Dynamic Adaptation of User Migration Policies in Distributed Virtual Environments

David Vengerov
Oracle Labs, Oracle Corporation
501 Island Parkway
Belmont, CA 94002
david.vengerov@oracle.com

***Abstract*—A distributed virtual environment (DVE) consists of multiple network nodes (servers), each of which can host many users that consume CPU resources on that node and communicate with users on other nodes. Users can be dynamically migrated between the nodes, and the ultimate goal for the migration policy is to minimize the average system response time perceived by the users. In order to achieve this, the user migration policy should minimize network communication while balancing the load among the nodes so that CPU resources of the individual nodes are not overloaded. This paper considers a multi-player online game as an example of a DVE and presents an adaptive distributed user migration policy, which uses Reinforcement Learning to tune itself so as to minimize the average system response time perceived by the users. Performance of the self-tuning policy was compared on a simulator with the standard benchmark non-adaptive migration policy and with the optimal static user allocation policy in a variety of scenarios, and the self-tuning policy was shown to greatly outperform both benchmark policies, with performance difference increasing as the network became more overloaded.**

I. Introduction

Distributed virtual environments (DVEs) have become a major trend in distributed applications. These highly interactive systems simulate a virtual world where multiple users share the same scenario. DVE systems are currently used in many different applications such as civil and military distributed training, virtual shopping malls, collaborative design, and e-learning [10]. However, the most popular application domain for DVE systems is that of commercial multiplayer online games (MOG) environments. In a typical MOG users interact with the game and with each other by generating independent and short-lived tasks, which can run on any network node (computing server). Each node has its own scheduler that assigns CPUs to incoming tasks. The migration of users between the nodes in a MOG network is the subject of active research.

Because of the very large scale of the online game world and of DVEs in general, the policy for migrating users needs to be fully distributed so that no single node can be fully responsible for migrating users in the whole network. Since the users care about system response time the most, the objective for such policy would be to minimize the time-average response time of all tasks in the system. While it is easy to develop some fixed rules for migrating users in response to an overload of computing resources on some node or an overload of the internode bandwidth due to increased communication between some users, such rules would not constitute the *optimal* policy and could perform very poorly in some circumstances.

In general, the task of finding the optimal user migration policy requires solving a stochastic optimal control problem. The Reinforcement Learning (RL) theory [13] has been developed for solving this problem in the case when equations governing the system's dynamics are unknown and the policy can only learn from experience by interacting with the environment and observing some feedback signal. The standard RL theory deals with a single agent interacting with a fully observable environment. It is not a simple matter to apply this theory to a large-scale DVE, where the user communication pattern changes dynamically over time and each node does not have the full observability of the complete network.

The main contributions of this paper are:

1) A distributed reinforcement learning framework for DVEs, which learns to continually improve the user migration policy so as to minimize the expected future system response time when both the workload and the user communication pattern are changing stochastically.
2) Simulation experiments demonstrating that the proposed RL framework can maintain the same average task response time in a DVE as competing algorithms but using fewer resources (network servers).

II. Related Work

An adequate solution to the problem of migrating users away from the overloaded servers is crucial for achieving a good performance in a DVE, and many researchers have proposed solutions for this problem. One of the earlier and more fundamental works is [7], where the authors proposed to perform user migrations so as to minimize a linear combination of the imbalance in computational load among the servers and the network communication cost of the resulting user allocation. They showed that this problem is NP-complete and provide an iterative heuristic algorithm for solving it. The usefulness of their cost function was questioned in [9], where the authors showed through simulations that a linear combination of CPU load and bandwidth utilization does not have a good correlation with the average system response time, which increases nonlinearly as server CPU utilization becomes close to 100%. They proposed an algorithm that is triggered

when a server's CPU utilization rises above 99% and that migrates users to the server with the lowest CPU utilization in such a way that the resulting estimated CPU utilization of the original server is reduced to 90%. In a later work [10], the authors proposed a migration algorithm that distributes the extra workload of the original server among the least loaded servers in the system instead of migrating this workload to a single server.

The migration algorithms proposed in [9] and [10] are similar to most of the other migration algorithms we found in the literature, which are focused on balancing the resource utilization among the servers. For example, the authors in [11] used thresholds on CPU utilization and network utilization, and when either one of these thresholds is exceeded for a server, the server migrates enough users to the least loaded server in the neighborhood so as to reduce both its own CPU and network utilization below 90% of the thresholds used to initiate migration. In [16], the proposed migration policy is triggered when a server's CPU utilization rises above a predetermined threshold, and the users are then migrated to the neighboring server with the smallest migration cost if that server is not expected to get overloaded as a result of migration. In [1], the migration algorithm attempts to minimize the maximum resource utilization of the servers while at the same time minimizing the cost of migration, measured by the time spent on migration.

Some migration algorithms focus on balancing the number of users across the servers (e.g., [6]). A similar but a slightly more sophisticated migration algorithm is proposed in [3], where each server determines whether the cause of a perceived quality of service violation (measured in terms of the user response time) is client load or inter-server communication and triggers either load shedding (to the neighboring nodes with the smallest number of users) or load aggregation, respectively. The load aggregation is performed by splitting the virtual space into regions and making sure that each server is handling the workload only from the neighboring regions.

