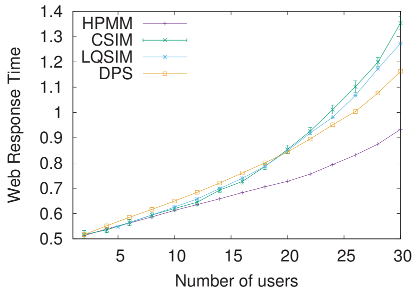
(a) Lock acquisition probability (Fig. 5-40).



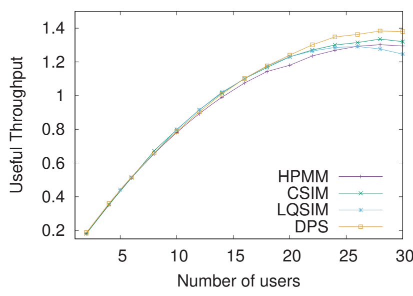
(b) Mean lock holding time (Fig. 5-41).



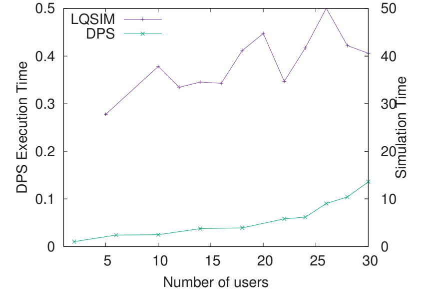
(c) Bottleneck utilization (Fig. 5-42).



(d) Response time of the Web Server (Fig. 5-43).

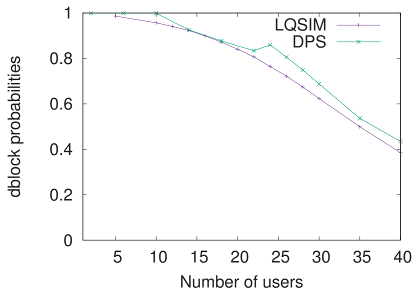


(e) Useful throughput (Fig. 5-44).

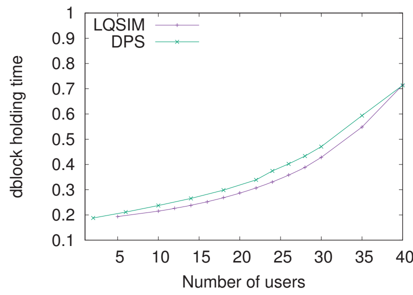


(f) Execution time(Unit:S.).

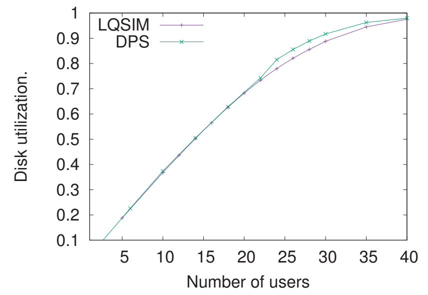
Figure 7: Results from solving the abort model shown in Figure 2 with CSIM, HPMM and DPS. CSIM and HPMM results are from Figures 5-40 through 5-44 of [20].



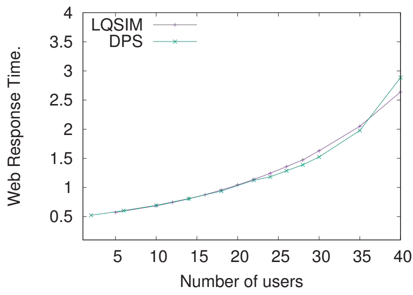
(a) Lock acquisition probability.



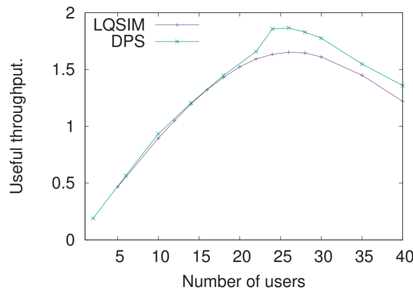
(b) Mean lock holding time.



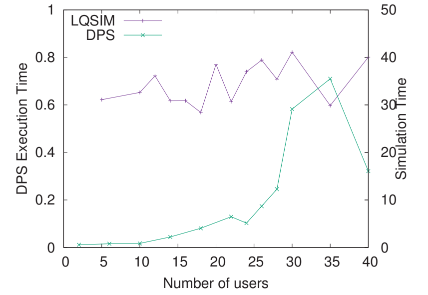
(c) Bottleneck utilization.



(d) Response time of the Web Server.



(e) Useful throughput.



(f) Execution time (Unit:S).

Figure 8: Results from solving the timeout decision ($T_{timeout} = 0.6$) model shown in Figure 2.

submodel are used as input to other submodels. Dynamic parameter substitution extends this method to allow arbitrary output to be fed back as input for subsequent iterations of the model. This method is implemented using the LQX language here.

