Inference and Learning in Networks of Queues

Charles Sutton
School of Informatics
University of Edinburgh
csutton@inf.ed.ac.uk

Michael I. Jordan
Computer Science Division
University of California, Berkeley
jordan@eecs.berkeley.edu

Abstract

Probabilistic models of the performance of computer systems are useful both for predicting system performance in new conditions, and for diagnosing past performance problems. The most popular performance models are networks of queues. However, no current methods exist for parameter estimation or inference in networks of queues with missing data. In this paper, we present a novel viewpoint that combines queueing networks and graphical models, allowing Markov chain Monte Carlo to be applied. We demonstrate the effectiveness of our sampler on real-world data from a benchmark Web application.

1 Introduction

Modern Web services, such as Google, eBay, and Facebook, run on clusters of thousands of machines that serve hundreds of millions of users per day. 1 The number of simultaneous requests, called the workload, is a primary factor in the system’s response time. There are two important types of questions about system performance. The first involve prediction, e.g., “How would the system perform if the number of users doubled?” Crucially, this requires extrapolating from performance on the current workload to that on a higher workload. The second type of question involves diagnosis, e.g., determining what component of the system caused a past performance problem, and whether that component was simply overloaded with requests, or whether it requires fundamental redesign.

We view the modeling of the performance of computer systems as an important case study for machine learning. Performance modeling is essentially a regression problem—predicting response time from workload—but standard regression techniques fall short, because they extrapolate poorly. Richer models have the potential to extrapolate by exploiting prior knowledge about the system, but it is difficult to incorporate detailed prior knowledge about a specific large-scale computer system. We wish to incorporate general prior knowledge about computer systems that is likely to be useful for extrapolation.

Networks of queues provide a natural general framework for developing prior models of computer systems. These are classical models that have a 50-year history of application to systems. They have two key advantages. First, they extrapolate naturally, predicting an explosion in system latency under high workload. Second, the structure of the network can reflect our prior knowledge about the system.

However, in modern computer systems the issue of missing data is unavoidable, because exhaustive instrumentation of the system state can be prohibitively expensive. To our knowledge, no previous work in queueing networks concerns parameter estimation or diagnosis with missing data.

In this paper, we present a new family of analysis techniques for networks of queues, inspired by a graphical modeling perspective. We view the queueing network as a structured probabilistic model over the times that requests arrive and depart from the system, and consider the case in which some of those are unobserved. We present a Markov chain Monte Carlo (MCMC) sampler for the missing arrival and departure times, allowing the parameters to be estimated using Bayesian inference. Essentially, this viewpoint combines queueing networks and graphical models.

Despite the naturalness of a graphical approach to queueing networks, and despite the 50-year history of queueing models, we are unaware of any previous

work that treats networks of queues within a graphical modeling framework. Perhaps the reason for this is that algorithmically, inference is far more complex for queueing networks than for standard graphical models. This has two causes: first, the local conditional distribution over a single departure can have many singularities (Section 3.1), and second, the Markov blanket for a departure can be arbitrarily large (Section 3.2).

On a benchmark Web application, we demonstrate that queueing networks extrapolate better than standard regression models (Section 6), and that the sampler can effectively reconstruct missing data, even when 75% of the data are unobserved (Section 7).

2 Modeling

Many computer systems are naturally modeled as networks of queues. For example, Web services are often designed in a multitier architecture (e.g., Figure 2) in which the stateless first tier computes the HTML response, and makes requests to a second tier that handles long-term storage, often using a database. In order to handle high request rates, each tier contains multiple machines that are functionally equivalent.

It is natural to model a distributed system by a network of queues, in which one queue models each machine in the system. Each queue controls access to a processor, which is an abstraction that models the machine’s aggregate CPU, memory, I/O, etc. If the processor is busy with a request, other requests must wait in the queue. The waiting time, which is the amount of time spent in queue, models the amount of total processing time that was caused by workload. The service time, which is the amount of time the request ties up the processor, models the amount of time that is intrinsically required for the request to be processed.

The queues are connected to reflect the system’s structure. For example, a model for a Web service might be: Each request is assigned to a random queue in the first tier, waits if necessary, is served, and repeats this process at the second tier, after which a response is returned to the user.

Each external request to the system thus might involve many requests to individual queues in the network. We will say that a job is a request to an individual queue, and a task is a series of jobs that are caused by a single external request to the system. For example, in a Web service, a task would represent the entire process of the system serving an external request. Each task would comprise two jobs, one for each tier.

