Efficient Use of Geographically Spread Cloud Resources

Yossi Kanizo, Danny Raz, Alexander Zlotnik
Department of Computer Science, Technion, Haifa, Israel
{ykanizo, danny, azlotnik}@cs.technion.ac.il

Abstract—The demand for cloud services in each geographical location changes over time depending on the time of the day. Thus, when one data center (say in the east coast of the US) experiences peak load, other data centers (say in Europe) experience lower load. This paper addresses the efficiency of load sharing between geographically spread cloud resources. We observe that despite the network latency, for several common services it is very beneficial to share the load across two or more data centers, each located in a different time zone.

We rigorously analyze a simple setting in which customers can be redirected between two servers, each experiencing a different local load. We show that a threshold-based load sharing scheme, in which loads are redirected when exceeding some threshold, is significantly more efficient than a static load sharing scheme, where loads are redirected independently of the current state. Our load sharing techniques can reduce the average service time by 40% during peak demand in typical service scenarios. Looking at the same result from a different perspective, we show that (in the same setting) deploying our geographically based load sharing scheme can provide similar user experience with 15%-20% less resources.

To further validate our approach, we deployed Wikipedia instances on Amazon EC2 both in Europe and the US and tested our techniques using real Wikimedia access logs. Our results show that threshold-based load sharing between the US and Europe, achieves an improvement of up to 32% in average service time over these logs.

I. INTRODUCTION

With the expansion of cloud computing, global organizations can now acquire resources in various locations around the globe. This can be done for many reasons, such as resilience, survivability, or commercial constraints, but in many cases the main reason is the desire to serve local clients with minimal service times, i.e., providing the needed QoS with the minimum possible cost.

The demand for cloud based services in each geographical location changes over time depending on the time of the day. In some cases, during peak demand hours, it may be beneficial to use resources from datacenters in another continent, where at that period of time the local resources may be underutilized. However, the benefits of load balancing are hindered by the communication delays induced by it. Since the servers are located in different parts of the world, these delays vary and can be a significant part of the overall user experience.

This paper addresses the issue of service efficiency from two points of view. The first has to do with the real time decision regarding the actual server that serves a certain user request. This decision is aimed at improving the user perspective in terms of response time, which depends on the server load and network latency. Another aspect of geographically spread services is the resource allocation problem, which means deciding how much resources should be acquired at each location.

Clearly, the actual performance of the load sharing algorithm depends on the amount of resources available at each location, but at the same time the optimal decision regarding the amount of resources needed at each location depends on the load sharing policy. Therefore, the goal of this paper is twofold. First, to study the effect of network delay in realistic scenarios and the way it must be considered by the load balancing algorithm in the adaptive and oblivious cases. Second, to study efficient ways of resources acquisition for the services at different geographically spread datacenters especially in the private cloud scenario.

Many global organizations today have servers in multiple locations around the globe. Examples of such organizations can be web service providers like Google, Yahoo, Microsoft and Wikimedia, commercial institutions with the need of running computing tasks for research and development like Intel or IBM, and cloud infrastructure or content distribution network providers like Amazon, RackSpace and others. While the services offered by these organizations are different, interestingly enough their daily workloads exhibit similar patterns of load arrivals with one or more peaks around noon, lower loads in the evening and minimal loads at night. Also, the ratio between peak and low demands can be tenfold (see for example [1], [2], [3]). Figure 1 depicts the average daily traffic fluctuation for a search service as appears in Figure 2.2 of [1], skewed for time zones of 4 major groups of Internet users: Europe (London), East and West coasts of the US and Asia (Beijing). One can observe that when the East Coast in US is at the peak daily load, the loads in London and Beijing are low. Thus, servers in Europe or Asia can assist in serving loads from the East Coast of US during these periods.

However, this global load sharing scheme should clearly take into account that the help of a remote datacenter bares a considerable communication delay caused by the network distance to the remote datacenter. In other words, such a load sharing is reasonable only when the added network delay is smaller than the delay introduced by the high load on the local datacenter. If, for example, the one way network delay

This research was supported in part by the Israel Science Foundation (grant no. 1129/10).
from Europe to the east coast of US is around 80ms, then load sharing is beneficial if the request can be served in the remote location 160ms faster than locally. This, of course, highly depends on the characterization of the service as well as on the amount of resources (for example number of servers) available at the various locations.

Consider a global service with two servers\(^1\), one in the US East Coast and one in Europe (London); One can see in Figure 1 that when the server in the US experiences the highest load, the server in Europe experiences a load of about 60% of the peak load. A simple static load balancing technique is to use the load reflected in this figure and to randomly send a portion of the requests from the loaded server (in the US) to the less loaded one in Europe. The exact portion to be sent depends on the ratio between service time and network delay, where the goal is to reduce the overall average delay experienced by the users. As we show in this paper, even this simplistic method can dramatically improve the user perspective of typical services. However, when the arrival process fluctuates between high and low loads, such a static policy can still result in an inefficient resource utilization. In this paper, we develop and analyze a threshold based partially informed policy, in which requests are forwarded to the remote server only when the local queue size increases to a certain threshold. We show that such a policy can produce a much better user experience than the static scheme. We extend this policy in several ways and present comprehensive simulation and real-life studies, based on actual global service logs. The studies examine the advantages of the proposed techniques, when considering realistic delays and service parameters.