The works described above form a representative sample of the user migration algorithms found in the literature. We have not found any existing algorithms that try to formally optimize the expected future system response time when both the workload and the user communication pattern are changing stochastically. Hence, the distributed reinforcement learning approach proposed in this paper for tuning user migration policies, which is able to perform such an optimization, makes it possible to deploy DVEs that work faster and/or use fewer resources.

III. Reinforcement Learning Background

We first describe the "classical" single-agent RL theory. The *agent* in RL theory refers to the decision-making entity that attempts to learn the policy that maps observed state variables into action to be taken in each state. The agent's state evolves as a Markov chain conditional on its current state and the action taken in that state, forming a Markov Decision Processes (MDPs) described by a quadruple (S, A, R, T) consisting of:

- A finite set of states S
- A finite set of actions A
- A function $c : S \times A \times S \rightarrow \Re$ describing the feedback (cost) signal received by the agent after each step
- A state transition function $T : S \times A \rightarrow PD(S)$, which maps the agent's current state and action into the set of probability distributions over S.

At each time t, the agent observes the state $s_t \in S$ of the system, selects an action $a \in A$ and the system changes its state according to the probability distribution specified by T, which depends only on s_t and a_t. The agent then receives a real-valued cost signal $c(s_t, a_t, s_{t+1})$. The agent's objective is to find a stationary policy $\pi : S \rightarrow A$ that minimizes the average future cost per time step starting from any initial state.

For a stationary policy π, the average cost per time step is defined as:

$$\rho^\pi = \lim_{T\rightarrow\infty} \frac{1}{T} \sum_{t=0}^{T-1} c(s_t, \pi(s_t), s_{t+1}), \tag{1}$$

The optimal policy π^* from a class of policies Π is defined as $\pi^* = \underset{\pi\in\Pi}{\operatorname{argmin}} \rho^\pi$. The *cost-to-go* $C^\pi(s)$ of a state s under a policy π is defined as the expected sum of the differences of observed step costs minus the average cost for all states:

$$C^\pi(s) = E[\sum_{t=0}^{\infty} (c(s_t, \pi(s_t), s_{t+1}) - \rho^\pi) | s_0 = s]. \tag{2}$$

The above definition implies the following recursive relationship:

$$\begin{aligned} C^\pi(s_t) &= E[c(s_t, \pi(s_t), s_{t+1}) - \rho^\pi] \\ &\quad + E[\sum_{\tau=t+1}^{\infty} (c(s_\tau, \pi(s_\tau), s_{\tau+1}) - \rho^\pi)] \\ &= E[c(s_t, \pi(s_t), s_{t+1}) - \rho^\pi] + C^\pi(s_{t+1}), \end{aligned}$$

which can be re-written as

$$E[c(s_t, \pi(s_t), s_{t+1}) - \rho^\pi] + C^\pi(s_{t+1}) - C^\pi(s_t) = 0.$$

Defining

$$\delta_t = c(s_t, \pi(s_t), s_{t+1}) - \rho_t^\pi + C^\pi(s_{t+1}) - C^\pi(s_t), \tag{3}$$

the quantity δ_t can be interpreted as the "error" term, for which the above analysis implies $E[\delta_t] = 0$. Therefore, if one defines $\hat{C}^\pi(s)$ to be the approximation to $C^\pi(s)$, then if after transferring to state s_{t+1} from state s_t one observes $\delta_t > 0$ (due to a larger than expected $c(s_t, \pi(s_t), s_{t+1})$), then one can conclude that $\hat{C}^\pi(s_t)$ had probably underestimated the true value of $C^\pi(s_t)$ and should be adjusted upwards. Similarly, if one observes $\delta_t < 0$ (due to a smaller than expected $c(s_t, \pi(s_t), s_{t+1})$), then one can conclude that $\hat{C}^\pi(s_t)$ had probably overestimated the true value of $C^\pi(s_t)$ and should be adjusted downwards.

This forms the basis for a well-known procedure called *temporal difference* (TD) learning for iteratively approximating $C^\pi(s)$, which lies at the core of many reinforcement learning

methods such as Policy Iteration [5]. Its simplest form is called TD(0):

$$\hat{C}^{\pi}(s_t) \leftarrow \hat{C}^{\pi}(s_t) + \alpha_t \delta_t, \tag{4}$$

where α_t is the learning rate, and ρ_t^{π} is updated as follows:

$$\rho_t^{\pi} \leftarrow (1-\alpha_t)\rho_t^{\pi} + \alpha_t c(s_t, \pi(s_t), s_{t+1}). \tag{5}$$

In this fashion, for every state s, $\hat{C}^{\pi}(s)$ has been proven to converge [14] to the true value $C^{\pi}(s)$, which represents the cost-to-go from state s under the policy π.

The TD approach based on assigning a cost to each state becomes impractical when the state space becomes very large or continuous, since visits to any given state become very improbable. In this case, a function approximation architecture needs to be used in order to generalize the cost-to-go function across the neighboring states. The architectures that are linear in tunable parameters are easiest to implement, and in this paper we use such an architecture: $\hat{C}(s,p) = \sum_{i=1}^{M} p^i \phi^i(s)$, where p is a vector of tunable parameters and $\phi^i(s)$ are fixed non-negative basis functions. This architecture can be thought of as a classical 3-layer neural network where only the connection weights between the hidden layer (which consists of the basis functions $\phi^i(s)$) and the output layer (which consists of the single "answer" unit) are tuned. A more complex architecture for $\hat{C}(s,p)$ can instead be "plugged into" the distributed RL framework proposed in this paper, where each state dimension has its own scaling parameter and the shape of each basis function is controlled by parameters as well.

