Heuristic Derivation of a Fluid Model from a Layered Queueing Network*

Murray Woodside
Carleton University
Ottawa, Canada
cmw@sce.carleton.ca

ABSTRACT

Fluid approximations are useful for representing transient behaviour of queueing systems. For layered queues a fluid model has previously been derived indirectly via transformation first to a PEPA model, or via recursive neural networks. This paper presents a derivation directly from the layered queueing mechanisms, starting from a transformation to a context-sensitive layered form. The accuracy of predictions, compared to transient simulations and steady-state solutions, is evaluated and appears to be useful.

CCS CONCEPTS

• Distributed computing models • Software performance • Markov processes

KEYWORDS

Performance models; Layered Queues; Fluid Approximations.

ACM Reference format:

Murray Woodside. 2018. Heuristic Derivation of a Fluid Model from a Layered Queueing Network. In *the Companion of the 2023 ACM/SPEC International Conference on Performance Engineering (ICPE'23 Companion), April 15–19, 2023, Coimbra, Portugal.* ACM, New York, NY, USA. *7 pages.* https://doi.org/10.1145/3578245.3584852

1 INTRODUCTION

A number of problems in performance management can benefit from a dynamic model of the system to predict the short-term effects of management actions. Model-based management has often used steady-state models and ignored the transient, on the assumption that the transient is short-lived (e.g. [5]). Model parameter tracking by Kalman Filters as in [19] also assumes a steady state reached over each measurement interval, yet the intervals may be too short to obtain a steady state; tracking based on a dynamic model should be better.

ICPE '23 Companion, April 15–19, 2023, Coimbra, Portugal

ACM ISBN 979-8-4007-0072-9/23/04…$15.00
https://doi.org/10.1145/3578245.35848520

A fluid model is a tractable way to make approximate dynamic predictions. This work describes a heuristic direct derivation. Fluid models for performance approximate the populations of customers at queues by continuous variables governed by an ordinary differential equation (ODE). Models derived by the mean-field theory of Kurtz [6] represent the expected value $x(t)$ of the random population vector $\tilde{x}$, with derivatives given by the expected rate of change of $\tilde{x}$:

$d\,\tilde{x}/dt$ = Exp($d\,\tilde{x}/dt$) = Exp{f($\tilde{x}$)}

and then approximate the right-hand side by a function of x.

Layered queueing networks (LQNs) are extended queueing networks with simultaneous resource possession of many resources. Steady-state performance estimates are found by approximate Mean Value Analysis (AMVA) [4][12]. Two approaches have been taken to make a fluid model of a LQN. The first transforms it to a PEPA model (based on process-algebra concepts), and uses a fluid approximation to the PEPA model [15]; the second creates a single approximate closed queueing network and transforms it [1][10]. This paper takes a different approach which avoids introducing extraneous modeling concepts such as PEPA and follows LQN semantics more closely.

In this paper the fluid model is based directly on the mechanisms of the layered servers and their simultaneous resource possession. The analysis is based on the methods of mean-field analysis, [6][11] but is heuristic and does not include any investigation of convergence to the solution of the underlying Markov model. Some LQN features are not covered by the transformation described here but the necessary extensions seem to be straightforward. The paper describes the derivation and some experience applying it to performance models. The representation of transient behaviour is satisfactory, for the purposes envisaged. The accuracy in steady-state is less good.

2 LAYERED QUEUEING

Figure 1 shows a layered queueing model, displayed as a high level architectural model with performance parameters. Software services are called *tasks* (eg. DB). Each task is a multiserver queue with a multiplicity of m_T for task T (for multi-threading), and runs on a *processor* (eg. DB_P) with a multiplicity of m_P for processor P (for multi-cores). The system users are modeled by one or more User tasks (called "reference tasks" below) have a multiplicity equal to the number of users, and a demand d_i equal to the thinking time.

A task offers one or more operations called *entries* (eg. dbAccess). Entry E_i has mean execution demand d_i and makes an

average of y_{ij} calls to other lower-layer entries E_j. Between calls, E_i demands slices of execution from its processor, with mean slice demands of $d_i/(1+\sum_j y_{ij})$. Calls may be synchronous or asynchronous, but this paper considers models with only synchronous calls. We suppose that each user of the system is represented by a token, which moves from the user task to the entries as they are executed.

Since its first introduction in [16], most layered queueing solutions have been for steady-state measures via AMVA [4][12], as for example implemented in the LQNS solver [3][2]. The LINE tool added fluid solutions [9][1].

