A Graphical Modeling Viewpoint on Queueing Networks

Charles Sutton
Michael Jordan

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2009-21
http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-21.html

February 2, 2009

Acknowledgement

We thank Randy Katz, George Porter, and Rodrigo Fonseca for useful conversations. This research is supported in part by gifts from Sun Microsystems, Google, Microsoft, Cisco Systems, Hewlett-Packard, IBM, Network Appliance, Oracle, Siemens AB, and VMWare, and by matching funds from the State of California's MICRO program (grants 06-152, 07-010, 06-148, 07-012, 06-146, 07-009, 06-147, 07-013, 06-149, 06-150, and 07-008), the National Science Foundation (grant #CNS-0509559), and the University of California Industry/University Cooperative Research Program (UC Discovery) grant COM07-10240.

A Graphical Modeling Viewpoint on Queueing Networks

Keywords: graphical models

Charles Sutton CASUTTON@CS.BERKELEY.EDU
Computer Science Division, University of California, Berkeley, CA 94720 USA

Michael I. Jordan JORDAN@CS.BERKELEY.EDU
Computer Science Division, University of California, Berkeley, CA 94720 USA

Abstract

Probabilistic models of the performance of computer systems have long been used to *predict* future performance. What has not been recognized, however, is that performance models can also be used to *diagnose* past performance problems. In this paper, we analyze queueing networks from the probabilistic modeling perspective, applying inference methods from graphical models that allow answering diagnostic questions from incomplete data. In particular, we present a slice sampler for networks of G/G/K queues. As an application of this technique, we localize performance problems in distributed systems from incomplete system trace data. On both synthetic networks and a benchmark distributed Web application, we identify bottlenecks with 25% of the overhead of full instrumentation.

1. Introduction

Diagnosis of performance problems in computer systems is a rich application area for machine learning, because data about system performance can be readily obtained. Many such diagnostic questions concern system performance in the face of load. For example: *"Five minutes ago, a brief spike in workload occurred. Which parts of the system were the bottleneck during that spike?"* A second type of question is diagnosis of slow requests: *"During the execution of the 1% of requests that perform poorly, which system components receive the most load?"* The bottleneck for slow requests could be very different than the bottleneck for typical requests if, for example, a storage or network resource is failing intermittently.

However, classical approaches to machine learning are not an entirely good fit: Supervised learning requires labeling failure data, a time-consuming task that may need to be performed anew for each application to be diagnosed. On the other hand, a fully unsupervised approach fails to exploit the known structure of the system. An appealing alternative is a model-based approach, in which we design a performance model that can be learned directly from measurements of system performance and that incorporates the structure of the system as an inductive bias. A class of performance model that is particularly well studied is queueing models. Queueing models predict the explosion in system latency under high workload in a way that is often reasonable for real systems, allowing the model to extrapolate from performance under low load to performance under high load. Queueing theory has been studied for over a hundred years, but the theory concerns approximating future behavior of the system, not inference about past system behavior or learning from incomplete data.

The main contribution of this paper is a new family of analysis techniques for queueing models, based on a statistical viewpoint. We collect a training set by sampling a small set of arrival and departure times from the system, treating the times that were not sampled as missing data. The issue of missing data is unavoidable in real systems, either because full instrumentation is too expensive, or because the true bottlenecks in the system are unknown. To learn the model parameters, we sample from the posterior distribution over missing data and parameters in a Bayesian fashion, using approximate inference algorithms for graphical models. Essentially, we view a queueing network as a

structured probabilistic model, a novel viewpoint that combines queueing networks and graphical models.

Specifically, we develop a slice sampler for networks of G/G/K queues (Section 3). Algorithmically, the sampler is significantly more complex for queueing networks than for standard graphical models, because the local conditional distribution over a single departure can have many singularities (Section 3.1), and because the Markov blanket for a single departure can be arbitrarily large (Section 3.2). On both synthetic data (Section 4.1) and data from a benchmark Web application (Section 4.2), we demonstrate the ability to find bottlenecks with 25% of the overhead of full instrumentation.

Despite the long history of queueing models, we are unaware of any existing work that treats them as latent-variable probabilistic models, and attempts to approximate the posterior distribution directly. Furthermore, we are unaware of any technique for estimating the parameters of a queueing model from an incomplete sample of arrivals and departures.

2. Modeling

In this section, we describe queueing networks from a novel viewpoint, as a structured probabilistic model over arrivals and departures to a system.