A different aspect of the reduction in the overall user perceived latency for the service is the ability to use a global load balancing system as a way to reduce the computation resources required by the service. When using common cloud services (such as EC2) the application owners can dynamically change the amount of resources at each location depending on the demand. Yet, such changes can take minutes and come with a cost and thus there is a need to make the best possible use of online resources. In private clouds (which are the current choice by many organizations) the resources acquiring decision is made in a completely different time scale (of say months), and the organization is committed to these resources for a longer time period.

Due to the common demand pattern, most services would allocate resources to handle peak demand (or what is common in the industry 95% of the peak demand). Since the ratio between the peak and average demand can be very big (a factor of 3) this means that much of the resources are underutilized most of the time. However, geographic-based load balancing allows the service owner to utilize resources in another location, and thus the service can provide the required QoS with fewer resources. We examine this aspect of load balancing and show that our load balancing technique can reduce the service resource requirement by almost 20% (for the typical services we considered).

This paper is organized as follows. In the next section we describe and analyze the load sharing model and study the methods of finding the optimal working parameters. In Section III we discuss two-way load sharing parameters. Then, in Section IV we present simulation experiment based on real-life data, and analyze the amount of resources that can be saved by deploying a global load balancing scheme. In Section V we describe the implementation and the performance of the scheme in a real emulated web service. Related work is described in Section VI and a brief discussion is provided in Section VII.

\(\text{II. Load Sharing Models}\)

We model each server as an entity that receives tasks and executes them in a scheme known as M/M/1. The arrival of tasks to the server is a Poisson stream of a given rate \(\lambda\) and the time required to execute a task follows exponential distribution with mean \(1/\mu\). The tasks are executed sequentially, one at a time, in a first-come-first-served (FCFS) policy.

Another known scheme is M/M/1/PS, where tasks arrival and sizes distributions are equal to M/M/1, but task execution is done through Processor Sharing. In this scheme the tasks do not wait in queue to get sole computation rights, but instead, all tasks are executed in parallel. The computation resources are equally divided between all the running tasks and a task finishes after it received the total computation resources that it required. In this processor sharing model, shorter tasks are expected to finish earlier than in M/M/1 as they do not wait in queue behind long tasks to begin execution. Longer tasks, however, are expected to take longer time to finish because other tasks will take some of the computation power during their execution. Nevertheless, as shown in [4], M/M/1 and M/M/1/PS have identical Markov chains and subsequently, our analysis, done for M/M/1, applies to M/M/1/PS as well.

To analyse load sharing schemes we consider a basic setting of two M/M/1 servers (denoted \(\alpha\) and \(\beta\)) located at two distant geographical locations. Tasks arrival to each server follows Poisson process with rates \(\lambda_\alpha\) and \(\lambda_\beta\), respectively. Task execution times are exponentially distributed with mean

\(\text{Fig. 1: Usage Patterns over 24 hours in GMT}\)
1/µ, where µ is the processing ability. We assume that the arrival rate to server α is at least as large as the arrival rate to server β, i.e., λα ≥ λβ. We refer to the arrival rate as the local load of the server.

While executing a task, other arriving tasks are managed according to a well defined policy, where they can either be forwarded to the other server or enqueued for later local execution according to FCFS policy. We also define the queue size at a server to be the number of tasks waiting for execution plus the task that is being executed (or zero if no such task exists). For simplicity, we assume that the round-trip networking delay between server α and server β is a constant value d. In the analysis, we use the same processing ability in both servers, however, this assumption can be relaxed, and different processing rates can be considered. We define the service time of a task as the time from the moment it arrives at the system until the moment it is executed. Our primary goal is to minimize the average service time among all tasks in the system. When a task is forwarded from one data center to another, the service time of the task is from the moment it initially arrives at the first server until the moment it gets back after being executed at the other server.

When each server executes only its own tasks, that is, there is no sharing of tasks, each server behaves as an independent M/M/1 queuing system [5]. As such, the overall expected service time for no-sharing, SN, is given by the weighted average of the two queuing systems service times. That is,

\[ S_N = \frac{\alpha \mu - \lambda_\alpha}{\alpha + \beta} \frac{\mu - \lambda_\beta}{\lambda_\alpha + \beta} \]  (1)

A. Static sharing

We now introduce the first scheme that shares tasks between the servers, named the static sharing scheme. In the literature this scheme is also referred to as static load balancing [6], [7].

In the static sharing scheme, each of the servers is aware of all the system parameters, that is, the task arrival rate at each of the servers, the processing ability and the expected communication delay, i.e., the values of λα, λβ, µ and d.

In the static sharing scheme, each task that arrives at server α is forwarded to server β with probability p. Thus, the stream of tasks that arrive at server α and are not forwarded to server β follows a Poisson process with rate (1−p) · λα. Moreover, the resulting stream of forwarded tasks follows a Poisson process with rate p · λα. Finally, combining two independent Poisson processes of rates p · λα and λβ results in a Poisson process of rate p · λα + λβ.

