Session-Level Load Balancing for High-Dimensional Systems

Dennis Roubos and Sandjai Bhulai

***Abstract*—Load balancing is critical for the performance of big server clusters. Although many load balancers are available for improving performance in parallel applications, the load-balancing problem is not fully solved yet. Recent advances in security and architecture design advocate load balancing on a session level. However, due to the high dimensionality of session-level load balancing, little attention has been paid to this new problem. In this paper we formulate the session-level load-balancing problem as a Markov decision problem. Then, we use approximate dynamic programming to obtain approximate load-balancing policies that are scalable with the problem instance. Extensive numerical experiments show that the policies have nearly optimal performance.**

***Index Terms*—Approximate dynamic programming, Markov decision processes, session-level load balancing.**

I. Introduction

MANY content-intensive applications have scaled beyond the point where a single server can provide adequate processing power. This raises the need for flexibility to deploy additional servers quickly and transparently to end-users. The technique that addresses this need is load balancing, i.e., the process of transparently distributing service requests across a group of servers. It also addresses several requirements that are becoming increasingly important in computer networks, such as *increased scalability*, *high performance*, and *high availability and disaster recovery*.

While many effective load-balancing strategies have been developed that perform load balancing on the level of service requests, new applications and architectures require load balancing on the level of user sessions. In these cases, the load-balancing algorithm is carried out only when a user requests a new session. This load-balancing problem on a session level is not yet solved completely and has received little attention due to its complexity.

The literature on the performance and effectiveness of load-balancing algorithms is widespread. We refer to [1] for an excellent overview of the available load-balancing techniques applied in different application areas. However, the vast majority of the performance-related papers on load balancing that have appeared are focused on *request-level* load balancing (cf. [2] for an overview of request-level load-balancing algorithms). However, a main disadvantage of request-level load-balancing algorithms is that they are highly vulnerable to unsecured transactions. In secure environments where confidentiality of data and integrity of the network is of high importance, it is crucial that load balancing is done at the *session level* instead of request level. Unauthorized data queries should be kept separate from accessing the data of clients directly. Therefore, a farm of terminal servers can serve as an intermediate layer to the outside world so that clients should request a session for all activities for outgoing data traffic. In these cases, the load-balancing algorithm is carried out only when a user requests a new session. Once the session has been assigned to a server, all subsequent service requests generated in this session are directed to this server. Note that it is not desirable to switch this session to a different terminal server due to large overhead and switching times. This creates additional complexity for load-balancing algorithms, since, e.g., the arrival moments of future requests during an active session are not known.

Motivated by this, a wide variety of commercial session-level load-balancing products have been brought to the market (see, e.g., [3], [4], [5]). Rather surprisingly, however, despite the large number of available products, relatively little is known about the efficiency of these session-level load-balancing algorithms. These observations have raised the need for studying and optimizing the effectiveness of session-level load-balancing algorithms.

In this paper, we study load balancing of sessions on a farm of terminal servers. Whenever a new session is requested by a client, the load balancer needs to assign it to an available terminal server. The sessions are active for as long as the clients do not terminate their sessions. Hence, all activities (e.g., browsing the web, opening files) by a client induce load on the terminal server to which the session is assigned to. We focus on load-balancing algorithms that are easy to implement in real systems. To this end, we formulate the problem in Section II. This model is of high dimensionality making standard solution techniques to derive the optimal policy intractable. Therefore, we present approximations in Section III. We study the performance of these approximations against the optimal policy and other well-known algorithms in Section IV. Finally, we conclude in Section V with the conclusions and topics for further research.

II. Problem formulation

Consider a network with a finite number of clients; we will denote the set of clients $\mathcal{I}$ by $\mathcal{I} = \{1, \ldots, I\}$. The state of each client $i \in \mathcal{I}$ is determined by the variable $x_i \in \{0, 1\}$ describing whether client i is in the off-state or on-state, respectively. The time that client i stays in the off-state and in the on-state is exponentially distributed with parameters γ_i

D. Roubos and S. Bhulai are with the VU University, Faculty of Sciences, De Boelelaan 1081a, 1081 HV Amsterdam. E-mail: droubos@few.vu.nl and sbhulai@few.vu.nl.

Manuscript received July XX, 2008; revised July XX, 2008.

and θ_i, respectively. When a client changes from an off-state to an on-state, i.e., he starts a new session, then his session has to be assigned to a server by a load balancer so to that his future service requests during that session may be handled. The set of servers $\mathcal{J}$ is given by $\mathcal{J} = \{1, \ldots, J\}$. If client i is in an on-state, i.e., $x_i = 1$, then $k_i \in \mathcal{J}$ denotes the server to which his session is assigned to by the load balancer.