2.1. Single Queues

In this paper, we consider two types of queues: G/G/K/FCFS and G/G/1/RSS queues.[1] To illustrate our viewpoint, however, we first describe a special case of the G/G/K/FCFS queue, the G/G/1/FCFS queue. A G/G/1/FCFS queue is a system that can process one request at time and has a queue to hold incoming requests. Each request e for $e \in [1, N]$ arrives at the system at a time a_e, where each *interarrival time* $\delta_e := a_e - a_{e-1}$ is drawn iid according to a density g. For example, if g is an exponential density, then the arrival times are drawn from a Poisson process. Requests are removed from the queue in a first-come first-served (FCFS) manner. The amount of time a request spends in the queue is called the *waiting time* w_e. Once the request leaves the queue, it begins processing, and remains in service for some *service time* s_e. The service times are drawn independently from a distribution with density f. The total *response time* is defined as $r_e := s_e + w_e$.

In this way, the model decomposes the total response time of a job into two components: the waiting time, which represents the effect of system load, and the service time, which is independent of system load. From this perspective, an attraction of queueing models is that they specify the distribution over response times as a function of the distributions over arrival and service times.

We can view the generative process for a G/G/1/FCFS queue, conditioned on the total number N of jobs that are ever processed, as follows. First generate interarrival times $\delta_e \sim f$ independently for $e \in [1, N]$. Then generate service times $s_e \sim g$ independently. Finally compute the arrival and departure times as

$$\begin{aligned} a_e &= a_{e-1} + \delta_e \\ d_e &= s_e + \max[a_e, d_{e-1}] \end{aligned} \tag{1}$$

Notice that if we consider only interarrival and service times, then all the variables are iid, but if we consider the arrival and departure times, complex dependencies arise. For example, certain combinations of arrival and departure times are impossible. In particular, a G/G/1/FCFS makes strong assumptions about the behavior of the system. The strongest is an order assumption, that the arrival order is the same as the departure order. This assumption seldom holds in computer systems. To relax this assumption, we consider two more complex queueing models: the G/G/1/RSS queue and the G/G/K/FCFS queue.

First, a G/G/1/RSS queue is like the FCFS queue, except that when the processor finishes a job, a new job is chosen randomly from all jobs currently in queue. As before, interarrival and service times are generated from f and g independently, but computing the resulting arrival and departure times is more complicated. To write the likelihood for this model, define $\gamma(e)$ as the predecessor of job e in the departure order of the queue and IQD_e as the set of jobs in queue when e departs. Both these variables and the departure times can be computed from the interarrival and service times by the system of equations

$$\begin{aligned} IQD_e &= \{e' \mid a_{e'} < d_e \text{ and } d_e < d_{e'}\} \\ \gamma^{-1}(e) &= \begin{cases} \text{Random}(IQD_e) & \text{if } IQD_e \neq \emptyset \\ \arg\min_{\{e' \mid d_e < a_{e'}\}} a_{e'} & \text{otherwise} \end{cases}, \\ u_e &= \max[a_e, d_{\gamma(e)}] \\ d_e &= s_e + u_e \end{aligned} \tag{2}$$

where Random(S) indicates an element of the set S, chosen uniformly at random, and u_e denotes the *commencement time*, that is, the time that e enters service.

[1] In this standard notation for queues, the first G means that the interarrival time follows a general (G) distribution, the second G refers to the service time, the 1 indicates that there is a single server, and FCFS indicates that jobs are removed from the queue in a FCFS basis.

It can be shown that $\gamma(e)$ is always the event immediately preceding e in the departure order of the queue.

The order assumptions made by this queueing discipline are far less strict than those of the G/G/1/FCFS queue, but still some combinations of arrivals and departures are infeasible. The main constraint is that whenever a job e arrives at a nonempty queue, at least one other job must depart before e can enter service.

To write the likelihood for this model, we need to incorporate both the service density and the random selection of jobs from the queue. For the latter purpose, define N_e to be the number of jobs in queue when e enters service, that is,

$$N_e = 1 + \#\{e' \mid a_{e'} < a_e \text{ and } u_e < d_{e'}\}. \tag{3}$$

Then the likelihood is

$$p(A) = \prod_e N_e^{-1} g(a_e - a_{e-1}) f(d_e - u_e) \tag{4}$$

Second, a G/G/K/FCFS queue has K processors, meaning that it can serve K jobs concurrently, and any additional jobs must wait in queue. So instead of a job entering service when the previous job departs, rather a job enters service when all but $K-1$ of the previous jobs have departed. To compute the departure times, we introduce auxiliary variables p_e to indicate which of the K servers has been assigned to job e, the time b_{ek} to indicate the first time after job e arrives that the server k would be clear, and c_e to indicate the first time after e arrives that any of the K servers are clear. Given the interarrival and service times, all other variables can be computed deterministically using the system of equations

