SD: a Divergence-based Estimation Method for Service Demands in Cloud Systems

Salvatore Dipietro
Department of Computing
Imperial College London
United Kingdom
s.dipietro@imperial.ac.uk

Giuliano Casale
Department of Computing
Imperial College London
United Kingdom
g.casale@imperial.ac.uk

***Abstract*—Estimating performance models parameters of cloud systems presents several challenges due to the distributed nature of the applications, the chains of interactions of requests with architectural nodes, and the parallelism and coordination mechanisms implemented within these systems.**

In this work, we present a new inference algorithm for model parameters, called *state divergence* (SD) algorithm, to accurately estimate resource demands in a complex cloud application. Differently from existing approaches, SD attempts to minimize the divergence between observed and modeled marginal state probabilities for individual nodes within an application, therefore requiring the availability of probabilistic measures from both the system and the underpinning model.

Validation against a case study using the Apache Cassandra NoSQL database and random experiments show that SD can accurately predict demands and improve system behavior modeling and prediction.

***Index Terms*—Service demands, inference, queueing, cloud, NoSQL database**

I. Introduction

Over the last decade, models to predict performance behavior of cloud systems have been increasingly used to drive automated resource management [1]. A common problem arising in performance model estimation is the accurate inference of *service demands*, which are the average computational requirements placed by different types of requests within a system. Existing estimation algorithms obtain demands from steady-state metrics such as throughput, response times, and resource utilization. However, existing techniques can have erratic performance and it is difficult to identify a method that is best in all cases [2]. The most commonly used demand estimation methods accept in input *mean* performance metrics, thus not fully exploiting during the inference process of a large part of the information available within a measured dataset.

In this work, we instead propose a demand estimation method that requires to collect *marginal state probabilities* for the number of requests processed by individual nodes of a cloud system. While probabilistic methods have appeared in recent work [3], our proposal differs as we wish to carry out a more advanced probabilistic analysis that takes into account the execution workflow of a request, in order to more accurately estimate demand at various stages of operation. We assume that each job is tagged with a class of service at a given time. Workflows are then described in terms of a sequence of *class-switching* steps for jobs that visits one or more resources in the system. That is, upon moving from a node to another, a job can switch its class so to represent a different phase of execution.

Class-switching has received a limited degree of attention in prior work on demand estimation, possibly with the exception of [4] which considers it in the context of a single multi-server resource. However, workflows arise commonly in distributed systems composed by multiple resources, which is the case we consider in this work. For example, a job that visits multiple times the same node, requesting different execution requirements at each visit, may be modelled by assuming that the job switches class in-between visits. Distributed NoSQL databases such as Apache Cassandra provide an example, in which class-switching can be used to express a workflow of execution through multiple nodes in order to retrieve the data needed by a query for its completion [5].

Despite their practical importance, class-switching models can be difficult to deal with due to state-space explosion, which is more rapid than in standard multiclass models. In order to increase their tractability, we consider class-switching in the context of product-form queueing network theory [6], which allows one to exactly transform a model with class-switching into a multiclass model without class-switching [7], [8]. Despite this equivalence result, we notice that in the presence of load-dependence there appear to be no exact formulas available to perform this mapping for computing marginal state probabilities, a gap that we also overcome in this work.

Leveraging the proposed methods to obtain marginal probabilities in the presence of class-switching, we propose a new demand estimation approach, based on information-theoretic divergence measures, called *State Divergence* (SD) estimation. SD seeks to minimize the divergence between marginal state probabilities and their corresponding empirical estimates, in order to produce accurate estimates for the demands. Because marginals capture each job class, without the aggregation implicit in the transformations from class-switching models to multiclass models without class-switching, it can explicitly capture the actual mix of requests in execution at a resource

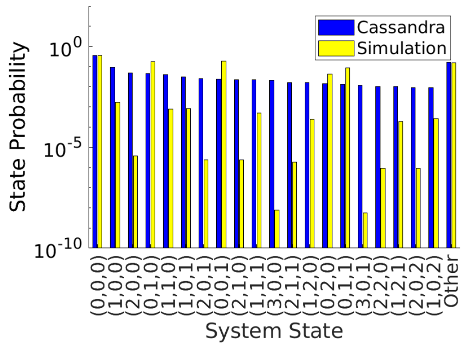


Fig. 1: Marginal probability difference between Cassandra and Simulation.

more accurately than using the aggregated model. The rationale for this method is that, by minimizing state divergence, one hopes to obtain model parameters that accurately capture the stationary behaviour of the system, as reflected by occupancy in a fraction of its observable states, more accurately than by looking only at its output performance metrics, such as throughput, response times, or CPU utilization.

We validate the SD estimation algorithm through several randomly generated models and with a case study conducted using the Apache Cassandra NoSQL database [9]. Our results demonstrate that SD can significantly reduce errors in performance prediction compared to state-of-the-art algorithms which do not explicitly account for class-switching. The obtained demands are able to match the system state while reproducing a more realistic behavior in the model compared to state-of-the-art estimation algorithms.