When client i is in the on-state, he generates service requests (i.e., jobs) with exponentially distributed interarrival times with parameter λ_i. All jobs of client i are routed to server k_i to which he is assigned to. Within the session, the client generates jobs of different types. The set of different types of jobs is denoted by $\mathcal{L} = \{1, \ldots, L\}$, and a job of client i is of type $l \in \mathcal{L}$ with probability $p_{i,l}$. All jobs destined for server k_i are pooled and are served in a processor-sharing fashion. We assume that a job on server $j \in \mathcal{J}$ of type $l \in \mathcal{L}$ has exponentially distributed service requirements with parameter $\mu_{j,l}$ when the server works at full rate on that job. Let Y be a $J \times L$ matrix with $Y_{j,l}$ the number of type l jobs on server j. Then, the load on server j is defined as the number of jobs on server j, and expressed as $y_j = \sum_{l=1}^{L} Y_{j,l}$ for all $j \in \mathcal{J}$.

The load balancer acts as a decision maker every time a client changes from the off-state to the on-state. It then needs to decide to which server his session needs to be assigned such that the imbalance in the loads of the different servers is minimized. More specifically, when the system is in state $s = (\vec{x}, \vec{k}, Y)$, the system is subject to direct costs $c(s)$ per unit time given by

$$c(s) = \frac{2}{J(J-1)} \sum_{m=1}^{J} \sum_{n=m+1}^{J} |y_m - y_n|.$$

Note that this cost function indeed penalizes differences in load between different servers. This might not be the ultimate goal in itself, but rather a means to achieve, e.g., high performance. However, this cost function is consistent with the objective function present in many load balancers (e.g., the Linux Virtual Server load balancer [6]).

Based on a state $s = (\vec{x}, \vec{k}, Y)$, i.e., the state of the clients, the assignment of the clients to the servers, and the number of jobs running on each server, the long-term average costs g for a given policy π can be computed by

$$g(\pi) = \lim_{T \to \infty} \mathbb{E}^{\pi} \frac{1}{T} \int_0^T c(\mathrm{d}S_t),$$

where S_t is the random variable denoting the state at time t. The objective of the load balancer is to find a policy π^* that minimizes the long-term average costs, thus $g := g(\pi^*) = \min_\pi \{g(\pi)\}$.

In order to derive optimal policies, we cast this problem as a Markov decision problem. This consists of a description of the state space $\mathcal{S}$, the possible actions $\mathcal{A}$, the transition probabilities p, and the cost structure c (discussed in the previous paragraph). To this end, first note that the information on the state of client $i \in \mathcal{I}$ (i.e., the variable x_i) and the information on the server to which he is assigned to (i.e., the variable k_i), can be combined into a single variable by adding 0 to the set $\mathcal{J}$. Indeed, the fact that $k_i > 0$ implies $x_i = 1$, and $k_i = 0$ implies $x_i = 0$. Let $\vec{k}$ be the vector $\vec{k} = (k_1, \ldots, k_I)$. By combining this information, we can reduce the size of the state space $\mathcal{S}$ defined by $\mathcal{S} = \{(\vec{k}, Y) : \vec{k} \in \big(\mathcal{J} \cup \{0\}\big)^I, Y \in \mathbb{N}^{IJ}\}$. A state $s = (\vec{k}, Y) \in \mathcal{S}$ represents that client i is assigned to server k_i if $k_i > 0$ and not assigned otherwise, and that there are $Y_{j,l}$ jobs of type l on server j. The action space $\mathcal{A} = \mathcal{J}$ with $a \in \mathcal{A}$ denoting that a client is assigned to server a when he becomes active.