For any approximation architecture $\hat{C}(s,p)$, the parameter vector p can be updated using the general "backpropagation" principle used in neural networks, which consists of computing at each step the partial derivative of the observed squared error with respect to each parameter and then adjusting each parameter in the direction that minimizes the squared error:

$$\begin{aligned} p_{t+1}^i &= p_t^i + \alpha_t \frac{\partial}{\partial p^i} \delta_t^2 \\ &= p_t^i + \alpha_t \delta_t \frac{\partial}{\partial p^i} \hat{C}^{\pi}(s_t, p_t) \\ &= p_t^i + \alpha_t \delta_t \phi^i(s_t), \end{aligned} \tag{6}$$

where the average cost estimate ρ_t is updated as in equation (5). The notation p_t^i refers to the value of the parameter p^i at time t during the learning phase. Note that when $\delta_t > 0$, the value of $\hat{C}^{\pi}(s_t, p_t)$ should be increased (using the same logic as the one used for equation (4)), which is achieved by increasing the value of all components of the parameter vector p, with the increase of each component i being proportional to the sensitivity of $\hat{C}^{\pi}(s_t, p_t)$ to that component (which is given by $\frac{\partial}{\partial p^i}\hat{C}^{\pi}(s_t, p_t)$). Also note that when the simple approximation $\hat{C}^{\pi}(s,p) = p$ is used for some scalar parameter p, equation (6) reduces exactly to equation (4).

The iterative procedure in equation (6) is guaranteed to converge to the locally optimal parameter vector p^* (the one giving the best approximation to $C^{\pi}(s)$ in its neighborhood) if certain conditions are satisfied [14]. First, the underlying Markov chain of states encountered under policy π has to be irreducible and aperiodic (i.e., the system can transfer from any state to any other state using n, $n+1$, $n+2$, ... time steps for n greater than some value N) and the learning rate α_t has to satisfy $\sum_{t=0}^{\infty} \alpha_t = \infty$ and $\sum_{t=0}^{\infty} \alpha_t^2 < \infty$. For example, $\alpha_t = 1/t$ satisfies these conditions. The final condition for convergence of the iterative parameter tuning process in equation (6) is that the basis functions $\phi^i(s)$ have to be linearly independent and the states for update have to be sampled according to the steady-state distribution of the underlying Markov chain for the given policy π (which happens naturally when the states are updated along the path traversed by the agent).

IV. Distributed Gaming Environment

Before describing the details of the proposed distributed reinforcement learning approach for tuning user migration policies, we need to give a brief description of the simulation environment within which the learning will be happening. As an instance of a DVE we chose to simulate a distributed gaming environment. In such an environment, the state of each user is saved in a data object, and user-generated tasks access other data objects in the network (the tasks change the user's state and maybe also the states of some other players). Some data objects do not represent any users and instead just represent some information about the game.

Each user keeps track of LocalLinks (number of other users on its current node with whom the user is communicating) and GlobalLinks (number of users on the other nodes with whom the user is communicating). The KeyNode for each user is the node that contains the data objects with which the user is communicating most often. The user communication pattern is updated once every K time steps by having each user, with probability p_c, add a communication link with a randomly selected new user or remove an existing communication link. The KeyNode is changed with probability p_{key} (randomly selected from the list of all nodes in the system) once every K time steps. The simulation results in Section VII are given for the case when $p_c = 0.5$, $p_{key} = 0.01$, and $K = 10$. We tried perturbing these values, but the relative performance of considered user migration policies remained the same.

The tasks generated by users in a game environment are usually short lived, on the order of a few milliseconds, and each task requires a single CPU for execution. At every time step, the scheduler on each node checks if any CPUs are available, and if the number of available CPUs $n > 0$, then n tasks are scheduled. Each of the n tasks is chosen from the user task queues on that node in a round-robin fashion. We modeled the task execution time as a random variable whose mean increased if the user is not located on his/her KeyNode and as the proportion of GlobalLinks for this user increased. If a user starts interacting with more other users then these interactions are expected to result in new tasks that the user has to perform. Thus, we modeled the number of tasks generated by each user at each time step as a random variable (with a

Poisson or a Zipf distribution) whose mean increased as the total number of communication links for this user increased.