The rest of the paper is organized as follows: in Section II and III we give a motivating example and review popular demand estimation algorithms, respectively. Section IV describes the reference model and some novel probabilistic formulas we have developed to analyze systems with class-switching. The SD algorithm is then developed in Section V, where we also present several possible divergence measures that can be used with our method. In Section VI and Section VII, we give experimental results on Apache Cassandra and random model instances. Finally, Section VIII, gives conclusions and outlines future work.

II. Motivation Example

Over the years, several inference algorithms have been developed to parameterize performance models of cloud systems [2]. For the purpose of this section, we focus our attention on the complete information (CI) algorithm [10]. We consider this algorithm estimates the demand taking, as input, samples of the arrival rate and execution time of the system request. This algorithm because supports multi-threading, multi-class and class-switching.

To better understand and illustrate the limitation of an existing method such as CI when the class-switch is in use, we want to analyse the difference between the marginal state probabilities of a real system and the performance model parameterized using the service demands obtained by CI. For the purpose of this experiment, we use a Cassandra cluster composed of three nodes, deployed on Microsoft Azure. As workload generator, we run YCSB [11] with 10 clients that continuously perform read queries to the database with a consistency level *ONE*.

Figure 1 compares the two marginal state probabilities. The horizontal axis shows the system state of a particular Cassandra node, given as the number of requests of three classes in that node. On the vertical axis, we show the probability that the system is in the corresponding state. The two models have very similar mean performance metrics such as throughput, response time and CPU utilization with the maximum error across the three metrics below 12%. Except for the first state, the other are significantly different. To understand the similarity of these two probabilities, we calculate their divergence using the Kullback-Leibler measure. This measure quantifies similarity between two probabilities distributions, see Section V-B. Closer the returned value is to 0, the more similar the two probabilities are. However, if the value is above 1, the algorithm suggests that the two probabilities are not very similar. In this case, the algorithm returns a value of 2.41, suggesting that even if the predicted performance metrics are close to the real system ones, the underpinning state dynamics in the performance model does not need to resemble the real system dynamics. Our goal in the rest of the paper is to propose a method that penalizes the estimation of demands that result in high divergence scores between predicted and measured state probabilities.

III. Related Work

Demand estimation is one of the most challenging steps for performance model parametrization. Existing techniques are based mostly on the statistical inference of indirect measurements such as throughput, response time and resource utilization.

Linear regression is one of the most popular statistical methods for the service demand inference. The regression method applied to service demand estimation has been presented, for the first time, by Bard and Shatzoff in 1978 to characterise the resource consumption of some specific functions of an operating system [12]. Then, the method has been extended in [13] to estimate the CPU demand for a single-thread application using multiple classes. However, this method can fail or produce wrong results if the observation time is too small, with outliers or if the variance across the samples is limited [13] .

Machine learning algorithms have also been exploited in demand estimation. One of the first and most used technique is based on the Kalman filter [2], [14]. Another machine learning techniques for demand inference includes clustering [15], which starting from observation data composed by timestamps, throughput and utilization, can recognize deviations over time of demands, such as those resulting from hardware upgrades.

Notation	Description
M	Number of nodes in the model.
K	Number of queueing stations.
R	Number of job classes.
C	Number of job chains.
N_r	Number of jobs in class r, $N = \sum_r N_r$
θ_{kr}	Service demand of class r at node k.
σ_r	Think time of class-r, $\sigma_r = \sum_{k=K+1}^{M} \Theta_{kr}$
c_i	Number of servers at node i
s_{ir}	Service time at node i for jobs in class r
e_{ir}	Visit ratio at station i for jobs in class r

TABLE I: Summary of main notation for the model input parameters

Recently, [16] propose the QMLE algorithm, a technique based on the maximum likelihood estimation, that uses mean queue length values, rather than response times, to perform demand estimation.

Demand estimation methods based on optimization have also been investigated. They usually require in input response time or CPU utilization datasets [17]. Optimization is then used to estimate the demands in [18] with a load-dependent model using quadratic programming techniques.

Despite their extensive validation on parameterization of models with isolated classes, none of the above methods has to our knowledge been validated in the presence of class-switching. Moreover, differently from the other works presented here, the minimization problem we consider here looks at the divergence between the probability state distribution of the real system and the one considered from the model. Mean performance metrics are included in constraints, but not as part of the objective function.

IV. Reference model

We focus on distributed systems that may be described by a network of M resources, cyclically visited by jobs belonging to R job classes. Under specific assumptions given by the BCMP theorem, these models admit, up to a normalising constant, a closed-form solution for their equilibrium distribution, from which marginal probability expressions can be explicitly derived. Model parameters for BCMP networks are listed in Table I.

Let $\pi(\boldsymbol{n})$ denote the equilibrium probability for state $\boldsymbol{n} = (\boldsymbol{n}_1, \ldots, \boldsymbol{n}_R)$ in a BCMP queueing network, where $\boldsymbol{n}_i = (n_{i1}, \ldots, n_{iR})$ is the set of jobs of the R classes running at station i, and we require $\sum_{\boldsymbol{n}\in\boldsymbol{S}} \pi(\boldsymbol{n}) = 1$, where $\boldsymbol{S} = \{\boldsymbol{n} \in \mathbb{N}^{MR} | n_{kr} \geq 0, \sum_{k=1}^{M} n_{kr} = N_r\}$ is the model state space.