We denote the transition rate of going from s to s' (before taking any action) by $p(s, s')$. If in state $s = (\vec{k}, Y) \in \mathcal{S}$ action $a \in \mathcal{A}$ is chosen for client $i \in \mathcal{I}$, then the state s' becomes $s' = (\vec{k} + e_i \cdot (a - k_i), Y)$, i.e., the i^{th} component of $\vec{k}$ is replaced by a. Therefore $p\big((\vec{k}, Y), (\vec{k} + e_i \cdot (a - k_i), Y)\big) = \gamma_i$ for this event. Similarly, the event that an active client $i \in \mathcal{I}$ becomes inactive occurs with rate $p\big((\vec{k}, Y), (\vec{k} - e_i \cdot k_i, Y)\big) = \mathbb{1}_{\{k_i > 0\}}\, \theta_i$. When client $i \in \mathcal{I}$ is active, he starts jobs of type $l \in \mathcal{L}$ with rate $p\big((\vec{k}, Y), (\vec{k}, Y + e_{k_i,l})\big) = \mathbb{1}_{\{k_i > 0\}}\, \lambda_i p_{i,l}$. Finally, a job of type $l \in \mathcal{L}$ on server $j \in \mathcal{J}$ is served in a processor-sharing fashion with rate $p\big((\vec{k}, Y), (\vec{k}, [Y - e_{j,l}]^+)\big) = \mu_{j,l} \cdot \frac{Y_{j,l}}{y_j}$. Next we uniformize the system (see Puterman [7, Section 11.5]). Without loss of generality, we assume for simplicity that the maximum rate of change in the system is bounded by 1 (we can always get this by scaling). Uniformizing is equivalent to adding dummy transitions (from a state to itself) such that the rate out of each state is equal to 1; then we can consider the rates to be transition probabilities. Moreover, uniformization allows us to develop a recursive procedure to compute optimal policies, as we will describe next.

Let $V(\vec{k}, Y)$ be a real-valued function defined on the state space. This function will play the role of the relative value function, i.e., the asymptotic difference in total costs that results from starting the process in state $(\vec{k}, Y)$ instead of some reference state. The long-term average optimal actions are a solution of the optimality equation (in vector notation) $g + V = TV$, where T is the dynamic programming operator acting on V defined as follows:

$$\begin{aligned}
TV(\vec{k}, Y) = {} & \sum_{i=1}^{I} \mathbb{1}_{\{k_i = 0\}} \gamma_i \min_{a \in \mathcal{J}} \big\{ V\big(\vec{k} + e_i \cdot (a - k_i), Y\big) \big\} \\
& + \sum_{i=1}^{I} \mathbb{1}_{\{k_i > 0\}} \theta_i V\big(\vec{k} - e_i \cdot k_i, Y\big) \\
& + \sum_{i=1}^{I} \sum_{l=1}^{L} \mathbb{1}_{\{k_i > 0\}} \lambda_i p_{i,l} V(\vec{k}, Y + e_{k_i,l}) \\
& + \sum_{j=1}^{J} \sum_{l=1}^{L} \mu_{j,l} \cdot \frac{Y_{j,l}}{y_j} V\big(\vec{k}, [Y - e_{j,l}]^+\big) \\
& + \big(1 - \sum_{i=1}^{I} [\mathbb{1}_{\{k_i = 0\}} \gamma_i + \mathbb{1}_{\{k_i > 0\}} \theta_i + \lambda_i] \\
& \qquad - \sum_{j=1}^{J} \sum_{l=1}^{L} \mu_{j,l} \cdot \frac{Y_{j,l}}{y_j} \big) V(\vec{k}, Y) \\
& + \frac{2}{J(J-1)} \sum_{m=1}^{J} \sum_{n=m+1}^{J} |y_m - y_n|,
\end{aligned} \tag{1}$$

for all $(\vec{k}, Y) \in \mathcal{S}$. The first term of Equation (1) models a

transition from the off-state to the on-state of a client. For client i this event can only occur when he is in the off-state, i.e., $k_i = 0$. In that case the load balancer has to decide to which server the client should be assigned to and stores the decision at the i^{th} entry of vector $\vec{k}$. Similarly, the second term models the transition from the on-state to the off-state of a client. The third term deals with the process that generates jobs when a client is active. The fourth term models the services on the servers. The fifth term is the dummy transition due to uniformization. The last term is the direct costs that are incurred at the epoch.

The optimality equation $g + V = TV$ is hard to solve analytically in practice. Alternatively, the optimal actions can also be obtained by recursively defining $V_{t+1} = TV_t$ for arbitrary V_0. For $t \to \infty$, the maximizing actions converge to the optimal ones (for existence and convergence of solutions and optimal policies we refer to [7]). In Section IV we adopt this approach to compute optimal policies and denote this algorithm by OPT. Also note that the state space is very big, $\mathcal{S} = J^I \times \mathbb{N}^{IJ}$. Hence, even the recursive procedure turns out to be intractable, due to the dimension of the state space. In order to avoid this curse of dimensionality, we look at state aggregation methods to reduce the dimensionality of the state space. This algorithm is described in the next section.

III. Approximation

In this section we develop approximations to the dynamic programming operator such that the state space will be reduced. The reduction should be such that the problem is scalable in the number of clients and servers while it still allows for efficient derivation of nearly optimal policies. To this end, we use aggregation on the subsets of the state space. A first reduction follows from ignoring the job types so that one solely counts the number of jobs that run on each server. The matrix Y then reduces to the vector $\vec{y}$ with entries $y_j = \sum_{l=1}^{L} y_{l,j}$. A further reduction can be achieved when one also ignores the characteristics of each client and only counts the number of active clients. We then obtain a vector $\vec{x}$ with entries $x_j = \sum_{i=1}^{I} \mathbb{1}_{\{k_i=j\}}$ for server $j = 1, \ldots, J$.