$$\begin{aligned} b_{ek} &= \max\{d_{e'} \mid a_{e'} < a_e \text{ and } p_{e'} = k\} \\ c_e &= \min_{k \in [0,K)} b_{ek} \\ p_e &= \arg\min_{k \in [0,K)} b_{ek} \\ u_e &= \max[a_e, c_e] \\ d_e &= s_e + u_e \end{aligned} \tag{5}$$

and the resulting likelihood is

$$p(A) = \prod_{e \in A} g(a_e - a_{e-1}) f(d_e - u_e) \tag{6}$$

2.2. Networks of Queues

Many systems are naturally modeled as a network of queues. For example, web services are often designed in a "three tier" architecture (Figure 1), in which the first tier is a presentation layer that generates the response HTML, the second tier performs application-specific logic, and the third tier handles persistent storage, often using a relational database. Each tier is replicated on redundant servers, so it is natural to use one queue for each server on each tier and one queue for the network connection. As another example, a distributed storage system might use one queue for each storage server, and one queue for each disk.

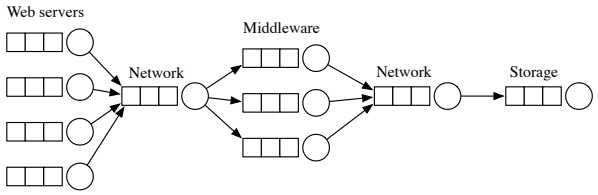


Figure 1. A queueing network model of a three-tier web service. The circles indicate servers, and the boxes indicate queues.

We model the process of a task through the system as a probabilistic finite state machine, where each state σ emits a new queue for the task according to a distribution $p(q|\sigma)$, and the transition distribution between states is $p(\sigma'|\sigma)$. We expect that the system FSM is defined in advance, for example, from a known protocol or multi-tier network.

A series of tasks can be represented compactly by the notion of an *event*. An event represents the process of a task arriving at a queue, waiting in queue, receiving service, and departing. Each state transition corresponds to an event $e = (k_e, \sigma_e, q_e, a_e, d_e)$, where k_e is the task that changed state, σ_e is the new state, q_e the new queue, a_e the arrival time, and d_e the departure time. Every event e has two predecessors: a within-queue predecessor $\rho(e)$, which is generated by the previous task to arrive at queue q_e, and a within-task predecessor $\pi(e)$, which is generated by the task's previous arrival. Arrivals to the system as a whole are represented using special *initial events*, which arrive at a designated initial queue q_0 at time 0 and depart at the time that the task entered the system.

Putting this together, the joint density for a set of events is $p(E) = \prod_{e \in E} f(d_e - u_e)\, p(q_e|\sigma_e)\, p(\sigma_e|\sigma_{\pi(e)})$.

3. Inference

In this section, we tackle the challenging task of developing a slice sampler for networks of G/G/K queues. Given a complete set of arrivals and departures to the system, Section 2 described how the service and waiting times can be computed, allowing answers to many

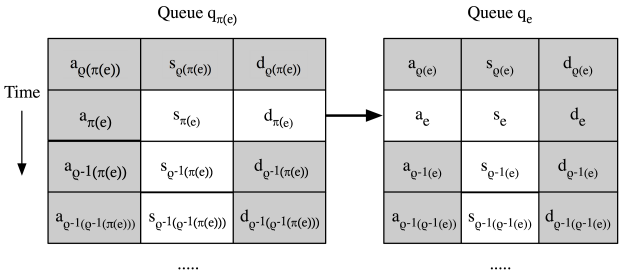


Figure 2. Variables used in the Gibbs sampler

diagnostic questions. However, complete data is not always available, either because the performance cost of instrumenting every request is unacceptable, or because the resources that cause the queueing effects may be unknown to the system developers.

Therefore, our concern is answering diagnostic questions from incomplete data. We assume that the observed training set contains all arrival, departure, and FSM path information for a randomly-selected subset of tasks. Also, whenever a task is observed, we also measure the total number of tasks, both observed and hidden, that have entered the system. (This information is trivial to collect in actual systems.) The inference task is to compute the posterior $p(E|O)$, where E is a full set of events, as defined in the last section, and $O \subset E$ is the observed subset. The posterior is intractable even for the simplest queueing models, so we sample from it using MCMC.