We focus here on closed models where the scheduling policy for the target station is processor sharing (PS), although others may be considered with similar expressions [6]. We assume that PS node i has $c_i \geq 1$ servers, such that when the node has up to c_i running jobs they run without contention. Note that infinite server nodes may be seen as a special case of multi-server nodes where $c_i = +\infty$. In this case, the product-form formula for the steady-state probability of this model is given by

$$\pi(\boldsymbol{n}) = \frac{1}{G_{\boldsymbol{\Theta}}(\boldsymbol{N})} \prod_{k=1}^{M} n_k! \prod_{r=1}^{R} \frac{\theta_{kr}^{n_{kr}}}{n_{kr}! \cdot \prod_{u=1}^{n_k} \min(u, c_k)} \qquad \boldsymbol{n} \in \boldsymbol{S} \tag{1}$$

where the factorials capture the ordering of jobs within the PS node and the product of $\min(u, c_i)$ functions describes load-dependence due to the multi-server nodes. Assuming that the number of server c_i is known for a given model, in the SD approach we see the service demand as the problem of assigning the values of the demands θ_{kr} so that (1) is as close as possible to the empirically-observed system states, for some subset of states.

From (1), the normalizing constant may be determined by requiring that state probabilities sum to one, which leads to the explicit formula

$$G_{\boldsymbol{\Theta}}(\boldsymbol{N}) = \sum_{\boldsymbol{n}\in\boldsymbol{S}} \prod_{i=1}^{K} n_i! \prod_{k=1}^{M} \prod_{r=1}^{R} \frac{\theta_{kr}^{n_{kr}}}{n_{kr}! \cdot \prod_{u=1}^{n_k} \min(u, c_i)} \tag{2}$$

However, computing $G_{\boldsymbol{\Theta}}(N)$ using direct summation over the state space $\boldsymbol{S}$ is usually infeasible unless the model is small, because the number of states generated by the model grows as $\mathcal{O}(N^{MR})$, where N is the total number of jobs in the model. To solve this problem, several approaches have been proposed using exact [19] or approximate methods [20] when $c_i = 1$.

A. Class-switching models

The BCMP theorem permits within the described class of models also to allow class-switching, whereby jobs departing from a node in class r can arrive in the destination node in class $s \neq r$. However, under class-switching the number of jobs in each class is no longer constant over time, implying that the underpinning state space is much larger than in regular multiclass networks without class switching. To address this issue, under class-switching the original R job classes can be partitioned into $C \leq R$ disjoint *chains* [7] , defined by the strongly connected components of the class-switching probability matrix. In other words, chains are defined such that a job within chain c can become over time any of the classes in that chain, but cannot become of a class belonging to any other chain $h \neq c$.

To do so, we first need to calculate the number of visits (e_{ir}) to each station and class of the original network and then compute the corresponding number of visits for each chain q at every node i in the aggregate model ($\hat{e}_{iq}$) as

$$\hat{e}_{iq} = \frac{\sum_{c\in C_q} e_{ic}}{\sum_{c\in C_q} e_{1c}} \tag{3}$$

where we assume that station $i = 1$ is the reference station for the computation of the system throughput for each class.

Thanks to this transformation, the class-switching model can now be solved as a standard multiclass model with job populations equal to the chain population. However, due to this transformation, the information regarding the service time of the different classes at a station is lost by the aggregation. Let q now represent the class in the new model associated to the original chain. The service demand for class q at station i needs to be computed as [8]

$$\hat{\theta}_{iq} = \sum_{r \in C_q} \alpha_{ir} \theta_{ir} \tag{4}$$

where α_i is a scale factor defined as

$$\alpha_{ir} = \frac{e_{ir}}{\sum_{c \in C_q} e_{ic}} \tag{5}$$

The aggregate demands $\hat{\theta}_{iq}$ are stored in matrix $\hat{\mathbf{\Theta}}$. The population of chain q is set to $\hat{N}_q = \sum_{c \in C_q} N_c$.

B. Marginal probabilities

In the next sections, we shall often compute the marginal probability that a set of jobs $\boldsymbol{n}_i = (n_{i1}, n_{i2}, \ldots, n_{iR})$ resides at station i. However, because the model is solved by aggregating classes into chains, the underpinning state space describes the state of station i as $\hat{\boldsymbol{n}}_i = (n_{i1}, n_{i2}, \ldots, n_{iC})$, where C is the number of chains, making it difficult to recover per-class metrics.

In [7, p. 110], the authors observe that probability of a particular mix of jobs $\boldsymbol{n} = (n_1, \ldots, n_R)$ being active in the system is the ratio of the normalizing constant of a closed model with fixed class populations $\boldsymbol{n}$ divided by the normalizing constant of the model with class-switching. Because the factors $f_i(\boldsymbol{n}_i)$ in the product-form solution can themselves be regarded as normalizing constants for degenerate models with a single station and population $\boldsymbol{n}_i$, one readily concludes that the mix probability derived in [7] can be used to establish the per-class population at a node in the aggregated model. This leads to the expression