One can expect the first reduction to give better policies for the load balancer than the second approximation. The reason for this is that the first reduction has more detailed information to base its decisions on. However, the second reduction is much faster in computation, since the number of states is significantly smaller. To study this tradeoff, we formalize the two approximations by studying the dynamic programming operator for these two reductions.

Approximation 1 (APP1)
Our first approximation is based on using the vector $\vec{k}$ and $\vec{y}$. Thus, the algorithm has information on each individual client and the total number of jobs running on each server. The dynamic programming operator is very similar to the one specified in Equation (1). The changes only occur in the terms for the job arrival and the job completion. The operator T is therefore given by

$$\begin{aligned} TV(\vec{k},\vec{y}) &= \sum_{i=1}^{I} \mathbb{1}_{\{k_i=0\}}\gamma_i \min_{a\in\mathcal{J}} \left\{V\big(\vec{k}+e_i\cdot(a-k_i),\vec{y}\big)\right\} \\ &+ \sum_{i=1}^{I} \mathbb{1}_{\{k_i>0\}}\theta_i V(\vec{k}-e_i\cdot k_i,\vec{y}) \\ &+ \sum_{i=1}^{I} \mathbb{1}_{\{k_i>0\}}\lambda_i V(\vec{k},\vec{y}+e_{k_i}) + \sum_{j=1}^{J} \bar{\mu}_j V\big(\vec{k},[\vec{y}-e_j]^+\big) \quad (2) \\ &+ \big(1 - \sum_{i=1}^{I}[\mathbb{1}_{\{k_i=0\}}\gamma_i + \mathbb{1}_{\{k_i>0\}}(\theta_i+\lambda_i)] \\ &- \sum_{j=1}^{J}\bar{\mu}_j\big)V(\vec{k},\vec{y}) + \frac{2}{J(J-1)}\sum_{m=1}^{J}\sum_{n=m+1}^{J}|y_m - y_n|, \end{aligned}$$

for all $(\vec{k},\vec{y}) \in \mathcal{S}^{(1)} := \big(\mathcal{J}\cup\{0\}\big)^I \times \mathbb{N}^J$, and with $\bar{\mu}_j = \sum_{l=1}^{L} p_l\mu_{j,l}$. The third term of Equation 2 models an arrival of a job generated by client i, but in this case the job type does not matter. Similarly, the fourth term considers a job completion. Information on the number of type l jobs on server j is lost, but we still have information on the probability that a job is of type l (i.e., p_l) and the service rate of a type l job on server j (i.e., $\mu_{j,l}$). We use that information to calculate a weighted average for the service rate of a job on server j, with p_l as the weights.

Approximation 2 (APP2)
The second approximation is based on $\vec{x}$ and $\vec{y}$ and thus uses even less information than APP1. Information about the individual clients are now aggregated into the number of clients on server j. The dynamic programming operator is then given by

$$\begin{aligned} TV(\vec{x},\vec{y}) &= \big(I - \sum_{j=1}^{J} x_j\big)\bar{\gamma}\min_{a\in\mathcal{J}}\left\{V(\vec{x}+e_a,\vec{y})\right\} \\ &+ \sum_{j=1}^{J} x_j\bar{\theta}V(\vec{x}-e_j,\vec{y}) + \sum_{j=1}^{J} x_j\bar{\lambda}V(\vec{x},\vec{y}+e_j) \\ &+ \sum_{j=1}^{J}\bar{\mu}_j V\big(\vec{x},[\vec{y}-e_j]^+\big) \quad (3) \\ &+ \big(1 - \big(I-\sum_{j=1}^{J}x_j\big)\bar{\gamma} - \sum_{j=1}^{J}[x_j\bar{\theta} + x_j\bar{\lambda} + \bar{\mu}_j]\big)V(\vec{x},\vec{y}) \\ &+ \frac{2}{J(J-1)}\sum_{m=1}^{J}\sum_{n=m+1}^{J}|y_m - y_n|, \end{aligned}$$