Designing a sampler for this situation is difficult because of several issues that are specific to queueing models. One issue is that because of the deterministic dependencies between arrivals and departures described in Section 2, the sampler must be constructed carefully to ensure the resulting samples of arrivals and departures are feasible. Indeed, it is challenging to simply find a good point at which to initialize the sampler (for details, see Section 3.5).

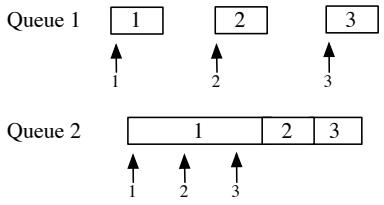
Two other issues directly influence the choice of sampling algorithm. First, the conditional distribution over a single departure contains singularities at the times when other events arrive and depart, as described in Section 3.1. For this reason, it is difficult to design a proposal function that is suitable for importance sampling, rejection sampling, or Metropolis-Hastings. A second difficulty, described in Section 3.2, is that the Markov blanket for a single departure can be arbitrarily large, because changing one departure time d_e can affect arbitrarily many later events, by changing the time those events enter service. Nevertheless, the expected number of later events that are altered is often reasonable, so inference is still possible. However, implementing a naive Gibbs sampler is difficult because sampling from the single-departure conditional distribution requires computing its normalizing constant. But the normalizing constant must be computed numerically, which can be expensive because of its many singularities.

To address these difficulties, we use a Gibbs sampler within a slice sampling framework (Neal, 2003). This has the advantage that we need only compute the local conditional distribution up to a constant, not sample from it. Specifically, the local conditional distribution is $p(a_e|E_{\setminus e})$, where a_e is the arrival of one of the unobserved events $e \in E \setminus O$. Because of the deterministic dependencies mentioned above, when resampling a_e, we must also resample $d_{\pi(e)}$, and the service times $s_{e'}$ for all later jobs e' in q_e. The notation $E_{\setminus e}$ means all of the information from E, except for those variables. These variables are illustrated in Figure 2.

To compute the distribution $p(a_e|E_{\setminus e})$, essentially we need to compute the list of events whose service time would be affected if a single departure time changed. This amounts to solving the system of equations in either (2) or (5). We describe how to do this for G/G/K/FCFS queues in Section 3.3 and for G/G/1/RSS queues in Section 3.4.

3.1. Difficulties in Proposal Functions

A simpler alternative to computing the exact conditional distribution for the Gibbs sampler is to sample from the single-variable conditional distribution using either an importance sampler or a rejection sampler. But designing a good proposal is difficult for even the simplest queueing models, because the shape of the conditional distribution varies with the arrival rate. To see this, consider two independent M/M/1/FCFS queues, each with three arrivals, as shown below:



Here the horizontal axis represents time, the vertical arrows indicate when jobs arrive at the system, and each box represents the interval between when a job enters service and when it finishes, that is, the service time. The notation M/M/1 means that the interarrival distribution is exponential with rate λ, and the service distribution is exponential with rate μ.

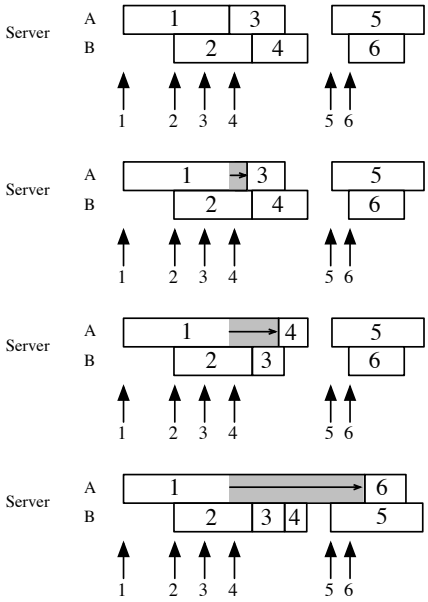


Figure 3. A departure with a large Markov blanket.

For each of these two queues, suppose that we wish to resample the arrival time of job 2, conditioned on the state of the rest of the system. At top, the queue is lightly loaded ($\lambda \ll \mu$), so the dominant component of the response time is the service time. Therefore, the distribution $a_2 = d_2 - \text{Exp}(\mu)$ is an excellent proposal for an importance sampler. (It is inexact because the shape of the distribution changes in the area $a_2 < d_1$.) However, this proposal would be extremely poor for the heavily loaded case at bottom, because there the true conditional distribution is $\text{Unif}[a_1; a_3]$. A better proposal would be flat until the previous job departs and then decay exponentially. But this is precisely the behavior of the exact conditional distribution, so we derive that instead.