The expected task service time² S_S is:

\[ S_S = (1-p) \frac{p}{1-p} \lambda_\alpha + \frac{p}{1-p} \lambda_\beta + \frac{1}{1-p} + p \lambda_\alpha d \]

The optimal forwarding probability p that minimizes S_S can be found using common mathematical methods. In the general case, this requires solving a fourth degree polynomial. In the special case where d = 0 the optimal forwarding probability p equals the total load of the two servers. That is, p∗ = (λα − λβ)/(2λα). Methods of finding the optimal forwarding values in a more general setting with more than two servers are discussed in [6], [7].

B. Threshold Based Sharing

In this section, we consider the threshold based sharing scheme. As explained in the introduction, in addition to the statistical data, in this case each server knows in real time the exact size of its own queue. Upon the arrival of a new task, based on this information, the server should decide whether to enqueue the task for local execution or to forward it to the remote server. When a server receives a forwarded task from another server, it must enqueue it for local execution regardless of its real-time queue size. In [8] this policy is referred to as a weak threshold policy. This is in contrast to a strong threshold policy, where the tasks are forwarded until they arrive to a server that has local load below the defined threshold.

Since server α is more loaded (λα ≥ λβ), in this section we consider the case where only server α forwards tasks to server β. We refer to this as a one-way scheme, as opposed to a two-way scheme, in which both servers can forward tasks (evaluated in Section III).

We define t to be the local threshold of server α. When the queue size of a server is strictly smaller than t, any new task arriving at server α will be put in its local queue. However, if the queue size is exactly t, then the new task is forwarded to server β. As described earlier, any task arriving at server β (either from local customer or from server α) is enqueued for local execution.

The one-way threshold based sharing scheme can be modeled by the Markov chain depicted in Figure 2, where a state is represented by i,j the sizes of the queues at each of the servers.

The stochastic process in server α follows a M/M/1/k queue with k = t. From the analysis of M/M/1/k queue, the forwarded load from server α to server β is

\[ \lambda_j = \frac{\lambda_\alpha \rho_j^k (1 - \rho_j)}{1 - \rho_j^{t+1}} \]  (2)

where ρ_j = λ_α/µ is the ratio between loads arriving at server α and its processing ability.

The arrival process in server β follows a Markov Modulated Poisson Process (MMPP) with arrival rates λ_α + λ_β and λ_β.
depending on the number of tasks in server $\alpha$. The overall stochastic process in server $\beta$ follows $MMPP/\lambda_1/1$ and it constitutes a quasi birth-death (QBD) process [5], as depicted in Figure 2. The special case of $MMPP/\lambda_1/1$ with $\lambda_\beta = 0$ and $d = 0$ is analyzed in [9].

To find the overall expected service time, the steady state probabilities of the servers need to be computed first. These steady state probabilities are essential for computing the expected task service times as appears in (13). To compute them, we apply the matrix geometric method, that is used to analyze QBD processes. The common technique to solve QBD processes is described in full in [5] p. 127. We only give here the specific inputs to this technique, customized to our analyzed system.

Let $\pi_{ij}$ be the steady state probability for the system to be in state $i, j$ (i.e., to have $i$ tasks at server $\alpha$ and $j$ tasks at server $\beta$). Also let $\pi_j$ be the vector $(\pi_{0j}, \pi_{1j}, \ldots, \pi_{tj})$ representing the steady state probabilities of all the states with $j$ tasks at server $\beta$. We refer to this meta-state as $F_j$. Let $Q$ be the generator matrix of the semi-Markov process that corresponds to the system. The steady state probabilities satisfy

$$\sum_{0 \leq i \leq t} \pi_{ij} = 1$$  \hspace{1cm} (4)

Next we define four $(t+1) \times (t+1)$ matrices, which are the building blocks of the block-diagonal matrix $Q$

$$Q = \begin{pmatrix} B & Q_0 & 0 & \cdots \\ Q_1 & Q_0 & 0 & \cdots \\ 0 & Q_2 & Q_1 & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix}$$

$B$ describes the transition rates within $P_0$. $Q_1$ describes the transition rates within $P_j$ for $j > 0$, $Q_0$ describes the transition rates from $P_j$ to $P_{j+1}$ and $Q_2$ describes the transition rates from $P_j$ to $P_{j-1}$.

The structure of these matrices is as follows:

$$B = \begin{pmatrix} -\lambda_\alpha - \lambda_\beta & \lambda_\alpha & 0 & 0 \\ \mu & -\lambda_\alpha - \lambda_\beta - \mu & \cdots & 0 \\ 0 & 0 & \cdots & \cdots \\ \lambda_\beta & 0 & 0 & 0 \\ 0 & \lambda_\beta & 0 & 0 \\ \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & 0 & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots \end{pmatrix}$$

$$Q_0 = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ \cdots & \cdots & \cdots \end{pmatrix}$$

$$Q_1 = \begin{pmatrix} -\lambda_\alpha - \lambda_\beta - 2\mu & \lambda_\alpha & 0 & 0 \\ \mu & -\lambda_\alpha - \lambda_\beta - 2\mu & \cdots & 0 \\ 0 & 0 & \cdots & \cdots \\ 0 & 0 & 0 & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & 0 & \cdots \end{pmatrix}$$

$$Q_2 = \begin{pmatrix} \mu & 0 & 0 & 0 \\ 0 & \mu & 0 & 0 \\ 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & \mu \end{pmatrix}$$