for all $(\vec{x},\vec{y}) \in \mathcal{S}^{(2)} := \mathcal{I}^J \times \mathbb{N}^J$, and with $\bar{\gamma} = \frac{1}{I}\sum_{i=1}^{I}\gamma_i$, $\bar{\theta} = \frac{1}{\bar{\gamma}\cdot I}\sum_{i=1}^{I}\gamma_i\theta_i$, $\bar{\lambda} = \frac{1}{\bar{\gamma}\cdot I}\sum_{i=1}^{I}\gamma_i\lambda_i$, and $\bar{\mu}_j = \sum_{l=1}^{L} p_l\mu_{j,l}$. The first term in Equation (3) models the transition of a client from the off-state to the on-state. Each client switches from the off-state to the on-state with intensity $\bar{\gamma}$, and in total there are $I - \sum_{j=1}^{J} x_j$ clients that are in the off-state. The second term considers a change from the on-state to the off-state. Since there is no information that determines which clients are exactly in the on-state, we

calculate a weighted average for the rate to switch to the off-state by weighting the individual rates θ_i by their fraction $\gamma_i/(\sum_{i=1}^{I}\gamma_i)$. The same reasoning applies to the calculations of the other weighted averages. This means that we give higher weights to the parameter values for clients that turn faster to the on-state than other clients do. The third and fourth terms of Equation (3) consider an arrival and completion of a job, respectively. The remaining terms of the operator are due to uniformization and the cost function.

IV. Numerical results

In this section we evaluate the performance (denoted by g in Section II) of our algorithm extensively by numerical experiments. We do this by comparing our algorithm to three other commonly used algorithms; the round robin (RR) algorithm, a load based (LB) algorithm and a least connection (LC) algorithm. We generate many random instances for different parameter values and compare the performance of these algorithms. We first start by explaining the RR, LC, and LB algorithm.

The round robin (RR) algorithm does not use any information about the system and assigns a new session in a cyclic manner to the servers such that the i-th decision is assigned to server $(i \mod J)+1$. The least connection (LC) algorithm focuses on the number of connections $x_j = \sum_{i=1}^{I} \mathbb{1}_{\{k_i=j\}}$ for server $j = 1,\ldots,J$. Thus, the LC algorithm selects server $\arg\min_j\{x_j\}$. The load based (LB) algorithm makes a decision based on the load $y_j = \sum_{l=1}^{L} y_{lj}$ for server $j = 1,\ldots,J$. Hence, the algorithm selects server $\arg\min_j\{y_j\}$.

Note that the second approximation (APP2) uses as information both the vectors $\vec{x} = (x_1,\ldots,x_J)$ and $\vec{y} = (y_1,\ldots,y_J)$. The first approximation (APP1) uses more information and is based on the tuple $(\vec{k},\vec{y})$ instead of the client-aggregated information $(\vec{x},\vec{y})$. The optimal policy (OPT) has the most detailed information $(\vec{k},Y)$, in which the load is not aggregated over the different job types. Therefore, we expect that RR, LC, LB, APP2, APP1, and OPT has an increasing performance, since they use more information, respectively. For the cases of LC and LB this is not completely true, because the algorithms focus on different parts of the state space ($\vec{x}$ versus $\vec{y}$ as compared to $(\vec{x},\vec{y})$). As more information is used, the scalability of the algorithms deteriorates, since the computation times becomes longer. In the results we will discuss this trade-off between performance and computing times as more information is used.

Next, we describe the setup of the experiments. We look at 4 network topologies with I = 2, 3, 5, and 10 clients, respectively. In all cases we have J =2 servers and L = 2 or 3 job types. For each network topology we randomly generate 50 parameter settings. The parameters γ (the rate of becoming active), θ (the rate of becoming inactive), λ (the rate at which jobs are generated in the active state) are drawn uniformly from the I-th dimensional unit cube $[0.5,5.0]^I$, $[0.5,3.0]^I$, and $[1.0,10.0]^I$, respectively. When there are $L = 2$ types, an arriving job is of type 1 with probability $p_1 = 1-p_2$ uniformly generated from the interval $[0.1,0.9]$. In case of $L = 3$ types, we have that p_1 and p_2 (and $p_3 = 1-p_1-p_2$) are uniformly generated from the interval $[0.1,0.8]$, $[0.1,0.9-p_1]$, respectively. The service rates $\mu_{j,l}$ are uniformly generated from the interval $[p_l(\lambda_1+\cdots+\lambda_I), 2p_l(\lambda_1+\cdots+\lambda_I)]$ to ensure stability of the system for all policies. Thus, we evaluate $50\times 4 = 200$ different experiments.