3.2. Difficulties in Markov Blankets

In this section, we describe an example that illustrates how the Markov blanket of a single departure is difficult to characterize. Consider the single G/G/2/FCFS queue shown in Figure 3, in which we wish to resample the departure d_1 to a new value d_1', holding all departures constant. Thus, as d_1 changes, so will the service times of jobs 3–6.

Three different choices for d_1' are illustrated in panels 2–4 of Figure 3. First, suppose that d_1' falls within the range (d_1, a_4) (second panel). This has the effect of shortening the service time s_3 without affecting any other jobs. If instead d_1' falls in (d_2, d_4) (third panel), then both jobs 3 and 4 are affected: job 3 moves to server B, changing its service time; and job 4 enters service immediately after job 1 leaves. Third, if d_1' falls even later, in (a_6, d_6) (fourth panel), then both jobs 3 and 4 move to server B, changing their service times, job 5 switches processors but is otherwise unaffected, and now job 6 can start only when job 1 leaves.

Finally, notice that it is impossible for d_1' to occur later than d_6 if all other departures are held constant. This is because job 6 cannot depart until all but one of the earlier jobs depart, that is, $d_6 \geq \min[d_1', d_5]$. So since $d_5 > d_6$, it must be that $d_6 \geq d_1'$.

Algorithm 1 Update dependent events for a departure change in G/G/K/FCFS queue.

function FORWARDPROPAGATE(e_0)
// Input: e_0, event whose departure has changed
stabilized $\leftarrow 0$
$e \leftarrow \rho^{-1}(e_0)$
while $e \neq$ NULL and not *stabilized* **do**
  $b_{ek} \leftarrow b_{\rho(e),k} \quad \forall k \in [0, K)$
  $b_{e,k(\rho(e))} \leftarrow d_{\rho(e)}$
  stabilized $\leftarrow 1$ **if** $b_e =$ old value of b_e **else** 0
  $c_e \leftarrow \min_{k \in [0,K]} b_{ek}$
  $p_e \leftarrow \arg\min_{k \in [0,K]} b_{ek}$
  $s_e \leftarrow d_e - \max[a_e, c_e]$
  $e \leftarrow \rho^{-1}(e)$
end while

3.3. G/G/K/FCFS Queues

In this section, we describe how to compute the conditional likelihood for a G/G/K/FCFS queue. Suppose we wish to resample the arrival time a_e of an event e; equivalently, this is the departure $d_{\pi(e)}$ of the previous-task event $\pi(e)$. As explained previously, computing $p(a_e | E_{\setminus e})$ directly, or sampling from it, is difficult. Instead, for the slice sampler it is sufficient to compute the joint density $p(a_e, E_{\setminus e})$, which is proportional to the conditional density.

Algorithmically, this amounts to setting $d_{\pi(e)}$ and a_e to the new value, and propagating the change through the system of equations (5), thereby obtaining new values of $c_{e'}$, $p_{e'}$, and $s_{e'}$ for all other events e'. The procedure for doing this is specified in Algorithm 1 for the departure $d_{\pi(e)}$ and in Algorithm 2 for the arrival a_e. The main idea in both algorithms is that any service time $s_{e'}$ depends on its previous events only through the processor-clear times $b_{\rho(e')}$ of the immediately previous event $\rho(e)$. Furthermore, b_e can be

Algorithm 2 Update dependent events for an arrival change in G/G/K/FCFS queue.

// Input: e_0, event whose arrival has changed
// Input: aOld, old arrival of event e_0
Update arrival order ρ for changed arrival of e_0
$aMin \leftarrow \min[a_{e_0}, aOld]$
$aMax \leftarrow \max[a_{e_0}, aOld]$
$E \leftarrow$ all events that arrive at q_{e_0} in $aMin \ldots aMax$
// First change events that arrive near e_0
for all $e \in E$ **do**
  $b_{ek} \leftarrow b_{\rho(e),k} \qquad \forall k \in [0, K)$
  $b_{e,k(\rho(e))} \leftarrow d_{\rho(e)}$
  $c_e \leftarrow \min_{k\in[0,K]} b_{ek}$
  $p_e \leftarrow \arg\min_{k\in[0,K]} b_{ek}$
  $s_e \leftarrow d_e - \max[a_e, c_e]$
end for
// Second, propagate changes to later events
$e \leftarrow \rho^{-1}(\textsc{lastElement}(E))$
stabilized $\leftarrow 1$ **if** $b_e =$ old value of b_e **else** 0
if not *stabilized* **then**
  ForwardPropagate(e) *// Algorithm 1*
end if

Algorithm 3 Update dependent events for a departure change in a G/G/1/RSS queue.