2.1 Context-Sensitive LQN for Fluid Modeling

The context of an entry execution is the set of entries along its calling path. In general an entry can execute in multiple contexts, called from different higher-level entries. To uniquely define the transitions in the state vector, the LQN model is transformed to a context-sensitive form, in which each entry has only one caller. This is done by splitting the model entries with multiple callers into identical copies, one per caller. The detailed behaviour of the model and thus its performance measures are unchanged by the splitting. The number of additional entries is in general more than linear in the number of excess callers because nested calls may also produce additional splits. In Figure 1(b) the entries appRequest and dbAccess have been split, and the split in appRequest causes an additional split in dbAccess.

Instead of splitting the entries, an alternative approach (taken in [15]) would be to send the reply to any one of the set of active callers, choosing non-deterministically). However this would require additional state to keep track of which call an entry is blocked on, and would give the same complexity.

2.2 The LQN Forwarding Transformation

A (synchronous) forwarded call transfers the return address along with a request, so the server can reply directly to the original caller. This mechanism is used to model referral of calls, and pipelining. In the LQNS solver the "forwarding transformation" [3] replaces the sequence of calls by a set of synchronous calls from the original caller, without imposing a sequence on them. This transformation was applied to forwarded calls. It gives the correct total average delay in a steady state analysis, but because it replaces the sequence of single calls by a random collection of calls in any order, it is not correct for transient analysis. A more complete transformation would replace the call by a sequence of activities, each making one call.

3 THE FLUID APPROXIMATION

The fluid approximation to the expectation of queue states is a differential equation whose elements estimate the expected queue state at time t. Following [6] the derivation considers a large ensemble of models and expected rates of change based on a large system population, but has also been found to be useful for modest state populations.

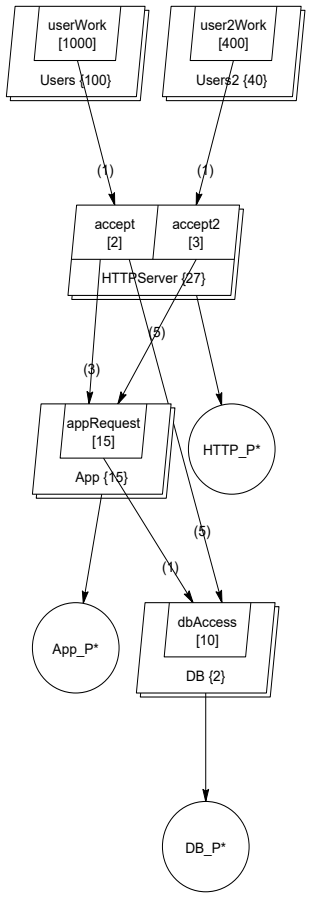


Figure 1(a) LQN for a Two-Tier System

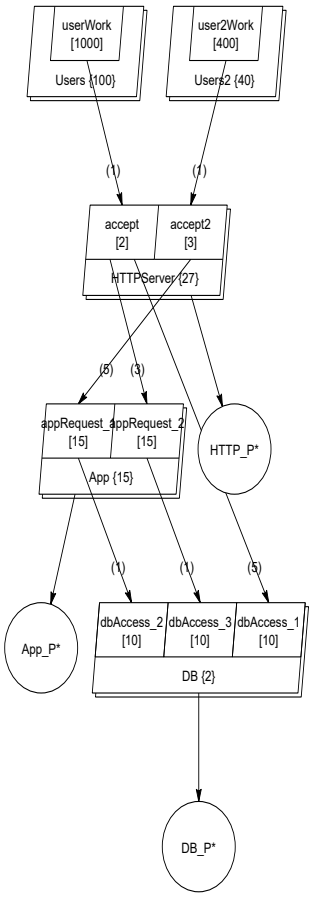


Figure 1(b) Entries Duplicated by Context

The main approximation is to replace the expectation of a nonlinear function of state by the function of the expected state, which is not necessarily a good approximation. Some heuristic adjustments are made to try to improve this.

The state is represented by tokens, one for each system user, that are located at different system entries. Each task is a server with a queue, and a class of service for each entry.

Define:

$\tilde{x}_i(t)$ = the random value of the state at time t, which is the number of tokens at E_i (on a call from E_i to E_j, the token moves to E_j)

$x_i(t)$ = the (approximate) expected value of $\tilde{x}_i(t)$

$T(i)$ = the task of which E_i is a part,

$m_{T(i)}$ = the multiplicity of $T(i)$

$\mathcal{E}_{T(i)}$ = the set of indices of entries of $T(i)$.

$P(i)$ = the processor for $T(i)$.