From the repetitive structure of the QBD and (3) it follows that for all $j > 0$:

$$\pi_{j-1}Q_0 + \pi_jQ_1 + \pi_{j+1}Q_2 = 0$$  \hspace{1cm} (5)

The transitions between states $P_j$ are independent of $j$ and $\pi_j$ can be defined as a function of $\pi_{j-1}$. Therefore, there exists a constant rate matrix $R$ that satisfies for each $j > 0$:

$$\pi_{j+1} = \pi_jR = \pi_1R^j$$  \hspace{1cm} (6)

Substituting $\pi_j$ from (5), for each $j > 1$

$$\pi_1R^jQ_0 + \pi_1R^jQ_1 + \pi_1R^jQ_2 = 0$$

Which can be simplified to

$$Q_0 + RQ_1 + R^2Q_2 = 0$$  \hspace{1cm} (7)

$R$ can be found using iterative succession as described in [10]:

$$R_{(0)} = 0, \hspace{0.5cm} R_{(n)} = -(Q_0 + R^2_{(n-1)}Q_2)Q_1^{-1}$$  \hspace{1cm} (8)

Due to lack of space we do not provide a detailed discussion regarding the convergence of $R$. In our simulations we use $R = R_{(100)}$.

In the steady state the drift of the process towards smaller $P_j$ must be higher than toward higher $P_j$. This is satisfied by the condition $\lambda_\alpha + \lambda_\beta < \mu$. This implies that the spectral radius of $R$ is smaller than one, $(sp(R) < 1)$ and allows for the following simplification:

$$\sum_{n=0}^{\infty} R^n = I + R + R^2 + \cdots$$  \hspace{1cm} (9)

$$I = \sum_{n=1}^{\infty} R^n = \sum_{n=0}^{\infty} R^n = (I - R) \sum_{n=0}^{\infty} R^n$$

$$\sum_{n=0}^{\infty} R^n = (I - R)^{-1}$$

$$I = \sum_{n=0}^{\infty} R^n$$

$$\sum_{n=0}^{\infty} R^n = (I - R)^{-1}$$
Using (9) in (4) we get
\[ \pi_0 + \pi_1 \sum_{n=0}^{\infty} R^n = \pi_0 + \pi_1 (I - R)^{-1} = 1 \]  
(10)

Where \( \pi_0 \) and \( \pi_1 \) are a solution of (10) and the other boundary conditions derived from (3):
\[ \pi_0 Q + \pi_1 Q_2 = 0 \]  
(11)
\[ \pi_0 Q_0 + \pi_1 Q_1 + \pi_1 RQ_2 = 0 \]  
(12)

The previous steps provide a way to calculate the steady state probabilities \( \pi_{ij} \).

The expected processing time is a weighted average of the service times for tasks executed on server \( \alpha \),
\[ S_T = \frac{1}{\lambda_\alpha + \lambda_\beta} \sum_{i=0}^{\infty} \lambda_\alpha p_\alpha (1 - \rho_\alpha) \frac{1 + i}{\mu} + \lambda_f (d + \sum_{j=0}^{\infty} \frac{1 + j}{\mu} \pi_{tj}) + \lambda_\beta \sum_{j=0}^{\infty} \frac{1 + j}{\mu} \pi_{tj} \]  
(13)

1) Finding the optimal threshold: In a system of two servers, one with high load where the threshold will be enforced and the other with low load that will assist in executing the tasks, it is interesting to find the threshold that will minimize the average task service time in the system.

To use this model in practice, one needs to determine the best possible threshold for given settings.

The service time of tasks executed (not forwarded) at server \( \alpha \) and their weight, increase monotonically when increasing the threshold value \( t \). At the same time, the weight of the communication delay and the weight and values of the average service times for tasks executed on server \( \beta \) monotonically decrease. The sum of two monotonic opposite series gets a minimal value in a single value of \( t \) - the optimal threshold. Small threshold will minimize the service time of tasks that are executed on server \( \alpha \), but will cause large load on server \( \beta \) and high weight for the communication delay. Large threshold will underutilize the potential benefit of load sharing. To find the optimal threshold, one needs to solve the system as described in Section 2.

Equation 13 can be replaced by:
\[ S_T = S_T^* + d \cdot \lambda_f \]  
(14)

Where \( \lambda_f \) is given by (2) and \( S_T^* \) is given by:
\[ S_T^* = \sum_{i=0}^{t-1} \rho_\alpha^{i+1} \frac{1 - \rho_\alpha}{1 - \rho_\alpha^{i+1}} (1 + i) + \rho_\beta^{i+1} \frac{1 - \rho_\beta}{1 - \rho_\beta^{i+1}} \sum_{j=0}^{\infty} \pi_{tj} (1 + j) + \rho_\beta \sum_{j=0}^{\infty} \sum_{i=0}^{t} \pi_{tj} (1 + j) \]

Table (I) contains the value of \( S_T^* \) for thresholds \( 1 \leq t \leq 7 \) in systems with \( 0.9 \geq \rho_\alpha \geq 0.5 \), \( 0.6 \geq \rho_\beta \geq 0 \). This table can be extended and pre-calculated in advance and together with (14) one can find the optimal threshold and the expected average task service time for many practical scenarios in O(1).