V. A Distributed Reinforcement Learning Framework

A. Choosing the Basis Functions

The choice of the function approximation architecture for generalizing the cost-to-go function across the neighboring states is one of the key decisions that needs to be made when applying RL to any large-scale problem. The basis functions $\phi^i(s)$ for the approximation architecture $\hat{C}^\pi(s,p) = \sum_{i=1}^{M} p^i \phi^i(s)$ should ideally be chosen to be sensitive to changes in each individual dimension of the state vector s. We follow [15] in using basis functions of the form $\phi^i(s) = \phi_1^i(s[1]) \cdot \phi_2^i(s[2]) \cdot ... \cdot \phi_n^i(s[n])$, where n is the dimension of the vector s and ϕ_j^i is either equal to ψ_j^{fall} (a "falling" function that is equal to 1 for $s[j] < s_{min}[j]$, falls linearly from 1 to 0 over the range $[s_{min}[j], s_{max}[j]]$ and stays at 0 for $s[j] > s_{max}[j]$) or ψ_j^{rise} (a "rising" function that is equal to 0 for $s[j] < s_{min}[j]$, rises linearly from 0 to 1 over the range $[s_{min}[j], s_{max}[j]]$ and stays at 1 for $s[j] > s_{max}[j]$). If each function ψ_j were to fall from 1 to 0 or rise from 0 to 1 as a step function at $s[j] = (s_{min}[j] + s_{max}[j])/2$, then ψ_j could be visualized as "boxes" that cover the space of possible values of s, so that whenever s belongs to box i, $\hat{C}^\pi(s,p) = p^i$. However, in the proposed "soft" neural network implementation, each "neuron" ϕ^i gets activated to the extent specified by $\phi^i(s)$, and the final output is a linear combination of parameters p^i and neuron activations $\phi^i(s)$.

The choice of the basis functions described above implies that there will be $M = 2^n$ different basis functions for an n-dimensional state vector. Since the number of the basis functions $\phi^i(s)$ increases exponentially with the dimension of the state vector s, a centralized learning algorithm that learns a single cost-to-go function for the whole network would be computationally infeasible: if the state of each network node is described by 4 variables (which is what we propose in Section V-B), then for a small network with 25 nodes, the algorithm would need to keep updating 2^{100} parameters.

In order to avoid this difficulty, we propose to use a distributed reinforcement learning framework, where each network node (agent) j learns its own cost-to-go function $\hat{C}_j(s_j, p_j)$ that predicts the expected future response time for all tasks it will be executing in the future based on its own parameter vector p_j.

B. Forming the State Vector

The RL theory was originally developed for Markov Decision Processes (MDPs), where the agent's state evolves as a Markov chain conditional on its current state and the action taken in that state. However, each agent (network node) has to make decisions in a Partially Observable Markov Decision Process (POMDP), where partial observability refers to the fact that no node can observe all relevant information for predicting the future evolution of its state (which depends on what is happening with the other network nodes and all users in the DVE).

A good overview of methods for learning the cost-to-go functions in POMDPs is given in [4]. Some researchers have reported that memoryless approaches, where the agent uses only the locally available observations, also give good results in practice. Since the goal of this paper is to demonstrate that RL can be used to find good user migration policies (that are better than any fixed migration rules) in DVEs rather than discovering the best possible algorithms to plug into the framework we propose, we will use a simple memoryless approach, where each network node uses only the locally available information for making the user migration decisions. That is, the agent simply assumes that it acts in a fully observable MDP, and the observation vector is treated as the state vector. The experimental results in Section VII show that even this simple approach gives very good results.

The accuracy of a node's cost-to-go function approximation architecture $\hat{C}(s,p)$ depends greatly on how useful the component variables of the vector s are for predicting the average response time of all tasks completed on that node in the near future. Therefore, one might be tempted to include all relevant variables into the state vector. However, if too many variables are used, the learning process will be very slow. Below is the list of variables that we chose to represent compactly the state of each node, with the expectation that the cost-to-go function $C(s)$ will be an increasing function in each of these variables.

- $s[1]$: The total number of communication links on the node, which is expected to be correlated with the rate at which tasks are generated on this node (users with more communication links are expected to generate more tasks)
- $s[2]$: The degree of nonlocality of the communication pattern of the users on this node (which we computed as the sum of GlobalLinks/TotalLinks for each user plus the number of users for whom this node is not the KeyNode, everything divided by the number of users on the node)
- $s[3]$: Number of current users on the node
- $s[4]$: Total number of queued tasks at the node

C. Making Migration Decisions

Assume that the approximate cost-to-go function $\hat{C}_j(s,p)$ for each node j has been tuned to some extent using equation (6) to reflect the expected future task response time on that node. We will now denote such a function as $\hat{C}_j(s_j, p_j)$ so as to see clearly which node's state vector and parameter vector are used to compute C as a linear combination of state features and parameters. Then, any particular user migration decision between two nodes i and j can be evaluated to see whether it would reduce the expected future average task response time on these nodes. More specifically, a user migration decision between nodes i and j is beneficial if

$$\hat{C}_i(s_i', p_i) + \hat{C}_j(s_j', p_j) < \hat{C}_i(s_i, p_i) + \hat{C}_j(s_j, p_j), \quad (7)$$

where s_i is the state of node i before the migration and s_i' is its state after migration. The *optimal* user migration decision

between nodes i and j would be the one that minimizes $\hat{C}_i(s'_i, p_i) + \hat{C}_j(s'_j, p_j)$.

In the experiments presented in Section VII, the user migration decisions were made once every 10 time steps either based on a fixed benchmark policy or based on the migration criterion described above. In the latter case, the migration algorithm performed the following steps:

1) Iterate through the list of all nodes in the network
2) Each node i iterates through its present users
3) For each user k, the node i iterates through the list of its neighboring nodes and computes B_{kj} – the difference between the right hand side and the left hand side of the migration criterion (7), which represents the benefit realized in the network if user k were to be migrated from node i to node j.
4) Let k^* and j^* be such that $B_{k^*j^*} >= B_{kj}$ for all k and j.
5) If $B_{k^*j^*} > 0$, then migrate user k^* from node i to node j^*. Otherwise, do nothing.