$$\pi(\boldsymbol{n}_i) = \frac{n_i!}{\prod_{u=1}^{n_i} \min(u, c_i)} \prod_{r=1}^{R} \frac{\theta_{ir}^{n_{ir}}}{n_{ir}!} \frac{G_{\hat{\mathbf{\Theta}}}^{-i}(\hat{\boldsymbol{N}} - \hat{\boldsymbol{n}}_i)}{G_{\hat{\mathbf{\Theta}}}(\hat{\boldsymbol{N}})} \tag{6}$$

where $G_{\hat{\mathbf{\Theta}}}^{-i}$ is the normalizing constant for the subnetwork composed by all stations except station i, and $\hat{\boldsymbol{n}}_i$ is obtained from $\boldsymbol{n}_i$ by summing classes that belong to the same chain. Note that both $G_{\hat{\mathbf{\Theta}}}$ and $G_{\hat{\mathbf{\Theta}}}^{-i}$ are defined for the aggregate model, and thus are computed using the aggregate population vector $\hat{\boldsymbol{N}}$ and the matrix of the aggregate demands $\hat{\mathbf{\Theta}}$.

To the best of our knowledge, (6) has not appeared before in the literature. This expression paves the way to define a method for estimating service demands in class-switching models, which is presented in the next section.

V. Estimation Algorithm

In this section, we describe our algorithm for demand estimation. Differently from other algorithms that look into aggregated performance metrics such as system throughput, system response time or CPU utilization (as described in Section III), the SD algorithm minimizes the divergence between the state probability distribution of the real system from the one generated by the model under test. The algorithm is defined in the following section and then we present some divergence measures that can be used to compare the two state probability distributions.

A. SD Algorithm

SD requires a performance model, provided by the user, of the system under test (SUT). To reduce the execution time of the method, we consider in this work only models that comply with the BCMP theorem, as described in Section IV and denote this model by $\mathcal{M} \equiv \mathcal{M}(\boldsymbol{N}, \mathbf{\Theta})$. However, the SD algorithm is in principle applicable to any other model from which the state probability distribution can be computed fast enough to apply computational optimization methods.

Let $\boldsymbol{n}^s = (\boldsymbol{n}_1^s, \ldots, \boldsymbol{n}_M^s)$ denote the s-th state sample ($s = 1, \ldots, S$) obtained from the SUT and defined such that $\boldsymbol{n}_i^s = (n_{i1}^s, \ldots, n_{iR}^s)$ is the state of station i. The principle underlying SD is to minimize the divergence between observed and predicted *marginal* state probabilities $\pi(\boldsymbol{n}_i^s)$ for the SUT, for all stations i and samples s. We denote the empirical and model-based marginal probability distributions with P and Q, respectively. That is, P consists of all the marginal probabilities $\pi(\boldsymbol{n}_i^s)$ for every node i and state sample $\boldsymbol{n}^s$ obtained from measurements on the SUT; similarly, Q consists of the corresponding marginal probabilities computed using the model $\mathcal{M}(\boldsymbol{N}, \mathbf{\Theta})$.

A conceptual difficulty arising upon optimizing state divergences is that this metric does not explicitly consider mean performance metrics, even though these are the typical metrics used once the model is fully parametrized. We propose to address this issue by constraining the divergence minimization to return predictions for mean performance metrics within a tolerance. The performance metrics considered include the system throughput (X), system response time (R) and the resource utilization (U). To differentiate the real and model-based values, we shall denote the former for example as $\widetilde{X}$ and the latter as X.

Let $D^*(P, Q) \geq 0$ be a generic non-negative f-divergence function used to compare the two probability distributions. Based on the previous considerations, we define the SD optimization problem as follows:

$$\begin{aligned} SD : \min_{\mathbf{\Theta}} \quad & D^*(P, Q) \\ \text{subject to} \quad & |\widetilde{X} - X| \leq \delta \cdot \widetilde{X} \\ & |\widetilde{R} - R| \leq \delta \cdot \widetilde{R} \\ & |\widetilde{U} - U| \leq \delta \cdot \widetilde{U} \\ & 0 \leq \theta_{ir} \leq \widetilde{R} \end{aligned} \tag{7}$$

where all the variables of the demand vector ($\mathbf{\Theta}$) have, as upper bound, the average response time measured on the real system during the experiment. On the other hand, $\delta \geq 0$ is a tolerance on the maximum relative error on the model mean performance predictions.

Different f-divergence measures can be used with this optimization problem. In the following section, we present the most popular algorithms and their properties.

B. Divergence measures

In this section, different f-divergence measures are presented, namely Bhattacharyya, Hellinger, Jensen Shannon and KullbackLeibler.

1) Bhattacharyya and Hellinger distance functions: The Bhattacharyya (BC) distance is a classical distance function [21] and it is used in several fields. It is defined as

$$D_{BC}(P||Q) = -\log BC \tag{8}$$

where BC is the Bhattacharyya coefficient that for discrete distribution is defined as

$$BC = \sum_{x \in X} \sqrt{P(x)Q(x)} \tag{9}$$

Since the sum of all the probabilities is 1, the Bhattacharyya coefficient lies between $0 \leq BC \leq 1$. Due to the logarithm, the Bhattacharyya distance D_{BC} is consequently defined between $0 \leq D_{BC} \leq \infty$. So, the Bhattacharyya measure is symmetric, unbounded and returns always positive values.