DPS was used to solve LQN models with generalized timeouts, which cover both timeout (on queueing delays) and abort decisions. The novelty of the solution here was the use of a multi-server pseudo task to partition the queue where customers could experience timeouts. Customers that cannot timeout are represented by the copies of the multiserver while customers that will timeout are represented by the queue of the multiserver. This approach was very effective in finding the timeout probabilities and throughput for the system when compared to simulation. With DPS, the average relative error in throughput was less than 8%. DPS has a higher accuracy in solving systems with large number of customers.

The DPS solution is quite efficient. All the execution times of DPS are much less than the simulation time. Since all computation is performed within the context of a single operating system process, no file input/output or process switching is required. This could be of tremendous benefit than the hybrid approach's solution .

The only deficiency of the DPS approach is the requirement to find expressions for feedback. In the future, common decision-based performance scenarios will be integrated into the LQN framework, thus relieving the modeller of this requirement.

Acknowledgement

This research was supported by the Natural Sciences and Engineering Research Council of Canada.

6. REFERENCES

[1] S. Baarir, M. Beccuti, D. Cerotti, M. De Pierro, S. Donatelli, and G. Franceschinis. The GreatSPN tool: Recent enhancements. *SIGMETRICS Perform. Eval. Rev.*, 36(4):4–9, Mar. 2009.

[2] S. Balsamo, A. D. Marco, P. Inverardi, and M. Simeoni. Model-based performance prediction in software development: A survey. *IEEE Trans. Softw. Eng.*, 30(5):295–310, May 2004.

[3] K. M. Chandy and D. Neuse. Linearizer: A heuristic algorithm for queueing network models of computing systems. *Commun. ACM*, 25(2):126–134, Feb. 1982.

[4] G. Chiola, G. Franceschinis, R. Gaeta, and M. Ribaudo. GreatSPN 1.7: GRaphical Editor and analyzer for timed and stochastic Petri nets. *Performance Evaluation*, 24(1):47–68, 1995.

[5] G. Franks, T. Al-Omari, M. Woodside, O. Das, and S. Derisavi. Enhanced modeling and solution of layered queueing networks. *IEEE Trans. Softw. Eng.*, 35(2):148–161, Mar.–Apr. 2009.

[6] G. Franks and L. Li. Efficiency improvements for solving layered queueing networks. In *Proc.* 3rd *WOSP/SIPEW International Conference on Performance Engineering (ICPE '12)*, pages 279–282, Boston, MA, USA, Apr. 22–25 2012.

[7] G. Franks, P. Maly, M. Woodside, D. C. Petriu, and A. Hubbard. *Layered Queueing Network Solver and Simulator User Manual*. Real-time and Distributed Systems Lab, Carleton University, Ottawa. http://www.sce.carleton.ca/rads/lqn/lqn-documentation/.

[8] J. B. Goodenough. Exception handling: Issues and a proposed notation. *Commun. ACM*, 18(12):683–696, Dec. 1975.

[9] R. Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. John Wiley & Sons, 1991.

[10] P. Kähkipuro. UML based performance modeling framework for object oriented systems. In *UML '99, The Unified Modeling Language, Beyond the Standard*, volume 1723 of *Lecture Notes in Computer Science*, pages 356–371. Springer-Verlag, Berlin, 1999.

[11] L. Li and G. Franks. Modelling decisions in layered queueing networks. July 24–27 2016.

[12] M. A. Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with Generalized Stochastic Petri Nets*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1994.

[13] D. A. Menascé. Two-level iterative queuing modeling of software contention. In *Proc.* 10th *IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS 2002)*, Fort Worth, TX, Oct. 12–16 2002.

[14] M. Mroz and G. Franks. A performance experiment system supporting fast mapping of system issues. In *Proc.* 4*th* *International ICST Conference on Performance Evaluation Methodologies and Tools (VALUETOOLS '09)*, Pisa, Italy, Oct. 20–22 2009.

[15] S. Ramesh and H. G. Perros. A multi-layer client-server queueing network model with synchronous and asynchronous messages. In *Proc.* 1st *International Workshop on Software and Performance (WOSP '98)*, pages 107–119, Santa Fe, NM USA, Oct. 12–16 1998.

[16] M. Reiser and S. S. Lavenberg. Mean value analysis of closed multichain queueing networks. *J. ACM*, 27(2):313–322, Apr. 1980.

[17] J. A. Rolia and K. A. Sevcik. The method of layers. *IEEE Trans. Softw. Eng.*, 21(8):689–700, Aug. 1995.

[18] C. U. Smith. Introduction to software performance engineering: Origins and outstanding problems. In M. Bernardo and J. Hillston, editors, *Formal Methods for Performance Evaluation (SFM 2007)*, volume 4486 of *Lecture Notes in Computer Science*, pages 395–428. Springer-Verlag, Berlin, 2007.

[19] C. M. Woodside, J. E. Neilson, D. C. Petriu, and S. Majumdar. The stochastic rendezvous network model for performance of synchronous client-server-like distributed software. *IEEE Trans. Comput.*, 44(8):20–34, Aug. 1995.

[20] P. Wu. *Extending Layered Queueing Network with Hybrid Submodels Representing Exceptions and Decision Making*. PhD thesis, Department of Systems and Computer Engineering, Carleton University, Ottawa, ON, Canada, May 2013.