In order to define a probabilistic model over the arrival and departure times of each job, we need to model both (a) which queues are selected to process each job in a task and (b) the processing that occurs at each individual queue. For (a), we model the sequence of queues traversed by a task as a first-order Markov chain. A task completes when this Markov chain reaches a designated final state, so that the number of jobs in each task is potentially random.

Second, to model the processing at individual queues, we consider several different possibilities. To describe them, we must first define some terminology that is common to all queue types. Each job has an arrival time, which is the time the job entered the queue. The waiting time of the job in queue is denoted by \( w_e \). Finally, once the job is selected for processing, it samples its service time \( s_e \). The service times are mutually independent. Once the service time has elapsed, the job completes. The time at which this happens is called the departure time \( d_e \).

A number of additional variables concern the progress of tasks through the network. For any job \( e \), we denote the queue that serves the job as \( q_e \). Every job has two predecessors: a within-queue predecessor \( \pi(e) \), which is the previous job (from some other task) to arrive at \( q_e \), and a within-task predecessor \( \pi(e) \), which is the previous job from the same task as \( e \). We always have \( d_{\pi(e)} = a_e \), meaning that every job arrives immediately after its within-task predecessor departs.\(^2\)

Finally, to simplify notation, arrivals to the system as a whole are represented using special initial jobs, which arrive at a designated initial queue \( q_0 \) at time 0 and depart at the time that the task enters the system. In this representation, we never need to consider arrival times explicitly, because every job’s arrival time is either 0 (for initial jobs), or else equal to the departure time of the previous job in the task.

In the remainder of this section, we describe three kinds of queues: first-come first-served queues (FCFS), queues which employ random selection for service (RSS), and processor sharing queues (PS).

2.1 Multiprocessor FCFS queues

The first type of queue that we consider is called a \( K \)-processor first-come first-served (FCFS) queue. This type of queue can process \( K \) requests simultaneously, that is, the queue is connected to \( K \) processors. When all \( K \) processors are busy, any additional requests must wait in queue. Once a processor becomes available, the waiting request that arrived first is selected for service.

The generative process for the model over departure times is as follows. For all jobs \( e \) that arrive at the

\(^2\)If we wish to model delays caused by a computer network, we can add queues that explicitly represent the computer network.
queue, we will assume that the arrival times \( a_e \) have been generated by some other queue in the network. So we describe a generative process over service, waiting, and departure times.

First, at the beginning of time, sample a service time \( s_e \) for each job \( e \) from some density \( f \). Then, with the arrival times and service times for all jobs in hand, the waiting times and departure times can be computed deterministically. To do this, 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. Then the departure times \( d_e \) can be computed using the system of equations

\[
\begin{align*}
    b_{ek} &= \max\{d_{e'} \mid a_{e'} < a_e \text{ and } p_{e'} = k\} \\
p_e &= \arg \min_{k \in [0,K]} b_{ek} \quad c_e = \min_{k \in [0,K]} b_{ek} \\
d_e &= s_e + u_e \quad u_e = \max[a_e, c_e].
\end{align*}
\]

(1)

2.2 Processor-sharing (PS) queues

A markedly different model is the processor-sharing (PS) queue (Kleinrock, 1973). A PS queue models a system that handles multiple jobs simultaneously via time sharing. To understand this queue, imagine the system in discrete time, with each time slice having a duration \( \Delta t > 0 \). When a job \( e \) arrives at the queue, it samples a service time \( s_e \). Then, at each time slice \( t \), all of the \( N(t) \) jobs in the system have their service times reduced by \( \Delta t/N(t) \). Once the remaining service time of a job drops below zero, it leaves the queue. Finally, to get the PS queue, take \( \Delta t \to 0 \).

The PS queue can be expressed as a generative model similarly to the FCFS queue, except with a different system of equations to convert from service times to departure times. This is

\[
    s_e = \int_{a_e}^{d_e} \frac{1}{N(t)} \, dt, \quad N(t) = \sum_{e=1}^{N} 1_{\{a_e < t\}} 1_{\{t < d_e\}}.
\]

(2)

The first equation is a statement of the procedure just described for determining a departure time in a PS queue. (The integral arises from taking the limit \( \Delta t \to 0 \).) The second equation is the definition of \( N(t) \).

2.3 Random selection for service

In a queue that employs random selection for service, instead of the earliest job being selected from the queue for service, a job is selected randomly among all those that are waiting. This type of queue can be handled in a similar manner to FCFS queues, except that the analogous equations to (1) are more complex (details omitted).