Update departure order γ for changed departure d_e
newPrev, *newNext* $\leftarrow$ Events departing immediately before and after the time d_e^{old}
oldPrev, *oldNext* $\leftarrow$ Events departing immediately before and after the time d_e
$dMin \leftarrow \min[d_{newPrev}, d_{oldPrev}]$
$dMax \leftarrow \max[d_{newNext}, d_{oldNext}]$
$L \leftarrow$ all events with departures in $dMin \ldots dMax$
for all $e \in L$ **do**
  $u_e \leftarrow \max[a_e, d_{\gamma(e)}]$
  $s_e \leftarrow d_e - u_e$
end for

computed recursively as

$$b_{ek} = \begin{cases} d_{\rho(e)} & \text{if } k = p_{\rho(e)} \\ b_{\rho(e),k} & \text{otherwise} \end{cases}. \tag{7}$$

After running Algorithms 1 and 2, in principle we can compute the new joint likelihood $p(a_e, E_{\setminus e})$ directly from (6). The problem with this naive approach is that it requires $O(|E|)$ time to update each event departure, so that each iteration of the Gibbs sampler uses $O(|E|^2)$ time. This quadratic computational cost is unacceptable for the large numbers of events that can be generated by a real system. To avoid this cost, we use a lazy updating scheme, in which first we generate the list of events Δ that would be changed by running Algorithms 1 and 2. Then we can compactly compute the new log-likelihood as

$$\ell_{\text{new}} = \ell_{\text{old}} + \sum_{e\in\Delta} \log f(s_e^{\text{new}}) - \log f(s_e^{\text{old}}) \tag{8}$$

If any s_e^{new} is negative, set $\ell_{\text{new}} = -\infty$, which will cause the slice sampler to try a different value for a_e.

3.4. G/G/1/RSS Queues

In this section, we describe computing the slice sampler for a G/G/1/RSS queue. As in the FCFS case, we compute the joint density instead of the conditional, by incrementally computing the change to the joint density resulting from a single departure change. Algorithm 3 describes how to update the dependent events in queue $q_{\pi(e)}$ in response to a change in $d_{\pi(e)}$. For the arrival a_e, none of the service times in q_e need to be updated.

There are two other issues that need to be considered. First, the new value $a_e = d_{\pi(e)}$ must still be feasible with respect to the constraints (2). This can be ensured by computing the new departure order γ for $q_{\pi(e)}$, and then verifying for all events in q_e and $q_{\pi(e)}$ that $\gamma^{-1}(e) \in IQD_e$ (or that the departure of e empties the queue, and $\gamma^{-1}(e)$ is the next event to arrive).

The second issue is that computing the joint density $p(a_e, E_{\setminus e})$ from (4) is complicated by the factors N_e^{-1}. These arise from the random selection of job e to enter service, out of the N_e jobs that could have been selected. These factors are crucial to the likelihood, because they are the only penalty on a job waiting in queue for a long time; without them, the sampled waiting times would become arbitrarily large.

To compute these, we need an efficient data structure for computing N_e, the number of jobs that were in queue when the event e entered service. This is implemented by two sorted lists for each queue: one that contains all of the queue's arrival times, and one that contains all of the departure times. From these, we can use binary search to compute the total number of jobs that have arrived before u_e (call that $\#A_e$) and the total number of jobs that have departed before u_e (call that $\#D_e$). Then we can compute $N_e = \#A_e - \#D_e$.

Then the new log-likelihood can be computed as

$$\ell_{\text{new}} = \ell_{\text{old}} + \sum_{e\in\Delta} (\log f(s_e^{\text{new}}) - \log N_e^{\text{new}}) - (\log f(s_e^{\text{old}}) - \log N_e^{\text{old}})$$

Here Δ must include all events e' whose commencement time falls in $a_e^{\text{old}} \ldots a_e^{\text{new}}$, because those events will have a new value of $N_{e'}$.

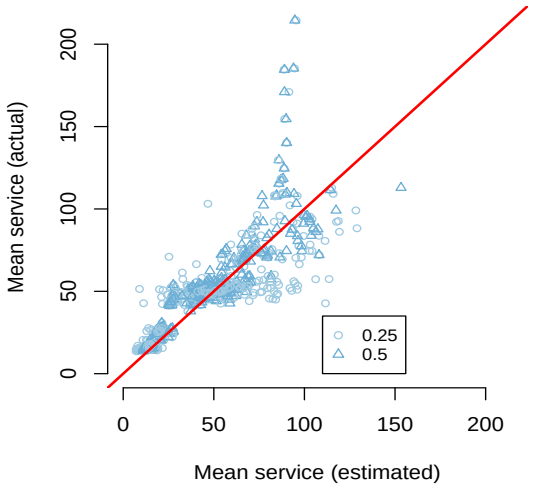


Figure 4. Performance of the sampler on synthetic data. Each point represents the estimated and actual mean service time of single queue in one of the networks.