Note that while all tasks arriving at server \( \alpha \) are part of a Poisson stream, the tasks that are sent from server \( \alpha \) to server \( \beta \) (with communication delay) and the tasks arriving to server \( \beta \).

The service time of tasks executed (not forwarded) at server \( \alpha \) and their weight, increase monotonically when increasing the threshold value \( t \). At the same time, the weight of the communication delay and the weight and values of the average service times for tasks executed on server \( \beta \) monotonically decrease. The sum of two monotonic opposite series gets a minimal value in a single value of \( t \) - the optimal threshold. Small threshold will minimize the service time of tasks that are executed on server \( \alpha \), but will cause large load on server \( \beta \) and high weight for the communication delay. Large threshold will underutilize the potential benefit of load sharing. To find the optimal threshold, one needs to solve the system as described in Section 2.

Equation 13 can be replaced by:
\[ S_T = S_T^* + d \cdot \lambda_f \]  
(14)

Where \( \lambda_f \) is given by (2) and \( S_T^* \) is given by:
\[ S_T^* = \sum_{i=0}^{t-1} \rho_\alpha^{i+1} \frac{1 - \rho_\alpha}{1 - \rho_\alpha^{i+1}} (1 + i) + \rho_\beta^{i+1} \frac{1 - \rho_\beta}{1 - \rho_\beta^{i+1}} \sum_{j=0}^{\infty} \pi_{tj} (1 + j) + \rho_\beta \sum_{j=0}^{\infty} \sum_{i=0}^{t} \pi_{tj} (1 + j) \]

Table (I) contains the value of \( S_T^* \) for thresholds \( 1 \leq t \leq 7 \) in systems with \( 0.9 \geq \rho_\alpha \geq 0.5 \), \( 0.6 \geq \rho_\beta \geq 0 \). This table can be extended and pre-calculated in advance and together with (14) one can find the optimal threshold and the expected average task service time for many practical scenarios in O(1).

Note that while all tasks arriving at server \( \alpha \) are part of a Poisson stream, the tasks that are sent from server \( \alpha \) to server \( \beta \) are sent only when server \( \alpha \) is at threshold state. Thus, tasks arrival to server \( \beta \) is highly correlated and if both servers experience the same average arrival rate (in terms of tasks per time unit) then tasks in server \( \beta \) will experience (in general) longer waiting time. For that reason, when seeking for the optimal threshold \( t \), one can start with the maximal \( t \) value for which \( \lambda_\alpha - \lambda_f \leq \lambda_\beta + \lambda_f \) and increase \( t \) as long as \( S_T \) decreases.

C. Evaluating the model

We implemented the solution as described above and tested its accuracy in determining the overall expected task service time for various values of \( \lambda_\alpha, \lambda_\beta, \mu, d \) and \( t \) (threshold).

Figure 3 depicts the results, where \( \lambda_\alpha \) varies from 0.3 to 0.95, \( \lambda_\beta = 0.3 \), \( \mu = 1 \) and \( d = 1 \). The model results are obtained using (13) with the computed \( \pi_{tj} \) values. The results of the simulation were obtained by running 2 million tasks on an ad hoc Java based simulator that we developed. One
can clearly see that the analytical model and the simulation have identical results and that the analytical model provides an accurate way to compute the expected service time for the one-way threshold based sharing scheme. Thus, in the rest of this section we only depict the simulation results.

Figure 4 depicts the average task service times for $\lambda_\alpha = 0.8$, $0.3 \leq \lambda_\beta \leq 0.5$, $\mu = 1$, $d = 1$ and different threshold values. The optimal thresholds for $\lambda_\beta=0.3$, $0.4$, $0.5$ are $2$, $3$ and $4$ respectively. The round-trip delay $d$ is equal to the average execution time of a task. Intuitively, high local loads on server $\beta$ require a larger threshold (for server $\alpha$); this ensures the invocation of load sharing in fewer, more extreme cases of real-time load on server $\alpha$.

When $d$, the round-trip delay between the servers, increases, the benefit of load sharing drops and the threshold should be adjusted accordingly. Figure 5 compares the average task service times for $\lambda_\alpha = 0.8$, $0.3 \leq \lambda_\beta \leq 0.5$, $\mu = 1$, and $d$ values that vary from zero to 100 times the average execution time of task ($\frac{1}{\mu} = 1$). At every point on this graph, the optimal threshold is used. When the network delays are significantly larger than the task’s execution time, the threshold is set to a very high value, thus, essentially preventing any load sharing between the servers.

Figure 6 compares the gains of static and one-way threshold based sharing policies for settings of $0.5 \leq \lambda_\alpha \leq 0.9$, $\lambda_\beta = 0.3$ or $0.5$, $\mu = 1$ and $d = 1$. It shows that when compared to no sharing, both policies improve the average task service time and that in realistic scenarios the threshold based load sharing is better by $10\%$-$20\%$ than the static sharing. For purposes of comparison, in every setting the optimal parameters for static

and threshold based sharing were used.