Figure 1 shows the comparison of the different algorithms for the four network topologies. The algorithms are listed on the x-axis. The relative difference between two algorithms, say ALG_1 and ALG_2, is listed on the y-axis and is computed as

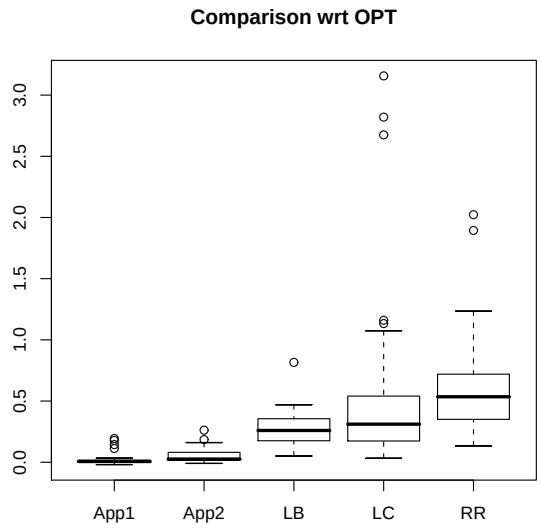
$$\frac{\text{performance}(\text{ALG}_1) - \text{performance}(\text{ALG}_2)}{\text{performance}(\text{ALG}_2)}.$$

Figure 1(a) shows the relative difference with respect to OPT (thus ALG_2 = OPT). Figures 1(b)–1(d) show the relative difference with respect to APP1, since the problem instances are too large to solve to optimality. In the boxplot, the thick line represents the median, surrounded by the 25% and the 75% quartile. This range is also called the IQR, the inter-quartile range. The whiskers represent the $1.5\cdot$ IQR range (cut off by the last point that falls into that range) in which most of the points fall. In case the points do not fall into this range, the points can be considered as outliers.

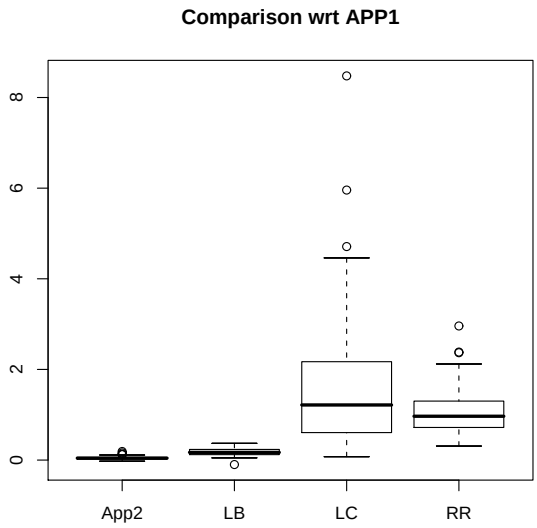
Figure 1(a) clearly shows that APP1 and APP2 consistently have a very good performance with little variance. The average deviation with respect to OPT is less than 2% and 5%, respectively, for APP1 and APP2. The LB and LC algorithms seem to have a comparable performance, but the LB algorithm has more consistent performance than the LC algorithm. Both these algorithms outperform the RR algorithm. Figure 1(b) shows the results for a bigger problem instance with three clients instead of two clients, and with three job types. For this problem instance, the optimal solution is already numerically intractable. Therefore, we compare the algorithms to APP1, since that uses the most information and is closest to the optimal policy. We have also done this for the last two topologies, since the problem instances are bigger in those cases. The results in Figures 1(b)–1(d) consistently show that APP1 and APP2 outperform the other algorithms. The LB algorithm seems to have reasonable performance as well when compared to the LC and RR algorithm.

For a fair comparison of the algorithms, we also need to study the complexity of the algorithms. The running time for the optimal policy was around 30 minutes, 2 days, 3 days for topology 1, 2 and 3, respectively. Topology 4 was practically beyond the reach of the optimal dynamic programming algorithm. APP1 and APP2 calculated the approximate policies within one minute overall. However, the difference in running time between APP1 and APP2 increases as the problem instances grow larger and larger. Hence, as one scales up the problem it could be more beneficial to switch from APP1 to APP2.

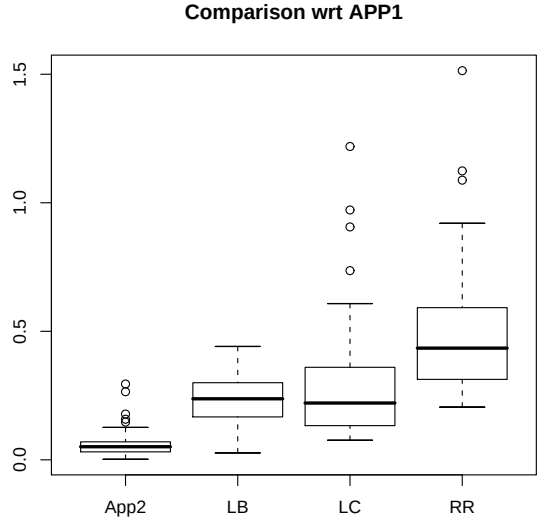
For an instance from topology 1, we look further into the optimal policy compared to the LB algorithm. The optimal policy, when a client switches from the off-state to the on-state, depends not only on the number of jobs on both servers, but also on the types of the jobs and whether the other client is in the on-state or off-state. The LB algorithm discards the latter