$\mathcal{E}_{P(i)}$ = the set of indices of entries that execute on $P(i)$.

$x^{T(i)}$ = the total tokens at $T(i)$, $\quad x^{T(i)} = \sum_{j\in\mathcal{E}_{T(i)}} x_j$.

x_i^S = the *in-service* part of x_i which is executing at $T(i)$. These tokens are queued at the processor $P(i)$.

x_i^Q = the *queued* part of x_i which is queued at $T(i)$, $x_i^Q = x_i - x_i^S \geq 0$.

$m_{T(i)}$ = the multiplicity of $T(i)$.

$m_{P(i)}$ = the multiplicity of $P(i)$.

$\boldsymbol{D}(i)$ = direct descendants of E_i, the set of entries E_j which are called directly by E_i ($y_{ij} > 0$).

$\boldsymbol{D}^+(i)$ = all descendants of E_i, being the set of entries E_j which are called directly and indirectly by E_i.

$A(i)$ = the unique calling entry of E_i (its ancestor).

Because of synchronous calls, when an entry makes a call its task thread is blocked, even though the token moves to the called entry. The number of blocked threads at $T(i)$ is $b_i(x)$:

$$b_i(x) = Exp\{\textstyle\sum_{E_j \in \boldsymbol{D}^+(i)} \tilde{x}_j\} = \sum_{E_j \in \boldsymbol{D}^+(i)} x_j \quad (1)$$

At the processor $P(i)$, $\tilde{x}_i^S(t)$ *in-service* tokens for E_i are requesting service, along with tokens for other entries that share $P(i)$. The number of busy servers of the multiserver $P(i)$ is

$$m' = \min\left(m_{P(i)},\ \textstyle\sum_{j \in \mathcal{E}_{P(i)}} \tilde{x}_j^S(t)\right)$$

and the mean number of servers devoted to tokens for E_i is

$$\tilde{x}_i^P(t) = \mathrm{Exp}\left\{(\tilde{x}_i^S(t)/\textstyle\sum_{j \in \mathcal{E}_{P(i)}} \tilde{x}_j^S(t)))m'\right\}$$
$$= \mathrm{Exp}\left\{\tilde{x}_i^S(t)\min\left(1,\ m_{P(i)}/\textstyle\sum_{j \in \mathcal{E}_{P(i)}} \tilde{x}_j^S(t)\right)\right\} \quad (2)$$

The non-linear function $\min(1, f(\tilde{x}))$ has an expectation less than $f(x)$ for x near to 1, depending on the distribution of $\tilde{x}$. A consideration of this issue in [13] led to a very successful proposal to replace the min function by a smoothed version of the form:

$$\mathrm{smoothmin}(1, y) = \quad y/((1 + y^p)^{\frac{1}{p}}) \quad (3)$$

using a value of p which depends on the distribution of the random function represented here by y. Here it is proposed to use smoothmin with a fixed value of p and replace Eq (2) with:

$$x_i^P(t) = x_i^S(t)\mathrm{smoothmin}\left(1,\ m_{P(i)}/\textstyle\sum_{j \in \mathcal{E}_{P(i)}} x_j^S(t)\right) \quad (4)$$

Rates will be expressed as r events/s, with a superscript to identify the event type. An entry is executed in slices separated by calls, giving a mean of $(1 + \sum_j y_{ij})$ slices for E_i with a mean demand of $d_i/(1 + \sum_j y_{ij})$. With an average of $x_i^P(t)$ servers given by Eq (4) the expected rate of completion of slices of E_i is

$$r_i^{slice} = x_i^P(t)\sigma_i\,[d_i/(1 + \textstyle\sum_j y_{ij})]^{-1} \quad (5)$$

where σ_i is a speed factor for $P(i)$.

The state evolves as tokens arrive at an entry, are admitted to the task thread pools, make calls, and finish task service. The rate of tokens finishing service at E_i is

$$r_i^{reply} = \text{reply rate} = r_i^{slice}/(1 + \textstyle\sum_j y_{ij}) \quad (6)$$

The rate of tokens at E_i making calls to other entries is

$$r_i^{calling} = \text{calling rate} = r_i^{slice} y_{ij}/(1 + \textstyle\sum_j y_{ij}) \quad (7)$$

The arrival rate at E_i due to calls from other entries is

$$r_i^{called} = \text{called rate} = \textstyle\sum_j r_j^{slice} y_{ji}/(1 + \sum_k y_{jk}) \quad (8)$$