III. TWO-WAY SHARING

Until now we analyzed systems with two datacenters and a one-way threshold based sharing scheme where redirection of tasks can be performed only by the most loaded datacenter. In this one-way scheme all tasks arriving at datacenter $\beta$ (from Poisson stream of arrivals $\lambda_\beta$ or from datacenter $\alpha$) are executed on datacenter $\beta$ regardless of the actual state (in terms of the size of the queue). This scheme is problematic, since there may be times in which the real queue size in datacenter $\beta$ is larger than the one in datacenter $\alpha$. This situation will occur more often if both local loads are high.

To address this shortcoming, we propose a scheme where each datacenter has a threshold on its load. When a datacenter receives a new task from its local arrivals process, it checks whether the queue size at the datacenter is equal or above its threshold, and if so, the task is forwarded to the other datacenter. Otherwise, the task is enqueued and eventually executed locally. Tasks arriving from the other datacenter (due to exceeding the threshold there) are always handled locally. Unlike in the one-way scheme, a datacenter here can have more tasks in its queue than its threshold size. In this case, the generator matrix $Q$ does not have a simple repetitive structure like the structure in the one-way threshold based sharing scheme and thus there is no easy way to define an analytical solution as we did in Section II.

Figure 7 compares the gains of one-way and two-way threshold based sharing policies. For every setting between $0.5 \leq \lambda_\alpha \leq 0.9$, $\lambda_\beta = 0.3$ or $0.5$, $\mu = 1$ and $d = 1$ the optimal threshold is considered. When datacenter $\beta$ is more loaded ($\lambda_\beta = 0.5$), the two-way threshold sharing policy improves the performance over one-way threshold based sharing.
scheme by 10%. Where as, when datacenter $\beta$ is less loaded ($\lambda_\beta = 0.3$), the two-way threshold based sharing policy has only 5% performance improvement over one-way threshold based sharing.

IV. REAL-LIFE USAGE PATTERNS AND RESOURCE SAVING

To evaluate the advantages of using our scheme, we simulate the expected results of using it on Wikimedia setting and their usage statistics (which are publicly available from: [3]). Wikimedia is the foundation that delivers Wikipedia and other popular content services. Its main service, Wikipedia, has more than 18M page views every hour and its main populations of users are located in Europe - 40%, North America - 30% and Asia - 18% ([3]). Being a globally popular service with publicly available usage statistics makes it an interesting testbed for analyzing our global load balancing techniques. Wikimedia serves the users using two sites, one in Amsterdam, serving requests from Europe, and one in Tampa, Florida, serving requests from the rest of the world.

To test our threshold based policies on Wikimedia data we select four time points on the 24 hours loads graph in Figure 8 and simulate the loads reflected then. In our simulations, we set the average task execution time to be 80ms, i.e., $\mu = 1/80$ and round trip communication time $d = 160\text{ms}$ (which reflects the actual round trip time between Florida and Amsterdam). Having communication delay twice the size of the average task execution time is similar to the setting of $\mu = 1$, $d = 2$ that was tested in Section III. We also select the peak load to be 80% of the processing ability, i.e., in order to simulate the peak load, we set $\lambda$ to be $0.8 \cdot \mu$. Other loads are simulated by setting $\lambda_\alpha$ and $\lambda_\beta$ to be relative to the simulation value of the peak load. The points of interest are detailed in Table II.

While improving the service times is important, it is also interesting to assess the potential of load sharing to minimize the equipment costs of datacenters, without hindering the overall service times. We assume a linear ratio of datacenter equipment cost to its processing ability, and study how much we can decrease of $\mu$ if a load sharing policy is used.

For every setting of $\lambda_\alpha$, $\lambda_\beta$ and $\mu$ with no sharing, we find $\mu'$ so that $S_T(\lambda_\alpha, \lambda_\beta, \mu') = S_N(\lambda_\alpha, \lambda_\beta, \mu)$, i.e., $\mu'$ that maintains the same average service time with an optimal two-way sharing policy.

Figure 9 depicts the ratio between $\mu'$ and $\mu$ in the points of interest. It can be seen that when there is a big difference between the loads of datacenter $\alpha$ and $\beta$, invoking a load sharing policy allows to use significantly less resources and still maintain the same level of service. When both datacenters have high loads, resources saving through threshold based sharing scheme is limited. For example, at 7:00 it is possible to reduce the processing ability of both servers by 15% and not experience a deterioration in the overall service time.

V. EMULATION OF A REAL-LIFE GLOBAL WEB SERVICE

It is well known that real workload does not follow the M/M/1 model. To address this, we tested the performance of our scheme on a real web service using real logs. To implement a global load sharing environment, we run our own Wikipedia servers on Amazon EC2 machines in Ireland and in Virginia, US. This setting is very similar to the real Wikimedia setting that operates a datacenter in Amsterdam serving requests from Europe, and a datacenter in Florida, US, serving requests from the rest of the world. To the best of our knowledge, this is the first time that real-life Wikipedia access traces are used to analyze load balancing schemes.