In reality, the above migration algorithm would be implemented asynchronously on each node, which can use any appropriate network protocol to establish a list of neighboring nodes that can potentially accept users from this node. Thus, the above algorithm can scale to any number of nodes in the network.

It is interesting to point out that the above approach to taking actions in each state allows one to use an online RL approach, where the cost-to-go function $C(s)$ is used to make user migration decisions while its parameters are being tuned. In contrast, the classical approach in equation (6) requires using a different (fixed) policy π for taking actions while the parametes of the cost-to-go function are being tuned. If an online RL approach is desired, then the classical approach is to learn future cost functions $Q(s, a)$ for state-action *pairs* using *Q-learning* [13]. While it is pretty straightforward to come up with a description of the state space in the user migration problem, it is much more difficult to come up with a description of the action space.

The action selection approach proposed in this section avoids the difficulty of defining the action space. It uses the insight that the state evolution in the distributed gaming environment described in Section IV (and in many other task-based DVEs) happens in two stages: first the impact of changing the state of each node is quickly observed, and then the state change at each node due to externally arriving tasks is observed at the next state reconfiguration opportunity. As a result, it is possible to reduce the noise in parameter updates by defining the state s_t in equation (6) to be the one observed *right after* implementing a user migration decision, while the state s_{t+1} (that is used to compute δ_t) is the one observed *right before* making the next user migration decision.

VI. Implementation Details

A. Benchmark policies

The distributed RL framework described in Section V was compared against several benchmark (non-adaptive) user migration policies that represented reasonable rules for this domain. We called the first benchmark the *SlowestUser* policy, since it selects on each node the user with the largest expected task execution time and tries to migrate it to a neighboring node where this user's tasks would be expected to complete sooner. If there are several neighboring nodes that satisfy this requirement, then the node with the smallest expected execution time is chosen. The expected task execution time for a user j on node k is monotonically related to the degree of non-locality of that user's communication pattern, which we can define as $L(j, k)$ = (ratio of global to total communication links the user j would have on node k) + $I(j, k)$, where $I(j, k)$ is an indicator function that is equal to 1 if node k is not the KeyNode for user j and is equal to 0 otherwise. For simplicity, the *SlowestUser* policy uses $L(j, k)$ in its computations in place of the difficult to compute expected task execution time.

We also implemented another benchmark policy, called the *LongestQueue* policy, which differs from the *SlowestUser* policy in only one respect: it considers for migration the user with the longest task queue. Our experiments showed that the average task response time of this policy was higher than that of the *SlowestUser* policy in all scenarios, and the difference between them increased as the average task generation rate increased for each user. This can be explained by the fact that by trying explicitly to minimize the task execution time, the *SlowestUser* policy increases the throughput of the system, which on average leads to a smaller backlog of tasks and hence a smaller average task response time.

The *SlowestUser* policy is an instance of the most popular class of user migration algorithms described in Section II, which migrate users so as to balance the CPU load in the network while trying to preserve the locality of user communication (if the CPU load is balanced across all nodes, then the node where the user's tasks can be executed the fastest is the node that contains most of the data objects with which the user is communicating). Because of its performance superiority over the *LongestQueue* policy, the *SlowestUser* is used as THE benchmark non-adaptive user migration policy. The final benchmark policy was the optimal static user allocation policy, which performed no user migrations while keeping an equal number of users on all nodes.

B. Parameter tuning for the adaptive policy

While the learning rate $\alpha_t = 1/t$ used in equation (6) works in theory, in practice it decreases too quickly and does not allow parameters to reach near-optimal values during short learning episodes (in practice, the agent should learn a good policy as quickly as possible). Moreover, the parameters for which the basis functions have very small values get updated much slower than the other parameters. In order to avoid these problems, in experiments presented in Section VII a separate learning rate is used for each parameter i: $\alpha_t^i = \alpha_0/[1 + (\sum_{\tau=0}^{t} \phi^i(s_\tau))/N]$, where $\alpha_0 = 0.2$ and $N = 50$.

The exact shapes of the basis functions $\phi^i(s)$ depend on the interval $[s_{min}[j], s_{max}[j]]$ chosen for each dimension j within which the corresponding functions ψ_j^{fall} and ψ_j^{rise} are

changing, as explained in Section V-A. Assuming that each node is allowed to have at most U_{max} users and there is a total of U_{TOT} users in the network, the widest possible ranges were selected for variables $s[1]$, $s[2]$ and $s[3]$, which were, correspondingly, $[0, U_{max} \cdot (U_{TOT} - 1)]$, $[0, 2]$ and U_{TOT}. The task of defining ranges for the variable $s[4]$ is more complicated, since there is no maximum value that it can take. Therefore, instead of using linear functions ψ_4^{fall} and ψ_4^{rise}, we decided to use $\psi_4^{fall}(x) = e^{-b \cdot x}$, where $b = -\ln(0.05)/10$, so that $\psi_4^{fall}(0) = 1$ and $\psi_4^{fall}(10) = 0.05$. The function $\psi_4^{rise}(x)$ was defined as $\psi_4^{rise}(x) = 1 - \psi_4^{fall}(x)$.