The rate of receiving replies at E_i is

$$r_i^{replied} = \text{replied rate } \textstyle\sum_{j \in D(i)} r_j^{reply} \quad (9)$$

The rate of admission r_i^{admit} to the thread pool of task $T(i)$ is at least the arrival rate r_i^{called} when there are free threads (when $\tilde{x}^{T(i)} + b_i < m_{T(i)}$). A maximum value r_i^{best} was set to the arrival rate plus the maximum possible rate of completions with no competition and no nested calls, which is $m_{T(i)}/d_i$:

$$r_i^{admit}(\tilde{x}_i) = r_i^{best}$$
$$= r_i^{called} + m_{T(i)}/d_i, \quad \text{if } \tilde{x}^{T(i)} + b_i < m_{T(i)}.$$

When all threads are blocked or busy, it is limited by the rate of service completions at $T(i)$:

$$r_i^{completion} = \textstyle\sum_{j \in \mathcal{E}_{T(i)}} r_j^{reply}$$

In this case the admission rate of tokens for E_i is assumed to be in the ratio of its task queue size x_i^Q (an approximation for FIFO queueing), giving:

$$r_i^{admit} = r_i^{completion}\,(\tilde{x}_i^Q/\textstyle\sum_{j \in \mathcal{E}_{T(i)}} \tilde{x}_j^Q), \text{ if } \tilde{x}^{T(i)} + b_i \geq m_{T(i)}$$

The admission rate has a step between two values, which gives a more severe non-linearity than the min function. It was also smoothed for the same reason, using a well-known function for a unit step:

$$\mathrm{step}(y) = 0 \text{ if } y < 0, \text{ or } 1 \text{ if } y \geq 0$$
$$\mathrm{smoothstep}(y) = 3(y/\epsilon)^2 - 2(y/\epsilon)^3$$

where ϵ was fixed. This gives the approximation

$$r_i^{admit}(x_i) = r_i^{best} +$$
$$[r_i^{TASK}(\tilde{x}_i^Q/\textstyle\sum_{j \in \mathcal{E}_{T(i)}} \tilde{x}_j^Q) - r_i^{best}]\ \mathrm{smoothstep}(\tilde{x}^{T(i)} + b_i - m_{T(i)}) \quad (10)$$

A reference task receives no calls and admits all replies, so $r_i^{admit} = r_i^{replied}$.

This admission rate ignores the fact that queueing at tasks is usually FIFO. The information in $x(t)$ is insufficient to represent the detailed state of a FIFO queue, which includes the order in the queue of requests for different entries, and the residual service time of the currently served tokens. On the other hand a FIFO queue treats the classes symmetrically, as does the approximation. This question was treated differently in [1], by a transformation to the processor-sharing discipline for the queue; this approach can be examined in future work.

The in-service tokens $x^S(t)$ and the queue length $x^Q(t)$ are both needed to calculate the rates of change, so both are modeled as states, with $x = x^S + x^Q$. The differential equations are then:

$\dot{x}_i^S$ = admit rate+sum of reply rates to it−calling rate−reply rate

$$= r_i^{called} + \textstyle\sum_{j \in D_i} r_j^{reply} - r_i^{calling} - r_i^{reply} \quad (11a)$$

$\dot{x}_i^Q$ = called rate − admit rate = $r_i^{called} - r_i^{admit}$ (for non-reference tasks). (11b)

Reference tasks representing system users receive no calls and make no replies, so $r_i^{called} = r_i^{reply} = 0$. They have no queue so there is no need for a queue state variable for them.

In summary, the steps to compute the right-hand sides replace the random variable $\tilde{x}$ by its expectation x and:

1. Determine $b_i(x)$ from Eq (1) and $x_i^P(x)$ from Eq (4) for all i,
2. Determine $r_i^{slice}, r_i^{reply}, r_i^{calling}, r_i^{called}, r_i^{replied}$ from Eq (5) – (9),

3. Determine r_i^{admit} from Eq (10),
4. Determine the right-hand sides from Eq (11a), (11b).

The ODEs were solved by the Matlab solver ode45 which was sufficiently fast.

3.1 Performance Measures

Some steady-state performance measures are:

- Entry throughput is the rate of departure of tokens, which is r_i^{reply}.
- Wait for service at entry E_i is, by Little's formula:

$$W_i = r_i^{reply} x_i^Q \quad (12)$$

- Entry service time X_i between admitting a call for service, and replying, is made up of two parts, the entry execution time $(1 + \sum_j y_{ij})/r_i^{slice}$ and the delay for nested calls (if any):

$$X_i = (1 + \sum_j y_{ij})/r_i^{slice} + \sum_{j \in \boldsymbol{D}(i)} y_{ij}(W_j + X_j). \quad (13)$$

- The response time seen by a User (represented by an entry of a reference task) is its service time X_i minus the think time.
- Entry utilization = $r_i^{reply} X_i$. (14)
- Task utilization = $\sum_{i \in \mathcal{E}_T} r_i^{reply} X_i$ (15)

3.2 Limitations and Possible Extensions

The above transformation has the following limitations, however removing them appears to be mostly straightforward.