The Bhattacharyya coefficient (BC) is also used to define the Hellinger distance (HE). HE is defined as [22]

$$D_{HE}(P||Q) = \sqrt{1 - BC} \tag{10}$$

It lies between $0 \leq D_{HE} \leq 1$ and it is a linear function, differently from the Bhattacharyya that has a logarithm behaviors.

2) KullbackLeibler and Jensen Shannon divergence: It is an asymmetric function, so output of the function of P and Q is not equal to the results of Q and P.

Since we use this metric in a discrete space to measure the divergence between two marginal probabilities, here we consider only the KullbackLeibler divergence discrete formulas. It is defined as

$$D_{KL}(P||Q) = \sum_{x \in X} P(x) \cdot \log \frac{P(x)}{Q(x)} \tag{11}$$

In other words, the closer the KL divergence value is to 0 the more similar the two probability distributions are. On the other hand, if the value is greater or equal to 1, several differences between the two probability distributions are present.

The Jensen-Shannon (JS) divergence is based on the KullbackLeibler measure. It differs from the latter for being a symmetric function, bounded within a finite range. The JS divergence is defined as

$$D_{JS}(P||Q) = \frac{1}{2} D_{KL}(P||M) + \frac{1}{2} D_{KL}(Q||M) \tag{12}$$

where M is the average between the two distributions and it is defined as

$$M(P||Q) = \frac{1}{2}(P + Q) \tag{13}$$

The Jensen Shannon divergence is bounded between $0 \leq D_{JS} \leq 1$

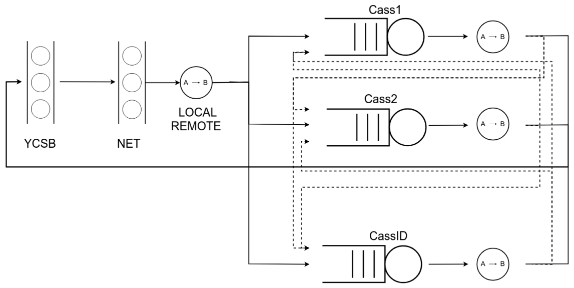


Fig. 2: Simplified Cassandra model. The model contains 2 infinite servers one representing the workload generator (YCSB) and one the network delay (NET). Each Cassandra node is represented by a queue (Cass*) and a class-switch to change the class of the requests to redirect them depending on the phase of its execution. An additional class-switch, called local-remote, it is necessary to convert part of the local requests in remote requests.

Class	Description
local-pars	Local request operations.
remote-pars	Remote request parsing operations.
remote-ID	It can represent a remote ending operations (remote-end) or data request(remote-incoming).

TABLE II: Description of the classes used for in simplified Cassandra model

VI. Evaluation

A. Cassandra Simplified Model

In this section, we illustrate the Cassandra model used for evaluating the SD algorithm. This is a simplified version of the model presented in [5]. With the aim to reduce the model complexity and the system state space, we have developed a model able to support only the Consistency Level *ONE*. We represent with a single queue each Cassandra node, as shown in Figure 2. Each Cassandra node uses the processor sharing (PS) scheduling policy. All the other stations used in [5], such as the networks and disk queues, have been grouped in a single infinite queue (or infinite server), called 'Net' positioned right after the workload generator. In addition, the workload generator has also been modelled as an infinite server.

To reduce the number of classes in the model, we aggregate together the *remote-end* and the *remote-incoming* classes in the *remote-ID* class as reported in Table II. The model differentiate the demands to use in each case based on the Cassandra *ID*. In fact, if this class is executed on the same ID node as the station, the *remote-end* operation is performed otherwise, this class represents a *remote-incoming*.

B. Experiment settings

In order to evaluate the SD method, we have collected an empirical trace from a Cassandra ring deployed on Microsoft Azure. As the validation is done for illustrative purposes, a small cluster composed by 3 Cassandra Virtual Machines (VMs) has been set up. However, the results are not expected to significantly depend on the number of nodes. Details of the testbed are reported in Table III.

TABLE III: Testbed details.

Resource	Description
Cloud provider	Microsoft Azure (IaaS).
VM type	Standard_D2s_v3.
CPU	2 vCPUs.
Memory	8 GBs.
Disk size	OS 30GB, Data 80GB.
Disk type	Premium (SSD).
Cassandra nodes	3 (version Apache Cassandra 3.11).
YCSB nodes	1 (version YCSB 0.12.0).
YCSB workload	Workload C (100% read).

We warm up the system running for 10 minutes the cluster with the predefined number of clients. Keeping the workload generator running, we start to record all the Cassandra traffic communication of the target node with the YCSB and with the other clients using *tcpdump* tool [23]. Different time lengths for sniffing traffic have been tested and we decided to use one minute since the marginal probability is stable. We develop some Python scripts to analyse the recorded traffic. Looking the source and destination of the connections, the scripts are able to detect the state of the system at each stage and so construct the marginal probability of the system for the recorded period of time.

C. Minimization algorithm settings

We have first tested the SD demand estimation approach using four different algorithms for the optimization of the underpinning non-linear program, namely Fmincon, GlobalSearch (GS), MultiStart (MS) and Genetic Algorithm (GA), which are all available in MATLAB version R2018b.