We tried updating the average cost per time step ρ_t for each node using the classical equation (5), where $c(s_t, \pi(s_t), s_{t+1})$ was computed as the total response time (waiting time plus execution time) for all tasks executed on that node since the last parameter update divided by the number of tasks executed during that period. However, we found that better results were obtained when ρ_t was computed in a more stable fashion as the total response time of all tasks executed since the beginning of the simulation divided by the number of tasks executed. Also, we found it necessary in equation (3) to multiply $c(s_t, \pi(s_t), s_{t+1}) - \rho_t$ by the number of tasks executed since the last parameter update so that each agent could learn to avoid states from which MANY tasks are expected to incur larger than average task response time. We also found it beneficial to start the parameter updating process only after the average cost ρ_t had stabilized somewhat, and so we executed the $SlowestUser$ policy for the first 30000 time steps in order to compute the initial value of ρ_t.

Parameters of each node's cost-to-go function were tuned for 570000 time steps, after each K time steps, right before making user migration decisions. At the end of the learning process, a testing phase of 200000 time steps would be executed, during which the user migration decisions would be performed using the migration criterion described in Section V-C based on $\hat{C}_j(s_j, p_{j,2})$ as the cost function for each node j. Ten trials were conducted for each experiment (which consisted of a learning phase and a testing phase), so that the standard deviation of the observed average task response time divided by the mean response time would be less than 1%. While 600000 time steps might seem like a long learning time, the tasks in a usual game environment are very short lived, on the order of a few milliseconds. If the average task duration is assumed to be 2 milliseconds, then 600000 time steps would be on the order of 600 seconds, which is not too long.

We have also evaluated a learning approach that is similar in spirit to the Policy Iteration (PI) RL approach developed by Ronald Howard [5]. In this approach, the *SlowestUser* benchmark policy would first be executed for 200000 time steps (policy iteration 0), and the parameter vector $p_{j,0}$ would be updated for each node j using the procedure described in equation (6). During the next 200000 time steps (policy iteration 1), the user migration decisions would be performed using the migration criterion (7) based on $\hat{C}_j(s_j, p_{j,0})$ as the cost function for each node j, while at the same time using equation (6) to update a new parameter vector $p_{j,1}$. During the next 200000 time steps (policy iteration 2), the user migration decisions would be performed using the migration criterion (7) based on $\hat{C}_j(s_j, p_{j,1})$ as the cost function for each node j, while at the same time using equation (6) to update a new parameter vector $p_{j,2}$. At the end of the learning process (which consisted of policy iterations 0, 1, and 2), a testing phase of 200000 time steps would be executed, during which the user migration decisions would be performed using the migration criterion (7) based on $\hat{C}_j(s_j, p_{j,2})$ as the cost function for each node j. This PI approach is supposed to be more stable than the online RL that we described earlier, but we found that as the average number of users per node increased in the network, the PI approach "blew up" earlier than the online RL approach (stopped being able to converge to user migration policies that allowed the network to process all tasks that were generated without the task queues blowing up). Therefore, we choose the online RL approach as the "reference" learning approach in Section VII.

VII. Simulation Experiments

Performance of the distributed RL approach described in Section V for learning user migration policies was evaluated by comparing it with the optimal static user allocation policy and with the *SlowestUser* benchmark migration policy described in Section VI-A.

The tasks for each user were generated (and placed in that user's task queue) using a Poisson stochastic process, where the task generation rate for each user j was a linear function $r_j = a_1 + \rho \cdot b_1 \cdot (LocalLinks_j + GlobalLinks_j)$, where $a_1 = 1$, $b_1 = 0.1$ and ρ is a variable parameter whose impact is described below. We have also simulated a heavy-tailed process, where the number of tasks generated per time step was $r_j \cdot k$ with probability $1/k$, where k is an integer randomly sampled from a uniform distribution on $\{1, 2, ..., 25\}$. The results for the heavy-tailed case were almost identical to those in the Poisson case when the process parameters were chosen so that the average number of tasks generated per time step was the same in each case. Therefore, we chose to present the results for only one case – Poisson task generation process.

When a task for user j was scheduled, its execution time was determined as $T_j = X + Y_j + Z_j$, where X is a random integer from the set $\{1, 2, 3\}$, $Y_j = 1$ with probability $GlobalLinks_j/(LocalLinks_j + GlobalLinks_j)$ and 0 otherwise (so that that tasks that need to access many remote objects take longer to execute), and $Z_j = 1$ if the user j is not on its KeyNode and is 0 otherwise.

Figure 1 shows the average task response times (over all tasks executed in the network during the testing period) of different policies for the simple scenario of 15 users migrating in a network of 5 nodes, with the task generation rate parameter ρ (described above) being plotted on the x-axis. The figure shows that the relative advantage of the user migration policy based on RL-tuned cost functions increases as the system load increases. This is a natural result for a queuing system, since the expected queue length grows to infinity if the task service

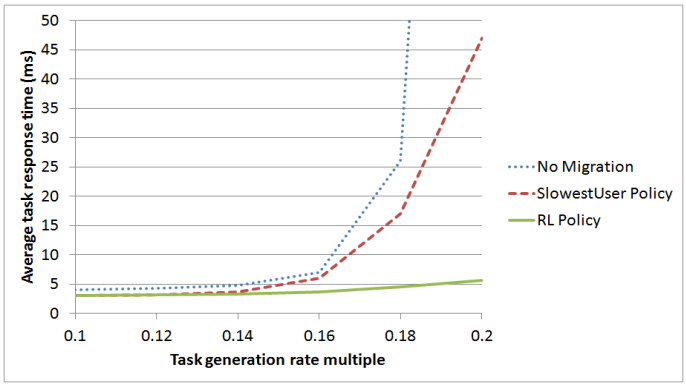

Fig. 1. Average task response time for 15 users on 5 nodes as a function of the task generation rate