1. All calls are synchronous.
2. No open arrivals processes.
3. There are no activities within entries; this can be introduced by defining an in-service state for each activity.
4. Incomplete handling of forwarded calls. The forwarding transformation described above can improved by moving the calls to activities in the originating entry which mirror the forwarding path structure.
5. No priorities.

Zero (or very small) demand on an entry creates numerical problems for solving the differential equation due to stiffness.

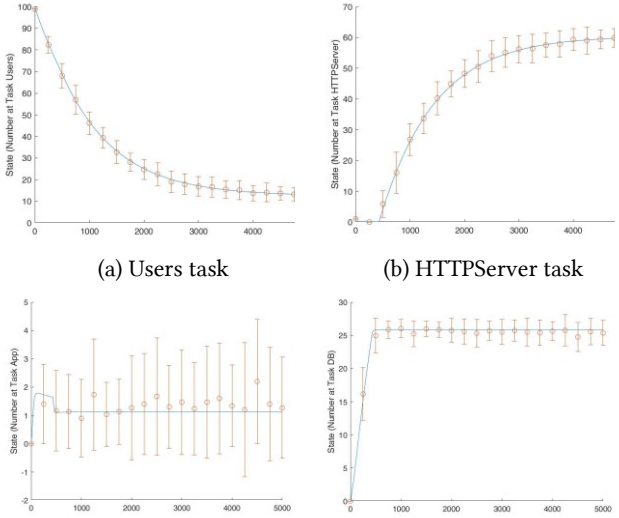
4 TESTING

The quality of the approximation was tested by comparing its predictions to simulations by LQSIM [2]. The simulation started from an idle system with N users. Across 30 independent runs the mean and standard deviation of the states of the tasks were computed and plotted for comparison with the fluid approximation. The states of entries on each task were summed because only task values were provided in the LQSIM trace.

4.1 Simple Two-Tier Example

The first example is a simplification of the model shown in Figure 1, without the second class of users (User2). There were 100 users and the comparison was made over 5000 ms.

In Figure 2 the simulation results are plotted as the mean plus/minus the standard deviation (not the confidence interval). The agreement is excellent. The onset of saturation at DB is clear, and most of the Users are blocked at the HTTPServer and App (a layered bottleneck).



(a) Users task (b) HTTPServer task

(c) App task (d) DB task

Figure 2. Comparison of the Fluid Model Predictions to Simulation for the Simple Two-tier Model

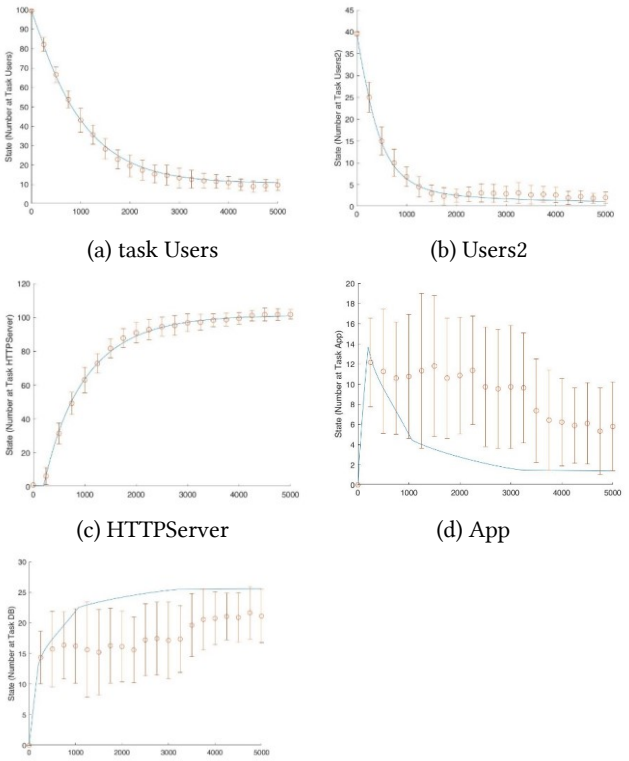
The entry throughput and waiting measures at t=10000 are compared to the steady-state estimates found by LQNS in Table 1. Notice that the entries of DB were split in creating the fluid model. The agreement is very good except for the waiting time calculation at entry appRequest.