2.4 Summary

We now summarize the generative process for a network of queues. First, for every task, sample a path of jobs and queues from a Markov chain. Second, set the arrival times for all initial jobs to 0. Third, sample each service time \( s_e \) independently from the service density for \( q_e \). Finally, compute the departure times by solving the system of equations: (a) for every queue in the network, the equations in (1) or (2), as appropriate (the queues need not all be the same type), and (b) for all non-initial jobs, \( d_{\pi(e)} = a_e \). We will call this system of equations the departure time equations.

This is the entire description of the generative model underlying a network of queues. The key insight here is to view the queueing network as a deterministic transformation from service times to departure times, via the departure time equations. The distinction between service times and departure times is important statistically, because while the service times are all iid, the departure times have complex dependencies.

The network can be interpreted as a graphical model if the order in which jobs arrive and depart at each queue is known. For example, for FCFS queues the model contains nodes for the service time, arrival time, departure time, and auxiliary variables for each job. The parents of each node are the other variables in the departure time equation that concern it. If the arrival and departure orders are unknown, then the model cannot be usefully described by a graph (Section 3.2).

Finally, we derive the joint density over the vector \( \mathbf{d} = \{d_e\} \). Let \( \mathbf{s}(\mathbf{d}) \) be the function defined by solving the departure time equations for \( \mathbf{s} \) with fixed \( \mathbf{d} \), and analogously \( s_e(\mathbf{d}) \) for a single job \( e \). To derive the density, we need the Jacobian of the function \( \mathbf{s} \). It can be shown that the Jacobian is a product \( \prod_e J(q_e, d_e) \), where \( J(q_e, d_e) = 1 \) for FCFS queues, but \( J(q_e, d_e) = N(d_e)^{-1} \) for PS queues. The joint is

\[
    p(\mathbf{d}) = \prod_e J(q_e, d_e) f(s_e(\mathbf{d})).
\]

(3)

3 Inferential Problem

In this section, we describe the inferential setting. First we explain the nature of the observations. If the departure and path information for every job were observed, then the service times could be computed deterministically by reversing the departure time equations, and parameter estimation is straightforward.

In computer systems, however, complete data is not generally available. In a system that receives millions
of requests per day, the overhead required to record
detailed data about every request can be unacceptable; thus, in practice it is common to reduce the data
by subsampling. To model this process, we assume
that whenever a task arrives at the system, it is cho-

se normal Bayesian approach to inference. Let 

\[ E = \{ (q_e, a_e, d_e) \} \]

\( q_e \) be the arrival, departure, and queue in-

formation for all jobs in the system, and \( O \subset E \)
be the information for the subset of tasks that are observed, and \( U = E - O \). Two posterior distributions are of
interest. First, if \( \theta \) is the vector of parameters for
the service distributions, we will be interested in the pos-
terior \( p(\theta | O) \) over model parameters. Second, we are
also interested in the posterior \( p(w_e | O) \) over the wait-
ing times \( w_e \) of individual jobs, because this can be
interpreted as our posterior belief over how much of
the response time was caused by workload.

In the rest of this section, we explain several compi-
lations in queueing models that make sampling from
the posterior difficult, and which provide strong design
constraints on the inference algorithm.

### 3.1 Difficulties in Proposal Functions

A natural idea is to sample from the posterior distri-

bution over the missing data using either an importance
sampler, a rejection sampler, or Metropolis-Hastings.
But designing a good proposal is difficult for even the
simplest queueing models, because the shape of the con-
tinued data about every request can be unacceptable; thus, in practice it is common to reduce the data
by subsampling. To model this process, we assume
that whenever a task arrives at the system, it is cho-

se normal Bayesian approach to inference. Let 

\[ E = \{ (q_e, a_e, d_e) \} \]

\( q_e \) be the arrival, departure, and queue in-

formation for all jobs in the system, and \( O \subset E \)
be the information for the subset of tasks that are observed, and \( U = E - O \). Two posterior distributions are of
interest. First, if \( \theta \) is the vector of parameters for
the service distributions, we will be interested in the pos-
terior \( p(\theta | O) \) over model parameters. Second, we are
also interested in the posterior \( p(w_e | O) \) over the wait-
ing times \( w_e \) of individual jobs, because this can be
interpreted as our posterior belief over how much of
the response time was caused by workload.

In the rest of this section, we explain several compi-
lations in queueing models that make sampling from
the posterior difficult, and which provide strong design
constraints on the inference algorithm.