3.5. Initialization

A final issue is how the sampler is initialized. This is challenging because not all sets of arrivals and departures are feasible: they must obey both the single-queue constraints in (2) or (5)—neither of which are convex—and also the network constraint $d_{\pi(e)} = a_e$. In addition to being feasible, the configuration should also be suitable for mixing: setting all latent interarrival and service times to 0 results in a feasible configuration, but one that makes mixing difficult. Or, if the service distribution belongs to a scale family (such as gamma or log-normal), initializing all of the service times to be identical causes the initial variance to be sampled to a very small value, which is also very bad for mixing.

Initialization proceeds as follows. For each unobserved task, we sample a path of states and queues from the FSM, and service times from an exponential distribution initialized from the mean of the observed response times. Sometimes the sampled service time will conflict with the observed arrivals and departures. In this case we use rejection, and if no valid service time can be found, we set the service time to zero. Finally, we run a few Gibbs steps with exponential service distributions, before switching to the actual service distributions in the model. This prevents zero service times, which would cause zero-likelihood problems with some service distributions (namely, the log normal).

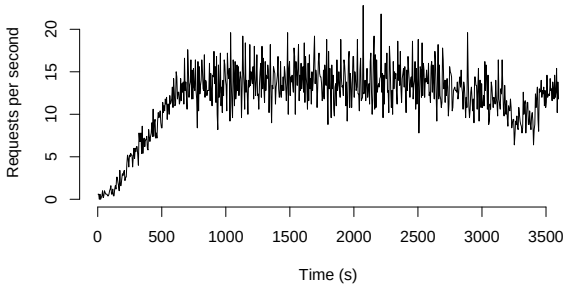


Figure 6. Workload used for the Cloudstone application

4. Results

4.1. Synthetic Data

To evaluate the sampler, we generate data from a variety of three-tier queueing networks, similar to Figure 1, but without the network queues. The networks vary in their numbers of queues (9, 30); queue types (G/G/K/FCFS and G/G/1/FCFS); and service distributions, which are chosen so that the expected utilization of each queue varies between 0.5 and 0.9. In all cases, the service distributions are exponential. In all, 72 different networks are used. For each network, 1000 tasks are generated, resulting in 4000 sampled events. To evaluate the ability of the sampler to reconstruct the service time of each queue from incomplete data, we observe the arrivals and departures for either 25% or 50% of the tasks, and record the estimated mean service for each queue. The sampler does not use the true parameters of the service distribution. Rather, the network parameters are sampled in a Bayesian fashion, using uninformative priors.

Figure 4 shows the estimated mean service time as a function of the actual mean of the entire sample. Each point represents a single queue in a single network. The estimated service times are well correlated with the true values ($\rho = 0.90517$ for the G/G/K/FCFS queues, and $\rho = 0.86367$ for the G/G/1/RSS).

4.2. Web Application

In this section, we demonstrate how this inferential framework can be used to diagnose performance problems on a benchmark Web application. We use Cloudstone (Sobel et al., 2008), a recently-proposed benchmark that is designed to model Web 2.0 applications. Cloudstone has been implemented on several platforms for Web development. The version that we use is implemented in Ruby on Rails, which is a popular application framework used by several high-profile commercial applications, including Basecamp and Twitter. Cloudstone was developed by professional Web developers with the intention of reflecting common idioms