Table 1 Steady-State Performance Measures vs LQNS

Entry	Throughput		Wait for Entry	
	fluid	lqns	fluid	lqns
userWork	0.0125	0.0127		
accept	0.0125	0.0127	4840	4760
appRequest	0.0375	0.0380	15	7.38
dbAccess_1	0.0625	0.635	215.6	229.9
dbAccess_2	0.0375	0.0381	276.9	229.8

4.2 Two-Tier System with Two Classes of Users

The model as shown in Figure 1 with two classes of users was compared in the same way with the results in Figure 3. The results for the lower layer servers are not as accurate, under-estimated for App and overestimated for DB. However the upper layers are modeled very well. It appears that the traffic to DB directly from HTTPServer displaces some of the traffic from App, compared to the simulation. This did not happen in the same way with just one class, but with two classes there is an additional degree of freedom, and the two classes put very different loads on DB.



(a) task Users (b) Users2

(c) HTTPServer (d) App

(e) DB

Figure 3. Comparison of the Fluid Model Predictions to Simulation for the Two-class Version

The steady-state results comparison by entry is given in Table 2. Compared to LQNS, the predicted throughput is shifted in favour of the first group of users, giving it a longer wait at the HTTPServer entry accept and increasing the load on appRequest. In this context the smaller waiting time at appRequest is surprising.

Table 2 Steady state results for the two-class model

Entry	Throughput fluid	lqns	Wait for Entry fluid	lqns
userWork	0.01071	0.00991		
user2Work	0.002814	0.004396		
accept	0.010713	0.00991	6285.3	7084.7
accept2	0.002814	0.004396	11896	7091.1
appRequest	0.04646	0.05172	14.92	62.06
dbAccess	0.0955	0.10127	247.1	187.6

4.3 A Business Reporting System (BRS) Example

This example was taken from [8][17] and is based on an industrial business intelligence system. It creates small and large reports and provides viewing of stored reports. The system has eight layers, so congestion propagation between layers is important. Figure 4 shows the context-transformed model, in which the split entries (near the bottom of the diagram) are indicated by suffices "_1", "_2", etc.