### 3.2 Difficulties Caused by Long-Range
Dependences

In this section, we describe another difficulty in queue-
ing models: the unobserved arrival and departure
times can have complex dependencies. Modifying one
departure time can force modification of service times
of many later jobs, if all other arrival and departure
times are kept constant. In other words, the Markov
blanket of a single departure can be arbitrarily large.

This can be illustrated by a simple example. Con-
sider the two-processor FCFS queue shown in Figure 1.
Panel 1 depicts the initial state of the sampler, from
which we wish to resample the departure \( d_1 \) to a new
value \( d'_1 \), holding all departures constant, as we would
in a Gibbs sampler, for example. Thus, as \( d_1 \) changes,
so will the service times of jobs 3–6.

Three different choices for \( d'_1 \) are illustrated in pan-
els 2–4 of Figure 1. First, suppose that \( d'_1 \) falls within
\((d_1, d_2)\) (second panel). Then, this shortens 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 ser-
tvice 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 proces-
sors; and job 6 can start only when job 1 leaves. Fi-
nally, 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 service times for a departure change in K-processor FCFS queue

1: function UpdateForDeparture(e₀)
2: // Input: e₀, job with changed departure
3: stabilized ← 0
4: e ← \( ρ^{-1}(e₀) \)
5: while e ≠ NULL and not stabilized do
6: \( b_k \leftarrow b_{ρ(e)}, k \forall k ∈ [0, K] \)
7: \( b_{k, ρ(e)} \leftarrow d_{ρ(e)} \)
8: stabilized ← 1 if \( b_k = \) old value of \( b_k \) else 0
9: \( c_e \leftarrow \min_{k ∈ [0, K]} b_k \)
10: \( p_e \leftarrow \arg\min_{k ∈ [0, K]} b_k \)
11: \( s_e \leftarrow d_{0} - \max[a_e, c_e] \)
12: \( e \leftarrow ρ^{-1}(e) \)

Algorithm 2 Update dependent service times for an arrival change in K-processor FCFS queue

1: function UpdateForArrival(e₀, aOld)
2: // Input: e₀, job with changed arrival
3: // Input: aOld, old arrival of job e₀
4: // Update arrival order \( ρ \) due to e₀
5: aMin ← \( \min[e₀, aOld] \)
6: aMax ← \( \max[e₀, aOld] \)
7: \( E \) ← all jobs arriving within \( aMin \ldots aMax \)
8: // First change jobs that arrive near e₀
9: for all \( e ∈ E \) do
10: \( b_k \leftarrow b_{ρ(e)}, k \forall k ∈ [0, K] \)
11: \( b_{k, ρ(e)} \leftarrow d_{ρ(e)} \)
12: \( c_e \leftarrow \min_{k ∈ [0, K]} b_k \)
13: \( p_e \leftarrow \arg\min_{k ∈ [0, K]} b_k \)
14: \( s_e \leftarrow d_{0} - \max[a_e, c_e] \)
15: // Second, propagate changes to later jobs
16: \( e \leftarrow ρ^{-1}\left(\text{lastElement}(E)\right) \)
17: stabilized ← 1 if \( b_k = \) old value of \( b_k \) else 0
18: if not stabilized then
19: UpdateForDeparture(e₀)

Algorithm 3 Update dependent service times for an arrival or a departure change in a PS queue.

1: function RelevantJobs(e, aOld, dOld)
2: // Compute set of jobs that are affected by change to the job \( e \)
3: // Input: e, job with changed arrival or departure
4: // Input: aOld, dOld, old arrival and departure times of \( e \)
5: \( a ← \min[a_e, aOld] \)
6: \( d ← \max[d_e, dOld] \)
7: return \( \{e′ | (a_e, d_e) \text{ intersects } (a, d)\} \)

1: function UpdateJobs(e, aOld, dOld)
2: // Update dependent jobs for an arrival or a departure change to the job \( e \)
3: // Input: e, job with changed arrival or departure
4: // Input: aOld, dOld, old arrival and departure times of \( e \)
5: Recompute \( N(t) \) for new arrival and departure times of \( e \)
6: \( Δ \leftarrow \text{RevelantJobs}(e, aOld, dOld) \)
7: for all \( e′ ∈ Δ \) do
8: \( s_{e′} \leftarrow \int_{a_{e'}}^{d_{e′}} \frac{1}{N(t)} \, dt \)

4 Sampling

In this section, we describe the details of the sampler. We focus on sampling from the posterior \( p(U|O) \) over the departure times for unobserved jobs. Once that sampler is in place, adding a Gibbs step for the parameters \( θ \) is straightforward. Exact sampling from the posterior is infeasible even for the simplest queueing models, so instead we sample approximately using Markov chain Monte Carlo (MCMC).