For our evaluation, we have used for all the algorithms the same objective function, non-linear constraints and bound $[1^{-10}, RT_{system}]$ for all the decisions variables (i.e., service demands), where RT_{system} is the average response time observed by the workload generator. In the case of the Fmincon algorithm, by default the starting point for all the variables are taken randomly inside the range. Since these can influence the final results, we set all them to half of the system response time value. Regarding the non-linear constraints, we limit the evaluation only to the overall system throughput to make sure that the set of testing demands has a relative error (δ) of less than 20% from the real one.

The Cassandra queueing model uses 5 random variables representing the demand due to the network delay and the demands for the four classes of the model (local, remote-pars, remote-end and remote-incoming), which are identical at all nodes. By default, the NC solver available in the LINE solver [24], [25] is used to analyze and retrieve performance and marginal probabilities of the model under test, using an implementation of the expressions in Section IV-B that we have added to the solver as part of the present paper. The two probability distributions are then compared using one of the divergence measures presented in Section V-B.

Concerning the workload generator, its demand is estimated considering the CPU time that YCSB spent into the system divided by the number of requests generated in that period of time.

D. Sensitivity analysis

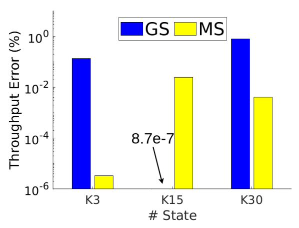
In this subsection, we performance a sensitivity analysis of our results with respect to the state space size sampled from the real system and the choice of divergence measures.

State Space: Since it is very expensive, or even intractable, to compute marginal probabilities for the complete state space, we have decided to limit the number of states to a maximum of k states. That is, we obtain marginal probabilities from the system for k states only. All the remaining states are aggregated into a single unobserved state to complete the state probability distribution. We evaluate the SD algorithm using $k = \{3, 15, 30\}$; we shall equivalently refer to these three cases as $K3$, $K15$, $K30$. The states to include into the sample space are selected based on the k value. The state selection is performed by dividing the probability distribution into three sections (high, medium, low) and take an equal number of states from each group. However, these three sections are applied to the state with the highest probability only because the marginal state probability trend decreases quite quickly.

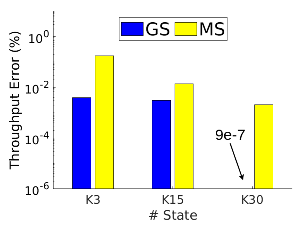
Divergence measures: In this section of the sensitivity analysis, we present the results of a set of experiments necessary to select the right combination of optimization and divergence measure able to provide stable results at the end of the SD algorithm execution. Ideally, this combination needs to provide an algorithm that it is reliable, so able to explore all the space generated by this model, and able to reduce the performance error of the algorithm increasing the state space used. We expect this behaviour since more information are used by the algorithm, more robust and accurate results should be provided since the divergence measure is performed on the marginal probability.

We start our investigation by checking which optimization algorithm performs better in this context. We take into consideration the case with 10 clients. To have a broader view, we compare these algorithms using 3 different sample space sizes ($K3, K15, K30$) and dividing the comparison of the optimization algorithms in two groups. We first compare Fmincon against the GA.

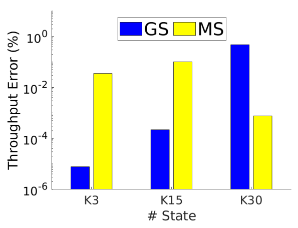
Due to space constraints, the comparison between Fmincon and GA is not presented here. However, the results show that the derivative-based algorithm outperforms GA in any of the analysed situations. In most of the cases, the difference between the two algorithms is in the order of one or more magnitudes. Since the derivative-based algorithm outperforms over the genetic one, we evaluate if more robust derivative optimization algorithms such as GS and MS are able to perform better. Figure 3 shows that these algorithms reduce further the throughput error and, in few cases, the errors are negligible. Considering now the divergence measure we notice that the HE measure using these two optimization algorithms presents the property that we are looking for the SD algorithm. In fact, increasing the searching space, it reduces the error gradually. Moreover, when we use GS with HE and $K30$ as state space, the algorithm reaches a negligible throughput error.



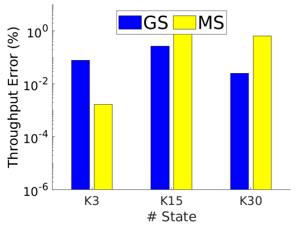
(a) BC



(b) HE

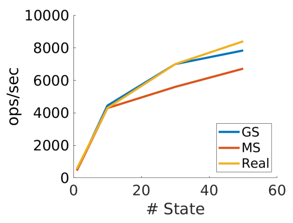


(c) JS

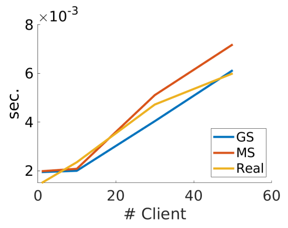


(d) KL

Fig. 3: Divergence comparison with 10 clients and using Largest strategy with GS and MS.



(a) K30 - Throughput



(b) K30 - Response Time

Fig. 4: Predicted throughput and response time performance using $K = 30$.