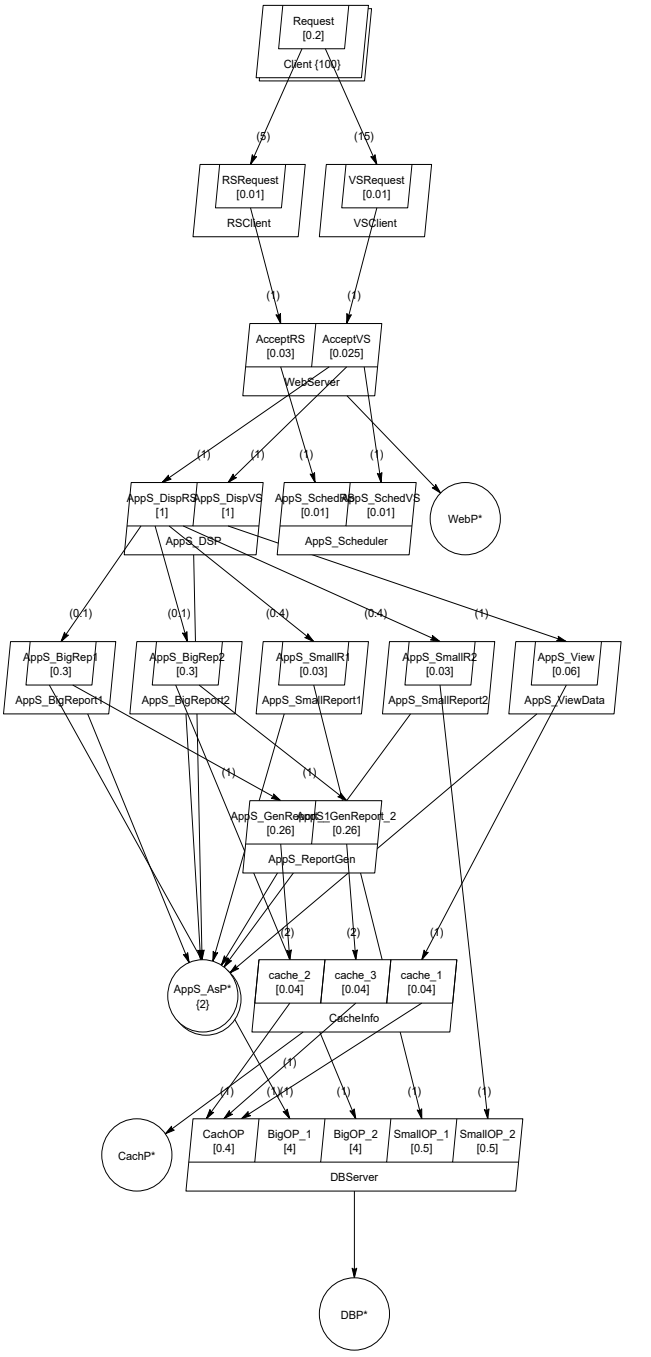


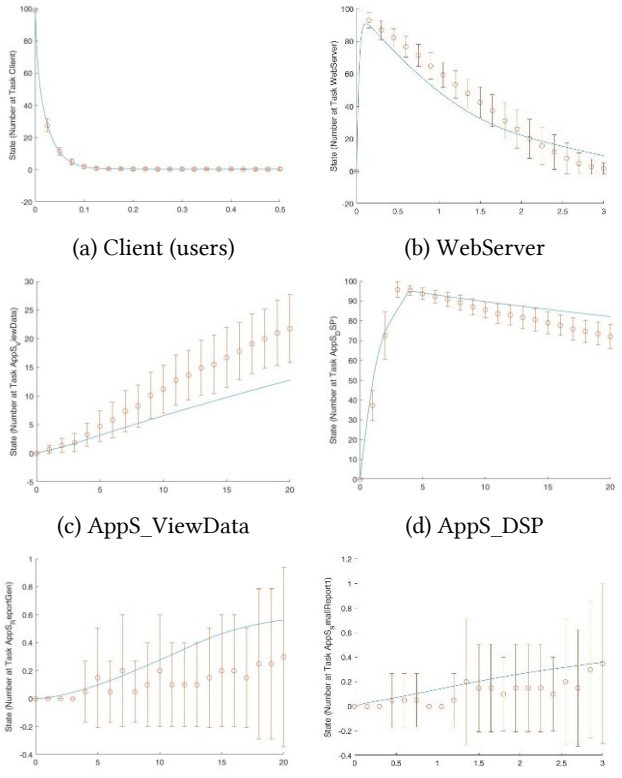
Figure 4. The BRS model after the context and forwarding transformations

Figure 5 shows a representative selection of the simulation comparisons for the 12 tasks. The remaining tasks had very small values. The traces cover different time periods depending on how

quickly they approach a steady state; the transients at the AppS_DSP and AppS_ViewData tasks are even longer than is shown.

Transient effects are very prominent in the start-up situation evaluated here, as the first requests propagate downwards through the layers. Notice how the number of in-service tokens at the WebServer task jumps upwards and then decays away; it eventually becomes quite small. The population of AppS_DSP has a long transient and becomes small at about 180 s., while Apps_ViewData rises steadily to that point in time and levels off at just above 70 tokens. AppS_ViewData is a slightly overloaded bottleneck which gradually builds up until it holds a significant fraction of the requests.

The agreement is quite good for all 12 tasks, although the approximation shifts some load from ViewData to DSP This example has a notable range of fast and slow behaviour by different tasks.



(a) Client (users) (b) WebServer

(c) AppS_ViewData (d) AppS_DSP

(e) AppS_ReportGen (f) AppS_SmallReport1

Figure 5. BRS Predictions Compared to Simulation

5 PREVIOUS WORK

Mean-field theory has been described for modeling the evolution of the expectation of populations within a Markovian system by, eg. [6][11]. Mean-field models for queues and queueing networks have a considerable literature which is surveyed, along with other models, in [14]. Much of the attention has been focused on queueing networks with a combination of infinite servers and processor-sharing nodes, eg [9]. An improved approach to such networks is described in [13], including mixed open and closed classes. It was shown in [10] how to derive the distribution of response times from a fluid model.

The first fluid model for LQN was based on first transforming it to a PEPA model [15]. The PEPA model deviates from the LQN semantics in two ways: it replaces FIFO queueing by random-order (by making a non-deterministic choice from a pool of requests), and it replaces the sending of a service reply to the caller, by sending to a non-deterministic choice from the set of callers waiting for a reply. The first of these is effectively also part of the present transformation, but the second is dealt with by the context transformation.

Fluid models in the LINE tool [1],[9] are created by the transformation described in [10]. From the description in [10] an LQN is converted to a queueing network in which the software queues are not represented, equivalent to all the tasks being infinite servers. Instead, the requests for service at a task are passed directly to the corresponding processor queue.