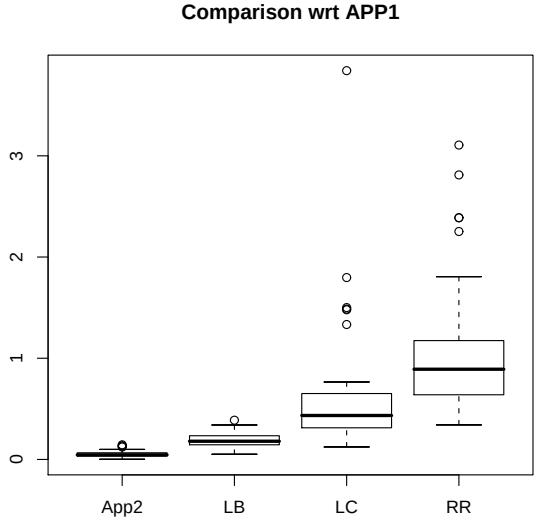
(a) Topology 1: 2 clients, 2 servers, 2 job types.



(b) Topology 2: 3 clients, 2 servers, 3 job types.



(c) Topology 3: 5 clients, 2 servers, 2 job types.



(d) Topology 4: 10 clients, 2 servers, 3 job types.

Fig. 1. Comparison of algorithms for different topologies.

two. If the LB algorithm would be optimal, then the policy would be to assign the client to, say, server 1 if the number of jobs on server 1 is less than (or equal to) the number of jobs on server 2. By looking at the optimal policy we see that the most important part to base the decision on is the number of jobs on the servers. Therefore, the LB algorithm performs well. However, in cases where the number of jobs on both server is approximately equal, more information is needed to assign the client optimally. In Figure 2 we make this point clear by showing the number of action conflicts that occur in the optimal policy by aggregating on the jobs. For example, the are 250 conflicts in the policy at 15 jobs on both servers. This means that the action taken using the optimal policy differs in 250 situations compared to the action taken by only looking at the number of jobs. For topologies different than topology 1, the number of jobs running on the servers remains the most important criterion to base decisions on. However, further numerical experiments show that the number of assigned sessions becomes more important as the length of a session increases; the longer the session, the more conflicting actions appear between the policy that uses both the number of sessions and jobs on the server compared to the LB algorithm. Furthermore, the number of sessions gets also more important when the arrival rate of jobs (λ) and the rate of becoming active (γ) increases, and when the service rate (μ) decreases.

The Markov decision problem is solved for exponentially

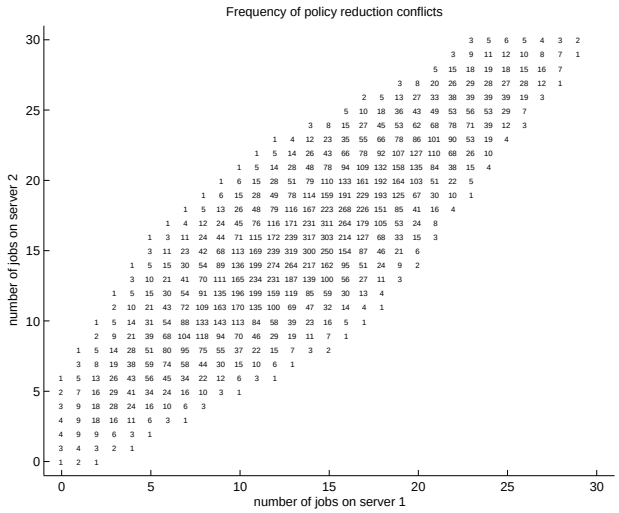


Fig. 2. The number of action conflicts when aggregating on the jobs.