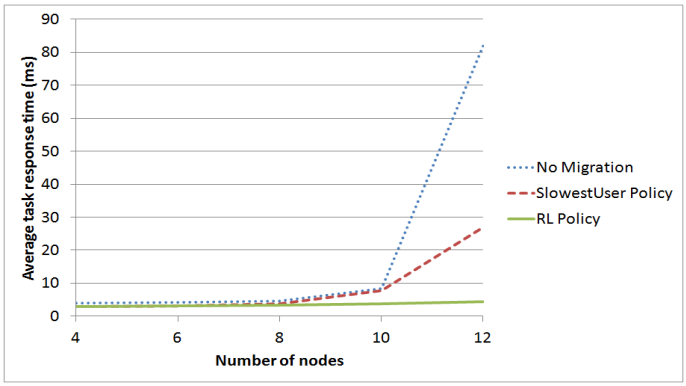

Fig. 2. Average task response time as a function of the number of nodes in the network if Users/Nodes=3

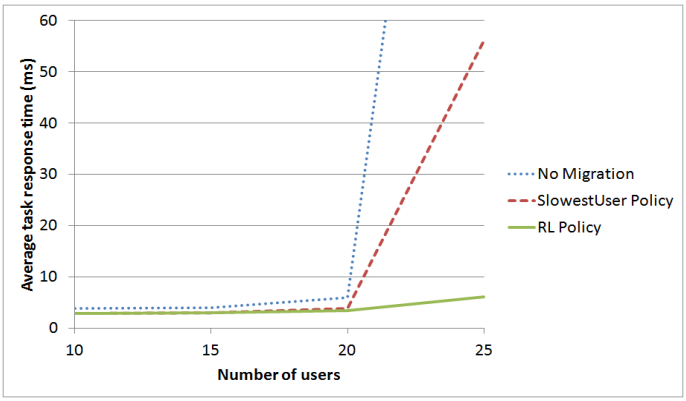

Fig. 3. Average task response time as a function of the number of users for 5 nodes.

```
(RULE 0) P[S,S,S,S] = -1013     SumA: +5364.21  LR: +0.0018
(RULE 1) P[S,S,S,L] = -1013     SumA: +2487.49  LR: +0.0039
(RULE 2) P[S,S,L,S] = -862      SumA: +4794.39  LR: +0.0021
(RULE 3) P[S,S,L,L] = -862      SumA: +2640.85  LR: +0.0037
(RULE 4) P[S,L,S,S] = -761      SumA: +6271.74  LR: +0.0016
(RULE 5) P[S,L,S,L] = -734      SumA: +4421.59  LR: +0.0022
(RULE 6) P[S,L,L,S] = -622      SumA: +6779.64  LR: +0.0015
(RULE 7) P[S,L,L,L] = -596      SumA: +4619.29  LR: +0.0021
(RULE 8) P[L,S,S,S] = -669      SumA: +1769.87  LR: +0.0055
(RULE 9) P[L,S,S,L] = -658      SumA: +957.55   LR: +0.0099
(RULE 10) P[L,S,L,S] = -572     SumA: +2365.11  LR: +0.0041
(RULE 11) P[L,S,L,L] = -553     SumA: +1307.70  LR: +0.0074
(RULE 12) P[L,L,S,S] = -542     SumA: +2109.68  LR: +0.0046
(RULE 13) P[L,L,S,L] = -450     SumA: +1746.36  LR: +0.0056
(RULE 14) P[L,L,L,S] = -342     SumA: +4016.36  LR: +0.0025
(RULE 15) P[L,L,L,L] = -331     SumA: +2347.16  LR: +0.0042
```

Fig. 4. Parameter values learned by RL for 12 nodes, 36 users, and task rate = 0.1

rate becomes less than or equal to the task arrival rate due to suboptimal scheduling/load balancing decisions. It is worth noting that the RL-based policy migrated during the testing period only 0.3% of tasks (that were in the task queue of the user being migrated) in the low load scenario and 6% of tasks in the high load scenario, which shows that a small amount of smart migration is sufficient to obtain large benefits in DVEs.

Figure 2 plots the average task response time for different policies as a function of the total number of nodes in the network while the average number of users per node was kept at 3 and the tasks for each user were generated using the Poisson process with the ρ parameter fixed at 0.1. This figure shows that the average task response time increases for all policies as the total number of nodes increases. This is so because the average task execution time for each user increases when the number of network node increases, since each user's chance of being located on its KeyNode decreases, its expected ratio of GlobalLinks to LocalLinks increases, and the task generation rate $r_j = a_1 + \rho \cdot b_1 \cdot (LocalLinks_j + GlobalLinks_j)$ increases.

Figure 3 plots the average task response time for different policies as a function of the number of users in our test network of 5 nodes. As expected, as the number of users increases, the average task response time for all policies increases because more tasks need to be processed on each node. The RL policy scales much better than the other two policies because it learns to migrate the users appropriately and keep the computational load on each node to a minimum.