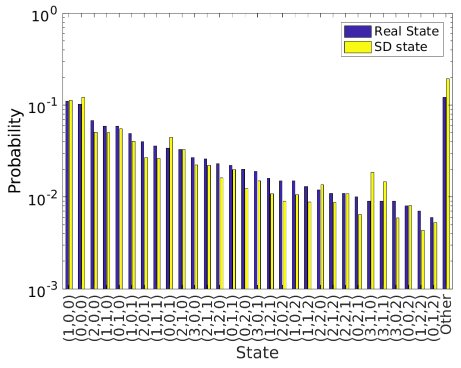


Fig. 5: State probability distribution for the model with $K30$ state and 10 clients.

For these reasons, we have decided to consider HE as the main divergence measure. Differently, as optimization algorithm, both the GS and MS can be used. However, in some cases, we still considering the Fmincon optimization algorithm because it is significantly faster than the GS or MS.

E. Cassandra Demand Estimation

We started to test the SD algorithm against different state space sizes and we notice that, as the size increases, the model prediction accuracy increases as well, which is expected. Figure 4 illustrates the throughput and response time prediction with $K30$. In this case, the SD algorithm using GS and MS accurately predicts system performance. The maximum percentage of error achieved with a highly loaded system is around 6.7% for the GS and 20% for the MS.

		$\delta = 0.2$	$\delta = 0.5$	$\delta = 0.7$	$\delta = 1$	Without
Fmincon	**K3**	215.21%	207.85%	194.50%	241.95%	54.10%
	K15	237.82%	213.45%	206.21%	255.92%	6.12%
	K30	334.69%	271.73%	240.30%	303.68%	3.27%
	K50	490.13%	432.53%	317.74%	427.76%	1.12%
GS	**K3**	72.39%	66.57%	64.53%	74.69%	54.23%
	K15	33.22%	25.54%	11.06%	11.24%	5.90%
	K30	33.98%	30.08%	18.56%	23.91%	3.96%
	K50	61.52%	48.13%	26.40%	31.91%	1.03%
MS	**K3**	50.36%	49.94%	51.80%	50.99%	53.44%
	K15	7.11%	5.16%	5.69%	6.30%	5.96%
	K30	4.42%	2.40%	3.58%	3.34%	3.48%
	K50	1.76%	0.78%	0.95%	0.71%	1.06%

TABLE IV: Average percentage of error of the founded demands with SD algorithm using the random models.

We also analyze the model response time prediction with K30 for the GS and MS algorithms. The results are shown in Figure 4b. For GS, the average system response time is close to the real one with an average error of around 14%. Similar is also the MS average error of 17%.

To show further benefits of the SD algorithm, we report in Figure 5 the state probability comparison between the real system and the model with 10 clients using $K30$ as sampled state space. It is possible to see that, differently from the one figure presented in Section II, the two distributions are very similar one each other and the distance value, using the HE measure, is 0.0938. This underlines that the set of demands returned by the SD algorithm is able not only to predict the performance metric of the system, but also to generate demands that correctly capture the real system dynamics.

VII. Random Models

To further investigate the robustness and performance of SD, we run a set of random experiments able to sensitivity of the result to different demands, δ parameter value, and method execution times.

To perform these experiments, we have created 100 sets of random demands, each one containing four demands, one for each service class, in the range between 0.00001 to 0.1. Each set of demands is then set into the model presented in Section VI-A to calculate the marginal state probability of the model and average performance metrics such as throughput, response time and CPU utilization of one node. These metrics are obtained by analyzing the model using the formulas presented in Section IV-B. For each random model, we then run the SD algorithm using different values of δ, chosen in the set $(0.2, 0.5, 0.7, 1)$, with and without non-linear constraints, different sample state space sizes ($K3$, $K15$, $K30$, $K50$), and with Fmincon, GS and MS searching algorithms. We run the random models with HE and 10 clients. In these experiments, we keep in consideration the Fmincon as reference point for the other algorithms but we do not consider this algorithm robust enough to be used for real demands estimation as it has been demonstrated in the previous section.

Table IV presents the mean percentage of error collected between the real and the found demands using different algorithms, search space and δ. It is clear that the MS algorithm outperforms the other two algorithms. In particular, for all the sampled state spaces bigger than $K3$ and under varying δ values, the algorithms are able to find with a very low percentage of error the real set demands. Even in the case with $K3$ the algorithm is able to find good demands compared to the other algorithms using the same state space. However, the absence of state information makes difficult for the algorithm to find better demands.

It is also interesting to see the case where the non-linear constrains are included in the search algorithms. In this situation, the algorithms are able to perform similarly to each other and the difference in performance is dictated only by the sampled state space size. Based on these results we have tested if, without non-linear constraints, the demand inference in Cassandra can improve as well or reduce the execution time. However, we noticed that the optimization algorithms returns, in this condition, higher final divergence values and in some case they are not able to converge. For this reason, we do not advise to use this method for a production system.

VIII. Conclusion and Future Work

In this paper, we have presented a new algorithm, called SD, that is able to infer the demands in distributed cloud applications. The SD algorithm differs from state-of-the-art approaches mainly for two reasons: firstly, it uses the marginal probability of the real system to perform the inference of the demands rather than average metrics as done in most algorithms. Secondly, users need to provide to the algorithm a representative queueing network model of the system, allowing to capture the entire workflows of the requests.