6 CONCLUSIONS

The state of a LQN has more structure than the state of a queueing network because of the simultaneous resource possession implied by blocking of task threads for nested requests, and this is reflected in the state structure used here. On the other hand the class structure and class switching is implicit in the entries (the customers at each entry are a separate class) and does not need to be considered explicitly, as in models for queueing networks.

The model derived here shows a reasonable accuracy for transient prediction, but needs to be improved for steady-state accuracy.

The features of LQN models that are not covered by this work can be included in an extended transformation. Activities define detailed sequences of operations within an entry, and would require additional state for the activities that are in-service. Forwarded messages are presently dealt with as in LQNS (thesis) but could be better handled by constructing a sequence of activities to order them. Second phases also require activities. Arrival processes and asynchronous calls (which give a mixed open/closed system) will be a significant extension.

ACKNOWLEDGMENTS

This research was supported by grant RGPIN-2016-06274 from the Natural Sciences and Engineering Research Council of Canada.

REFERENCES

[1] G. Casale, 2020. Integrated performance evaluation of extended queueing network models with LINE, Proc Winter Simulation Conference, pp 2377–2388.

[2] G. Franks, G. et al, 2022. Layered Queueing Network Solver and Simulator User Manual, Carleton University, at http://www.sce.carleton.ca/rads/lqns/userman22.pdf, accessed Jan 20, 2022.

[3] G. Franks. 1999. Performance Analysis of Distributed Server Systems. Ph.D. thesis, Carleton University.

[4] G. Franks, T. Al-Omari, M. Woodside, O. Das, S. Derisavi, 2009. Enhanced modeling and solution of layered queueing networks, IEEE Trans. on Software Eng., vol. 35, no. 2, pp. 148-161.

[5] A. Gias, G. Casale, M. Woodside, 2019. ATOM: Model-driven autoscaling for microservices. Int. Conf on Distributed Computing Systems, pp 1994-2004.

[6] T.G. Kurtz, 1970. Solutions of ordinary differential equations as limits of pure markov processes," J. Applied Probability, vol. 7, no. 1, pp. 49-58.

[7] E. Incerto, M. Tribastone, C. Trubiani, 2017. Software performance self-adaptation through efficient model predictive control, Proc 32nd IEEE/ACM Int. Conf. on Automated Software Engineering (ASE), , pp. 485–496.

[8] A. Martens, H. Koziolek, S. Becker, R. Reussner. 2010. Automatically improve software architecture models for performance, reliability, and cost using evolutionary algorithms. In Proc. 1st Joint WOSP/SIPEW Int. Conf. on Performance Engineering. ACM, New York, NY, 105–116.

[9] J. F. Pérez, G. Casale, 2017. Line: evaluating software applications in unreliable environments. IEEE Trans. Reliab. 66(3): 837-853.

[10] J. F. Perez, G. Casale, 2013. Assessing SLA compliance from palladio component models, Proc. 15th Int. Symp. Symbolic Numeric Algorithms Sci. Comput., 2013, pp. 409–416.

[11] E. Renshaw, 2011. Stochastic Population Processes, Oxford University Press.

[12] J. A. Rolia and K. C. Sevcik. 1995. The method of layers. IEEE Trans. Softw. Eng. 21, 8, 689–700.

[13] J. Ruuskanen, T. Berner, K. Årzén, A. Cervin,2021. Improving the mean-field fluid model of processor sharing queueing networks for dynamic performance models in cloud computing, Performance Evaluation, 151 .

[14] J.A. Schwarz, G. Selinka, R. Stolletz, 2016. Performance analysis of time-dependent queueing systems: Survey and classification, Omega 63, pp 170–189.

[15] M. Tribastone, 2013. Fluid model for layered queueing networks, IEEE Trans. Software Engineering, v. 39, pp 744-756.

[16] C.M. Woodside, 1986. An active-server model for the performance of parallel programs written using rendezvous", J. Systems and Software, pp. 125-131.

[17] X. Wu, 2003. An Approaqch to Predicting Performance for Component-based Systems, MASc thesis, Carleton University.

[18] X. Wu, M. Woodside, 2004. Performance modeling from software components," Proc. 4th Int. Workshop on Software and Performance, pp. 290-301.

[19] T. Zheng, M. Woodside, M. Litoiu, 2008. Performance model estimation and tracking using optimal filters", IEEE Trans. on Software Engineering, V 34 , no. 3 pp 391-406.