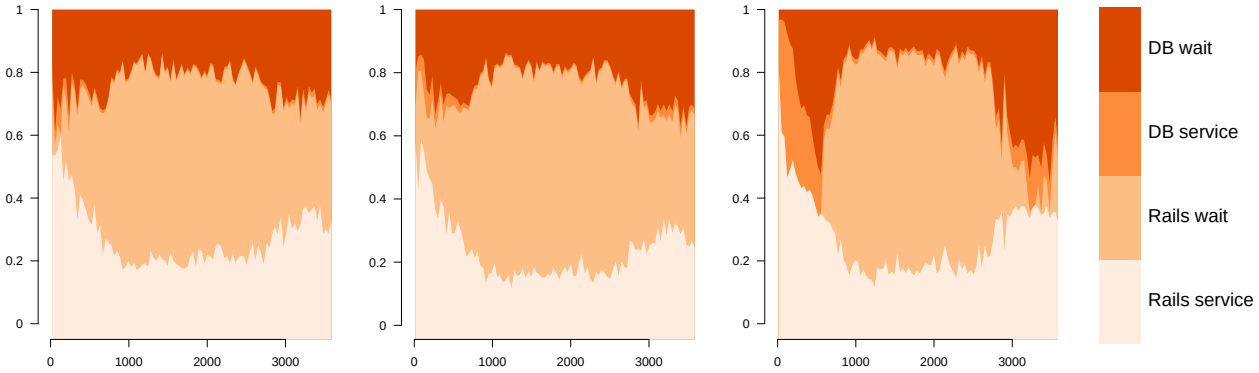


Figure 5. Reconstruction of the percentage of request time spent in each tier, from 25% tasks observed (left), 50% tasks observed (center), and all tasks observed (right). The x-axis is the time in seconds that the task entered the system, and the y-axis the estimated percentage of response time.

of Rails usage in actual applications.

We run a series of 42,936 requests to Cloudstone in a one-hour period, using the workload generator supplied as part of the benchmark. The number of requests sent per second is shown in Figure 6; this is derived from a production workload supplied by Ebates.com, but scaled down to the capacity of Cloudstone. The application is run on Amazon's EC2 utility computing service, with Rails running in parallel on 5 virtual machines, each running two threads, and a single VM running a MySQL database. For each request, we record which of the Rails instances handled the request, the amount of time spent in Rails, and the amount of time spent in the database. Each Cloudstone request causes many database queries; the time we record is the sum of the time for those queries. (We also record the amount of time spent on the network, but for this workload the network time is insignificant.)

We model the system as a network of G/G/1/RSS queues: one for each Rails process (10 queues in all) and one queue for the database. This choice is motivated by the architecture of Rails, because each Rails instance processes exactly one request at a time, but I/O is performed in parallel. The service distributions are mixtures of four exponential distributions (the number of components was chosen using AIC). Interestingly, G/G/K/FCFS queues are an extremely poor fit to this data. A total of 128,808 events (in the sense defined in Section 2.2) are caused by the 42,936 requests.

Our goal is to infer the system bottleneck, that is, what component contributes most to the system response time. Although we measure directly how much time is spent in Rails and how much in the database, this does not indicate how much time is due to intrinsic processing and how much is due to workload. This distinction is important in practice: If system latency is due to workload, then we expect adding more servers to help, but not if latency is due to intrinsic processing. Therefore, the goal is to infer the expected percentage of time a request spends in Rails waiting and service, and what percentage in database and service. These proportions change depending on the workload, so as the workload changes over time, the estimated proportions should change as well. Furthermore, we wish to infer the proportions from as little data as possible, to minimize the overhead of logging on the Rails machines, on which latency is critical.

Figure 5 displays the proportion of time per-tier spent in processing and in queue, as estimated from 25%, 50%, and 100% of the total amount of data. Qualitatively, the proportions estimated from only 25% of the data strongly resemble those one the full data set: In either case, it is apparent that the Rails waiting time dominates all other components, and that DB waiting time dominates DB service time.

5. Discussion

Queueing models have been long studied in telecommunications, operations research, and performance modeling of computer networks and systems. Queueing theory focuses on analytic approximations to the long-run behavior of the system—such as the steady-state distribution or large-deviations bounds—but does not consider the problem of inferring system behavior from incomplete data. In the systems community, there has been recent interest in modeling dynamic Web services by queueing networks (Urgaonkar et al., 2005; Welsh, 2002). There is also recent work using queueing models to initialize more flexible models,

namely regression trees (Thereska & Ganger, 2008).

References

Neal, R. M. (2003). Slice sampling. *The Annals of Statistics*, *31*, 705–741.

Sobel, W., Subramanyam, S., Sucharitakul, A., Nguyen, J., Wong, H., Patil, S., Fox, A., & Patterson, D. (2008). Cloudstone: Multi-platform, multi-language benchmark and measurement tools for Web 2.0.

Thereska, E., & Ganger, G. R. (2008). Ironmodel: Robust performance models in the wild. *SIGMETRICS*.

Urgaonkar, B., Pacifici, G., Shenoy, P., Spreitzer, M., & Tantawi., A. (2005). An analytical model for multi-tier internet services and its applications. *SIGMETRICS*.

Welsh, M. (2002). *An architecture for highly concurrent, well-conditioned internet services*. Doctoral dissertation, University of California, Berkeley.