Wikimedia generates dumps of their databases on a daily basis3 and maintains an open source project of the software that runs on its backend servers4. In addition, Wikimedia has released a representative set of access traces5 collected between Oct 2007 to Jan 2008 and they were analyzed in [11]. Each request was recorded with a probability of 10%. Therefore, the published traces contain only 10% of the original accesses to Wikimedia.

A. Testing Environment

For testing the threshold based load sharing scheme, we created a setting of Wikipedia servers and implemented load balancing proxies and clients as depicted in Figure 10. Our Wikipedia servers (wiki servers) were customized from an image of a virtual machine created in Amazon EC2 by Bitnami Inc. The image has preinstalled MediaWiki software with Apache server and MySQL database. Initially, the wiki

![Fig. 8: Wikimedia requests statistics collected on Monday, Jun 20, 2011](image1.png)

![Fig. 9: Resource saving on Wikimedia data](image2.png)

---

3 New Wikimedia database dumps: http://dumps.wikimedia.org
4 MediaWiki software project: http://www.mediawiki.org
5 Wikimedia access traces: http://www.wikibench.eu
database of the server is empty, with no wiki entries. In order to populate the server with wiki entries that are most relevant to the period of the traces, we use Wikimedia database dumps from June 2008\(^6\). The process of importing a database dump and configuring the MediaWiki settings is technical and the details are omitted due to space constraints. The database dump contains wiki articles (wikis) and meta information about the images that are used in the articles (such as file sizes). The images themselves, however, are not published with the database dumps, due to space and copyright issues. Since they constitute a large portion of the access traces, we created image files filled with random numbers according to the original image sizes.

To enforce threshold based load balancing policy we implemented the LBProxy, which is a customization of a Tomcat server virtual machine created in Amazon EC2 by Bitnami Inc. The LBProxy receives requests using servlet methodology and serves each request using a separate thread. For each arriving request, the LBProxy reissues the same request either to its local wiki server, or to a remote LBProxy, depending on the load balancing policy. When a response for the new request is returned, it is then used to respond to the original request.

We generate requests using client machines that run on EC2 as well. The clients are customized from the code available at [11]. To analyze accesses that are affected by diurnal changes we focus on wiki requests of a specific language that is likely to be used in a single timezone. Languages such as English, Spanish, French and Dutch, are used in different continents and do not fit our needs. We focus on the Italian language, which has the 5th largest collection of wiki articles, with almost 1M wikis in July 2012.

When the browser loads a wiki article, it first creates a request for the article’s URL. Then, based on the page contents, it generates requests for pictures (if relevant for the article) and many auxiliary look and feel files including stylesheets, images etc. The published traces contain all these requests. Each access record consists of four entries: (i) ID (with no specific meaning), (ii) access time (in milliseconds), (iii) URL, and, (iv) indicator for save operations.

All the machines that we use are of c1.xlarge type in Amazon EC2. These are 8 virtual core machines with computing power of 2.5 ECU each. The main reason for selecting these high throughput machines for clients and LBProxy is to minimize their effect on measurements. For the wiki server we first used weaker virtual instances, but they were unable to deal with the given loads. Therefore, we ended up using c1.xlarge. Dealing with loads was also reported problematic in [11], leading them to reduce loads by sampling traces and keeping less than 0.5% of the original English trace. In the end, in [11], the experiments reach a maximum average load of 6 requests/sec, while we run up to 28 requests/sec as an average load.

B. Data Characteristics

Figure 11 displays the average rate of requests per second of Italian wiki requests during 24 hours on Wednesday, Dec 5, 2007\(^7\). The graph clearly resembles Figure 1.

While running the experiments, we noticed a significant difference between running traces for the first time (cold runs), and subsequent runs. We found that in first runs, when wiki articles are not in the wiki server’s memory, the execution of wiki requests is very slow, leading to system failure. In subsequent runs, when the memory successfully fills up, the execution time improves dramatically. Figure 12 depicts the difference in execution time between requests served only from disk (cold run) and requests served from memory. The trace used for this graph had a very low load, allowing the server to demonstrate its processing ability $\mu$.

---

\(^6\)Wikimedia database: http://www.linguateca.pt/GikiCLEF/GIRA/collections

\(^7\)The graph is skewed to fit GMT+2, the Italian time zone, however it is suspected that access traces were post-processed to Florida time.
executing the average execution time of a wiki article is 400ms. Having been seen that in a warm run (when wiki articles are in memory), are wiki articles, results in 14.6 wiki article requests/sec. It can time to execute. Serving 28 requests per second, 52% of which, onds, and requests of wiki articles that take substantially longer

Requests of wiki content images that are executed in millise-

of locally served wiki article requests exceeds the threshold. LBProxy only if it is a wiki article request and the number request arrives at the LBProxy, it is forwarded to the remote overall average service time per access request.