Through a case study using the Apache Cassandra NoSQL database, we have shown that the SD algorithm is able to identify a set of demands able to not only match the average metrics performance, but also the system behavior represented by the marginal probability of the analyzed system.

Further validation has been conducted on a set of random models where the demands have been generated within a specific range. In this case, we have demonstrated that in absence of system noise, the SD algorithm is able to perform well and incur a small error around 2%.

Future work may further investigate the SD algorithm with other different types of applications and other families of queueing network models. In addition, one may analyze weather a hybrid solution composed by a two-steps algorithm (a preliminary search and a full search) can reduce the average error and its execution time.

References

[1] D. Ardagna, G. Casale, M. Ciavotta, J. F. Pérez, and W. Wang, “Quality-of-service in cloud computing: modeling techniques and their applications,” *J. Internet Services and Applications*, 2014.

[2] S. Spinner, G. Casale, F. Brosig, and S. Kounev, “Evaluating approaches to resource demand estimation,” *Performance Evaluation*, 2015.

[3] W. Wang, G. Casale, A. Kattepur, and M. K. Nambiar, “QMLE: A methodology for statistical inference of service demands from queueing data,” *TOMPECS*, 2018.

[4] I. Perez, D. Hodge, and T. Kypraios, “Auxiliary variables for bayesian inference in multi-class queueing networks,” *Statistics and Computing*, Nov 2017.

[5] S. Dipietro, G. Casale, and G. Serazzi, “A queueing network model for performance prediction of apache cassandra,” in *Proceedings of the 10th EAI International Conference on Performance Evaluation Methodologies and Tools*, 2017.

[6] F. Baskett, K. M. Chandy, R. R. Muntz, and F. G. Palacios, “Open, closed, and mixed networks of queues with different classes of customers,” *J. ACM*, vol. 22, pp. 248–260, Apr. 1975.

[7] J. Zahorjan, “An exact solution method for the general class of closed separable queueing networks,” in *Proc. of Conference on Simulation, Measurement and Modeling of Computer Systems*, 1979.

[8] G. Bolch, S. Greiner, H. De Meer, and K. S. Trivedi, *Queueing networks and Markov chains: modeling and performance evaluation with computer science applications*. John Wiley & Sons, 2006.

[9] A. Lakshman and P. Malik, “Cassandra: A decentralized structured storage system,” *SIGOPS Oper. Syst. Rev.*, vol. 44, pp. 35–40, Apr. 2010.

[10] J. F. Perez, S. Pacheco-Sanchez, and G. Casale, “An Offline Demand Estimation Method for Multi-threaded Applications,” in *IEEE 21st International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*, IEEE.

[11] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, “Benchmarking cloud serving systems with ycsb,” in *Proc. of the 1st ACM Symposium on Cloud Computing*, ACM, 2010.

[12] Y. Bard and M. Shatzoff, “Statistical methods in computer performance analysis,” *Current Trends in Programming Methodology*, vol. 3, pp. 1–51, 1978.

[13] J. Rolia and V. Vetland, “Parameter estimation for performance models of distributed application systems,” in *Proc. of the 1995 Conference of the Centre for Advanced Studies on Collaborative Research*, 1995.

[14] P. Zarchan and H. Musoff, *Fundamentals of Kalman Filtering: A Practical Approach*.

[15] P. Cremonesi and A. Sansottera, “Indirect estimation of service demands in the presence of structural changes,” *Performance Evaluation*, 2014.

[16] W. Wang, G. Casale, A. Kattepur, and M. Nambiar, “Maximum likelihood estimation of closed queueing network demands from queue length data,” in *Proc. of the 7th ACM/SPEC on International Conference on Performance Engineering*, 2016.

[17] Z. Liu, C. Xia, P. Momcilovic, and L. Zhang, “Ambience: Automatic model building using inference,” in *Congress MSR03*, 2003.

[18] D. Kumar, L. Zhang, and A. Tantawi, “Enhanced inferencing: Estimation of a workload dependent performance model,” VALUETOOLS ’09.

[19] G. Casale, “Exact analysis of performance models by the method of moments,” *Performance Evaluation*, vol. 68, no. 6, pp. 487 – 506, 2011.

[20] P. Schweitzer, “Approximate analysis of multiclass closed networks of queues” presented at the,” in *International Conference on Stochastic Control and Optimization*, 1979.

[21] A. Bhattacharyya, “On a measure of divergence between two statistical populations defined by their probability distributions,” *Bull. Calcutta Math. Soc.*, vol. 35, pp. 99–109, 1943.

[22] E. Hellinger, “Neue begründung der theorie quadratischer formen von unendlichvielen veränderlichen.,” *Journal für die reine und angewandte Mathematik*, vol. 136, pp. 210–271, 1909.

[23] Tcpdump, “Tcpdump,” 2018.

[24] J. F. Prez and G. Casale, “Line: Evaluating software applications in unreliable environments,” *IEEE Transactions on Reliability*, 2017.

[25] G. Casale, “Automated multi-paradigm analysis of extended and layered queueing models with line,” *ACM/SPEC ICPE*, 2019.