We use the slice sampler of Neal (2003). Slice sampling is an MCMC method that, for our purposes, can be viewed as a “black box” for sampling from univariate continuous densities. Its key advantage is that it requires the ability only to compute the unnormalized conditional density, not to sample from it or to compute its normalizing constant. At each iteration, for each unobserved departure time \( d_e \), we want to sample from the one-dimensional density \( p(d_e | E_{e}) \), where \( E_{e} \) means all of the information from \( E \), except for \( d_e \) and \( d_{ρ^{-1}(e)} \) (which must be equal). But since slice sampler only requires the density up to a constant, it is sufficient to compute the joint \( p(d_e, E_{e}) = p(E) \), which is much simpler. In the following sections, we describe how to compute the joint density.

4.1 Overview

The joint density, given in (3), is a product over all jobs. Computing this product naively would require \( O(N) \) time to update each job, so that each iteration of the Gibbs sampler would require \( O(N^2) \) time. This cost is unacceptable for the large numbers of jobs processed by a real system. Fortunately, this cost can be circumvented using a lazy updating scheme, in which first we generate the set of relevant jobs \( Δ \) that would be changed if the new value of \( d_e \) were to be adopted. Then the new density is computed incrementally by updating the factors in (3) only for the relevant jobs. Notice that a job is relevant either if its service time changes, or if its Jacobian term changes.

Therefore, computing the unnormalized density requires computing the jobs whose service time would be affected by a change to a single departure. This amounts to setting \( d_e \) and \( a_{ρ^{-1}(e)} \) to the new value and propagating these two changes through the departure time equations, yielding new service time estimates for all other jobs in the queues \( q_e \) and \( q_{ρ^{-1}(e)} \).

So each type of queue requires two algorithms: a propagation algorithm that computes the modified set of service times that results from a new value \( d_e = a_{ρ^{-1}(e)} \), and a relevant job set algorithm that computes the set of jobs \( Δ \) whose factor in (3) has changed. Next
we describe these algorithms for FCFS queues (Section 4.2) and for PS queues (Section 4.3). RSS queues are omitted for lack of space.

4.2 FCFS Queues

The propagation algorithms for the FCFS queue are given in Algorithm 1 (for the departure times) and Algorithm 2 (for the arrival times). These algorithms compute new values of $b_{e'k}$, $u_{e'}$, $c_{e'}$, $p_{e'}$, and $s_{e'}$ for all other jobs $e'$ and processors $k$ for all other jobs in the queue. The main idea is that any service time $s_{e'}$ depends on its previous jobs only through the processor-clear times $b_{\rho'(k)}k$ of the immediately previous job $\rho(e)$. Furthermore, each $b_{ck}$ can be computed recursively as $b_{ck} = d_{\rho(e)}$ if $k = \rho(e)$ and $b_{ck} = b_{\rho(e),k}$ otherwise.

The relevant job set for the FCFS queue is simply the set of jobs whose service times were updated by Algorithms 1 and 2.

4.3 PS Queues

The propagation algorithm for the PS queue is given in Algorithm 3. The same algorithm is used for arrival and departure changes. For this algorithm to be efficient for large $N$, a special data structure is needed to store the step function $N(t)$, the number of jobs in the queue at time $t$. By using two sorted lists, one for arrival times and one for departure times, $N(t)$ can be computed efficiently by binary search.

The relevant job set algorithm is RELEVANTJOBS in Algorithm 3. It requires a data structure for interval intersection. Our implementation uses a treap.

5 Experimental Setup

Cloudstone (Sobel et al., 2008) is an application that has been recently proposed as an experimental setup for academic research on Web 2.0 applications, such as Facebook and MySpace. The application was developed by professional Web developers with the intention of reflecting common programming idioms that are used in actual applications. For example, the version of Cloudstone that we use is implemented in Ruby on Rails, a popular software library for Web applications that has been used by several high-profile commercial sites, including Basecamp and Twitter.

The architecture of the system (Figure 2) follows common real-world practice. The apache Web server is used to serve static content such as images. Dynamic content is served by Ruby on Rails. In order to handle a large volume of requests, multiple copies of Rails are run on separate machines; each is indicated by a “Rails” box in Figure 2. Because the Rails servers are stateless, they access data on a shared database running on a separate machine.