This corresponds to a typical 7 hours time difference be-

tween the two locations and measure the distributed load sharing and resource optimization in realistic deployment aspects of such a load sharing system, deploy

based on periodic updates. These papers concentrate on the

destination where the requests should be forwarded to is

authors use a threshold on the real-time usage rate of the loca-

papers that discuss global threshold based load balancing using

highly correlated. Consequently, the behavior of the system as well as the mathematical methods are rather different. Two papers that discuss global threshold based load balancing using locally known information are [14], [15]. In these papers the authors use a threshold on the real-time usage rate of the local CPU-disk as indication for the local load. The decision of the destination where the requests should be forwarded to is based on periodic updates. These papers concentrate on the deployment aspects of such a load sharing system, deploy heuristics, and do not provide a theoretical model to analyze the tradeoff between the load and the network based delays.

Fig. 13: Wikipedia Service: Gain of the Threshold Based Load Balancing

The graph also depicts that there are two classes of requests. Requests of wiki content images that are executed in milliseconds, and requests of wiki articles that take substantially longer time to execute. Serving 28 requests per second, 52% of which, are wiki articles, results in 14.6 wiki article requests/sec. It can be seen that in a warm run (when wiki articles are in memory), the average execution time of a wiki article is 400ms. Having 8 virtual cores in the c1.xlarge machine, each capable of executing $0.4^{-1} = 2.5$ requests/sec, allows to execute up to 20 wiki article requests per second. Or in other words, to have an average load of more than 73% in a real system.

C. Experiments

To test the effectiveness of the threshold based load sharing scheme, we emulate load sharing between our machines running on Amazon EC2 in the US East Coast and in Europe. Running half hour experiments, we generate trace-driven local loads. In the US we generate wiki requests of daily peak (e.g., starting at 19:00, see Figure 11). At the same time in Europe we generate normal time access requests (e.g., starting at 2:00). This corresponds to a typical 7 hours time difference between inter-continental locations. We enforce a threshold based load sharing policy between the two locations and measure the overall average service time per access request.

Based on the data depicted in Figure 12, we restrict the threshold policy only to wiki article requests. Whenever a request arrives at the LBProxy, it is forwarded to the remote LBProxy only if it is a wiki article request and the number of locally served wiki article requests exceeds the threshold. In each experiment, both LBProxy servers have the same threshold.

Figure 13 depicts that threshold based load sharing policy significantly improves the average service time of tasks in the tested system. The average service times of all requests and wiki article requests in particular, are reduced from 332ms and 588ms to 250ms and 445ms, respectively. Introducing a 32% decrease in the average service time. Similarly to our simul-

ations, the communication delay in these experiments affects the optimal threshold selection. The wiki server machine has 8 virtual cores, making it reasonable to set the threshold at 8. However, due to communication delay posed by forwarding access requests, the optimal threshold is around $t = 10$.

VI. RELATED WORK

Load balancing is a well studied field with applications in many areas such as operations, customer service, networking, etc. While the majority of the research papers in this area either analyze static policies [6], [7] (where only statistical information is available) or informed policies [12], [13], [14], [15] (full information model), this paper addresses what we call a partially informed policy that may depend on accurate, real-time, local information (such as the threshold based scheme).

In WAN settings, the communication delay poses a significant penalty on service time and load-information relevancy. Tantawi et al consider in [6] a model of distributed load balancing with communication delay and propose an algorithm for minimal mean service time using static load balancing policy. Their model was later simplified in [7]. Further development of this model is found in [12]. In [13], the authors focused on the cost related to obtaining the load information, and compared the benefit of using static and dynamic policies. Our paper differs from these works in the scope of information used for making load balancing decisions.

Previous works addressing one-way threshold based load balancing between two servers [9], [16] use a similar method of analyzing the system as we do. The system is modeled as a quasi-birth-death (QBD) process and solved using (an optimized) matrix geometric method [5]. However, these works do not address the communication delay and its effects on the optimal threshold. Moreover, the works address a setting of load arrivals only to one main server and not a distributed setting where loads arrive to all servers.

The concept of threshold-based load balancing is discussed first in [8]. The analysis uses differential equations and assumes a large number of servers, which allows ignoring the high correlation between server states. Our work differs in that we consider a small number of machines, so the queues are highly correlated. Consequently, the behavior of the system as well as the mathematical methods are rather different. Two papers that discuss global threshold based load balancing using locally known information are [14], [15]. In these papers the authors use a threshold on the real-time usage rate of the local CPU-disk as indication for the local load. The decision of the destination where the requests should be forwarded to is based on periodic updates. These papers concentrate on the deployment aspects of such a load sharing system, deploy heuristics, and do not provide a theoretical model to analyze the tradeoff between the load and the network based delays.

VII. DISCUSSION

In this paper we rigorously study partially informed load balancing policies with the presence of network delay. We use this model to evaluate the advantages of both online distributed load sharing and resource optimization in realistic scenarios. Our results indicate that the proposed threshold schemes can be very beneficial for modern cloud based global services. Our model is based on Poisson streams and M/M/1 and M/M/1/PS queues, but according to the emulation of real logs the advantages of the scheme are substantial in real-life scenarios as well.
ACKNOWLEDGEMENTS

The use of the Amazon infrastructure was generously supported by the AWS in Education Research Grant. We would like to thank Amir Nahir for fruitful discussions that helped analysing cloud machines behavior and processing capacity.

REFERENCES