In order to get a better idea of what the RL policy has learned, Figure 4 shows the final parameter values at the end of the learning process for the case corresponding to the last data point in Figure 2, where the network consisted of 12 nodes and 36 users, with the task generation rate parameter of 0.1. "SumA" and "LR" for each rule i are, correspondingly, $\sum_{\tau=0}^{t} \phi^i(s_\tau)$ and α_t^i, whose computation was explained at the beginning of Section VI-B. The parameter values are all negative because the average cost per time step in our implementation adapts slower than the parameter values (and hence the policy for making migrations), implying that the difference $c(s_t, \pi(s_t), s_{t+1}) - \rho_t$ in equation (3) is always negative (initially ρ_t is pretty large because the $SlowestUser$ policy is used for the first 30000 time steps, as was described in Section VI-B). As one can see from Figure 4, the RL policy learns that a smaller average cost per time step will be obtained in states where each state variable $s[j]$ is small and hence the neurons $\phi^i(s)$ composed mostly of "falling" functions are activated to a larger extent than those composed mostly of

the "rising" functions, as was explained in Section V-A. This corresponds to the intuition stated in Section V-B about the cost function being increasing in each state variable.

Finally, we would like to point out that if the task generation rate multiple is reduced below the one we used in our experiments and if it is assumed to scale slower as the total number of communication links for each user increases, then all considered user migration policies will be able to keep the average task response time stable when more nodes and users are present in the network.

VIII. Conclusions

This paper presented an adaptive distributed and scalable user migration framework for a distributed virtual environment, where each network node uses Reinforcement Learning to tune its migration policy so as to minimize the average task response time in the whole network. During experimental evaluation, the proposed framework has outperformed all considered benchmark policies, with performance difference increasing as the network became more overloaded. The distributed reinforcement learning framework for tuning user migration policies presented in this paper is very general and can apply to many other DVEs besides the one considered in this paper, with a virtual machine (VM) migration in a data center being another very important potential application area.

References

[1] L. Chen, C.-L. Wang, F. Lau, "Process Reassignment with Reduced Migration Cost in Grid Load Rebalancing," In Proceedings of the 17th International Heterogeneity in Computing Workshop (HCW'08), April 14, 2008.

[2] D. P. Bertsekas and J. N. Tsitsiklis. *Dynamic Programming and Optimal Control*. Volume 2. Athena Scientific, 2007.

[3] J. Chen, B. Wu, M. Delap, B. Knutsson, H. Lu, C. Amza, "Locality Aware Dynamic Load Management for Massively Multiplayer Games," In Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming, pp. 289-300, June 15-17, 2005.

[4] S. W. Hasinoff. "Reinforcement Learning for Problems with Hidden State," Technical Report, University of Toronto, Department of Computer Science, 2002.

[5] R. A. Howard. *Dynamic Programming and Markov Processes*. John Wiley & Sons, Inc., New York, 1960.

[6] K. Lee, D. Lee, "A Scalable Dynamic Load Distribution Scheme for Multi-Server Distributed Virtual Environment Systems with Highly-Skewed User Distribution," In Proceedings of the ACM symposium on Virtual reality software and technology, pp. 160-168, 2003.

[7] J. C. S. Lui, M. F. Chan, "An Efficient Partitioning Algorithm for Distributed Virtual Environment Systems," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 13 , Issue 3, pp. 193-211, March 2002.

[8] A. K. McCallum. *Reinforcement Learning with Selective Perception and Hidden State*, PhD. Thesis, University of Rochester, Department of Computer Science, 1995.

[9] P. Morillo, J. M. Orduna, M. Fernandez, J. Duato, "Improving the Performance of Distributed Virtual Environment Systems." *IEEE Transactions on Parallel and Distributed Systems*, Vol. 16, No. 7, pp. 637-649, 2005.

[10] P. Morillo, S. Rueda, J. M. Orduna, J. Duato, "A Latency-Aware Partitioning Method for Distributed Virtual Environment Systems." *IEEE Transactions on Parallel and Distributed Systems*. Vol. 18, No. 9, pp. 1215 - 1226, 2007.

[11] B. Ng, R.W.H. Lau, A. Si, F.W.B. Li, "Multi-server support for large scale distributed virtual environments," *IEEE transactions on multimedia*, Vol. 7, Issue 6, pp. 1054-1065, 2005.

[12] W. Ning, C. Hong-ming, J. Li-hong, "A Dynamical Adjustment Partitioning Algorithm for Distributed Virtual Environment Systems," In Proceedings of The 7th ACM SIGGRAPH International Conference on Virtual-Reality Continuum and its Applications in Industry, Session: Volume graphics, Article No. 19, 2008.

[13] R. Sutton, A. G. Barto. *Reinforcement Learning: An Introduction.* MIT Press, 1998.

[14] J. N. Tsitsiklis and B. Van Roy, "Average Cost Temporal-Difference Learning," Technical Report LIDS-P-2390, MIT Laboratory for Information and Decision Systems, 1997.

[15] D. Vengerov, "A Reinforcement Learning Framework for Utility-Based Scheduling in Resource-Constrained Systems." *Future Generation Computer Systems*, Volume 25, Issue 7, pp. 728-736, July 2009.

[16] T. Wang, C.-L. Wang, F.C. Lau, "An Architecture to Support Scalable Distributed Virtual Environment Systems on Grid," *The Journal of Supercomputing*, Vol. 36, No. 3, pp. 249-264, June 2006.