Algorithm	HypExp	Unif	HypExp	Unif
	Topology 1		Topology 2	
OPT	0.000	0.000	-	-
APP1	0.028	0.018	0.000	0.000
APP2	0.055	0.058	0.060	0.076
LB	0.232	0.316	0.222	0.281
LC	0.355	0.579	1.734	1.858
RR	0.555	0.631	1.266	1.393
	Topology 3		Topology 4	
OPT	-	-	-	-
APP1	0.000	0.000	0.000	0.000
APP2	0.057	0.078	-0.013	-0.031
LB	0.256	0.347	0.131	0.183
LC	0.286	0.291	0.579	0.458
RR	0.451	0.614	0.949	1.109

TABLE I
COMPARISON FOR NON-EXPONENTIAL SERVICE DISTRIBUTIONS

distributed random variables. In the last part of this section, we study the impact on the performance of the algorithms when the required service time for a job has a hyperexponential, and an uniform distribution. In fact, the algorithms OPT, APP1, and APP2 are still based on the assumption of exponential random variables, but the policies derived using OPT, APP1, and APP2 are used in an environment with other distributions for the job service times. Therefore, we actually study the impact of the assumption of exponentially distributed service times.

For the hyperexponential distribution, we generate two service rates, say $\mu'_{j,l}$ and $\mu''_{j,l}$, uniformly from the interval $[p_l(\lambda_1 + \cdots + \lambda_I), 2p_l(\lambda_1 + \cdots + \lambda_I)]$. The probability p_1 that the service has rate $\mu'_{j,l}$ is generated uniformly from the interval $[0.1, 0.9]$, and the probability that the service rate is $\mu''_{j,l}$ is $p_2 = 1 - p_1$. For the algorithms OPT, APP1, and APP2 we assume exponentially distributed service times with parameter $\mu_{j,l} = p_1\mu'_{j,l} + p_2\mu''_{j,l}$. For the uniform distribution with parameters $a_{j,l}$ and $b_{j,l}$, we generate $a_{j,l} = 0.8/\mu_{j,l}$ and $b_{j,l} = 1.2/\mu_{j,l}$ with $\mu_{j,l}$, as before, uniformly chosen from the interval $[p_l(\lambda_1 + \cdots + \lambda_I), 2p_l(\lambda_1 + \cdots + \lambda_I)]$.

Table I shows the performance of the algorithms. For topology 1, the results are compared against the algorithm OPT, and the results of the last three topologies are compared against APP1 (due to the computational intractability of OPT for these topologies). Although the policy obtained from the OPT algorithm may not be the best possible policy in case of hyperexponentially and uniformly distributed service times, it still performs better than the other algorithms. Table I clearly shows that the algorithms OPT, APP1, and APP2 have very good performance even in cases with non-exponential service time distributions, despite the fact that the algorithms assume exponential distributions.

V. CONCLUSIONS AND DISCUSSION

We have formulated the problem of session-level load balancing as a Markov decision problem. For already small problem instances the Markov decision problem has shown the be numerically intractable due to the high dimensionality of the state space. Therefore, we have developed two approximations that are based on a state aggregation technique and consequently use less information for decision making. The first approximation aggregates information on the load, and the second additionally aggregates client information. In the numerical experiments, we have seen that both approximations outperform well-known algorithms that are frequently used in practice. The load-based (LB) algorithm performs reasonably well over a broad range of parameters. However, as the problem size grows, the gains achieved by the approximations grow larger against little additional computational complexity.

Interesting avenues for further research include the replacement of the relative value function by a value function with an approximation structure that is parameterized by a vector $\vec{r}$ and matrix R of low dimension. The advantage of this method is that it requires less storage space, since the number of parameters in $\tilde{V}$ is smaller than the number of states. However, a disadvantage is that it requires more time to compute the coefficients $\vec{r}$ and R, since one starts with an initial policy, estimates these parameters, finds a new policy, and continues until the policy cannot be improved anymore. Since the value function is approximated by $\tilde{V}$ it is also not guaranteed that a better policy is found in each step.

A second topic that requires more investigation is the weight factor that is used in the second approximation APP2. In APP2 the clients that become active faster have a higher weight in the estimation of $\bar{\gamma}$, $\bar{\theta}$ and $\bar{\lambda}$. We have also tried to use the fraction of time that client i is in the on-state. However, this approach seems to perform worse than the one we used in our APP2 algorithm. A better weight factor might improve the performance of the algorithm and warrants more study.

REFERENCES

[1] C. Koppaparu, *Load Balancing Servers, Firewalls and Caches*. John Wiley & Sons, 2002.
[2] T. Bourke, *Server Load Balancing*. O'Reilly Media, 2001.
[3] "http://www.icmgworld.com/corp/k2/k2.loadbal.asp."
[4] "http://msdn2.microsoft.com/en-us/ms972338.aspx."
[5] "http://www.systinet.com/doc/ssj-65/ssj/administration_load_balancing.html."
[6] "http://kb.linuxvirtualserver.org."
[7] M. Puterman, *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, Inc., 1994.