In our setup, we run 10 copies of Rails on 5 machines, two copies per machine. We run the apache server, the load balancer, and the database each on their own machine, so that the system involves 8 machines in all.

We run a series of 2663 requests to Cloudstone over 450s, using the workload generator included with the benchmark. A total of 7989 jobs are caused by the 2663 requests. The workload is increased steadily over the period, ranging from 1.6 requests/second at the beginning to 11.2 requests/second at the end. The application is run on Amazon’s EC2 utility computing service. 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.

6 Prediction

In this section we demonstrate that networks of queues can effectively extrapolate from the performance of the system at low workload to the worse performance that
The model to oscillate wildly outside the training data (for example, to make negative predictions). When the queueing models perform poorly, it is typically because the model underestimates the capacity of the system, so that the predicted response time explodes at a lower workload than the actual response time.

7 Diagnosis

In this section, we demonstrate that our sampler can effectively reconstruct the missing arrival and departure data. The task is to determine from the Cloudstone data which component of the system contributes most to the system's total response time, and how much of the response time of that component is due to workload. Although we measure directly how much time is spent in Rails and in the database, the measurements do not indicate how much of that 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. Furthermore, we wish to log departure times from as few tasks as possible, to minimize the logging overhead on the Rails machines.

We model the system as a network of PS queues (Figure 2): one for each Rails server (10 queues in all) and one for the database. The latency caused by apache, by the load balancer, and by the network that connects all of the component is minimal, so we do not model it. The service distributions are exponential.

Figure 3 displays the proportion of time per-tier spent in processing and in queue, as estimated using the slice sampler from 25%, 50%, and 100% of the full data. Visually, the reconstruction from only 25% of the data strongly resembles the full data: it is apparent that as the workload increases from left to right, the Rails servers are only lightly loaded, and the increase in response time is due to workload. Although we measure directly how much time is spent in Rails and in the database, the measurements do not indicate how much of that 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. Furthermore, we wish to log departure times from as few tasks as possible, to minimize the logging overhead on the Rails machines.

To obtain a quantitative measure of error, we group time into 50 equal-sized bins, and compute the mean service time for each bin and each tier of the system. We report the root mean squared error (RMSE) between the reconstructed service times and the service times that would have been inferred had the full data been available. We perform reconstruction on ten different random subsets of 25% and 50% of the jobs. We use two baselines: (a) one that always predicts that the response time is composed only of the service time (denoted “Wait = 0”) and (b) a linear regression of the per-job waiting time onto the workload in the last 500 ms. Results are reported in Table 2.

The MCMC sampler performs significantly better at
reconstruction than the baselines, achieving a 25% reduction in error for 25% data observed, and a 54% reduction in error for 50% data observed. Linear regression performs poorly on this task, performing worse than the trivial “Wait=0” baseline. Interestingly, the performance of linear regression, unlike the queueing network, does not improve with additional data. This supports the idea that the poor performance of linear regression is due to limitations in the model.

8 Discussion

In the computer systems community, there has been recent interest in modeling Web services by queueing networks (Urgaonkar et al., 2005; Welsh, 2002). Previous work in statistics (Heggland and Frigessi, 2004; Fearnhead, 2004; Jones, 1999) has considered parameter estimation in single-processor FCFS queues, but does not consider more complex queueing models, or networks of queues. The single-processor FCFS queue simplifies the situation considerably from the more general models considered here. In practice the more sophisticated models are essential for accurately modeling real systems, as shown in Section 6.

Another related research area is network tomography (Castro et al., 2004), which concerns problems such as estimating link delays in a network solely from measurements of the end-to-end delay. This is a markedly different inferential problem from ours, in that the network tomography literature does not focus on how much of the link delay is caused by workload. For this reason, in our setting the observed data always includes the number of requests in the system, a measurement that is usually assumed to be unavailable in the network tomography setup.

Finally, the current work can be viewed as a coarse-grained generative model of computer performance. More detailed models could be of significant interest.

Acknowledgements

This research is supported in part by gifts from Sun Microsystems, Google, Microsoft, Amazon Web Services, Cisco Systems, Cloudera, eBay, Facebook, Fujitsu Labs America, Hewlett-Packard, Intel, Network Appliance, SAP, VMWare and Yahoo! and by matching funds from the University of California Industry/University Cooperative Research Program grant COM07-10240.

References


