Reinforcement learning based schemes to manage client activities in large distributed control systems

Y. Gao,[*] J. Chen, and T. Robertazzi
Stony Brook University, Stony Brook, New York 11794, USA

K. A. Brown
Brookhaven National Laboratory, Upton, New York 11973, USA

(Received 15 June 2018; published 2 January 2019)

Large distributed control systems typically can be modeled by a hierarchical structure with two physical layers: console level computers (CLCs) layer and front end computers (FECs) layer. The control system of the Relativistic Heavy Ion Collider (RHIC) at Brookhaven National Laboratory (BNL) consists of more than 500 FECs, each acting as a server providing services to a large number of clients. Hence the interactions between the server and its clients become crucial to the overall system performance. There are different scenarios of the interactions. For instance, there are cases where the server has a limited processing ability and is queried by a large number of clients. Such cases can put a bottleneck in the system, as heavy traffic can slow down or even crash a system, making it momentarily unresponsive. Also, there are cases where the server has adequate ability to process all the traffic from its clients. We pursue different goals in those cases. For the first case, we would like to manage clients' activities so that their requests are processed by the server as much as possible and the server remains operational. For the second case, we would like to explore an operation point at which the server's resources get utilized efficiently. Moreover, we consider a real-world time constraint to the above case. The time constraint states that clients expect the responses from the server within a time window. In this work, we analyze those cases from a game theory perspective. We model the underlying interactions as a repeated game between clients, which is carried out in discrete time slots. For clients' activity management, we apply a reinforcement learning procedure as a baseline to regulate clients' behaviors. Then we propose a memory scheme to improve its performance. Next, depending on different scenarios, we design corresponding reward functions to stimulate clients in a proper way so that they can learn to optimize different goals. Through extensive simulations, we show that first, the memory structure improves the learning ability of the baseline procedure significantly. Second, by applying appropriate reward functions, clients' activities can be effectively managed to achieve different optimization goals.

DOI: 10.1103/PhysRevAccelBeams.22.014601

I. INTRODUCTION

The control system of the Relativistic Heavy Ion Collider (RHIC) at BNL is a large distributed discrete system. It provides operational interfaces to the collider and injection beam lines [1]. The architecture consists of two hierarchical physical layers: console level computers (CLCs) layer and front end computers (FECs) layer, as shown in Fig. 1. The console level is the upper layer of the control system hierarchy, which consists of operator consoles, physicist workstations and server processors that provide shared files, database, and general computing resources. The front end system contains more than 500 FECs, running on VxWorks™ real-time operating system. Each of them consists of a VME chassis with a single-board computer,[1] network connection, and I/O modules. FECs are distributed around 38 locations, including the control center, service buildings and 18 equipment alcoves accessible only via the ring tunnel. Along with data links and hardware modules, they are the control systems' interface to accelerator devices.

One of the most fundamental concepts in RHIC control system is the accelerator device object (ADO) [1]. ADOs are instances of C++ or Java classes which abstract

[*]ygao@bnl.gov

[1]They can have different processor architectures, e.g., POWER3E, MV2100, MV3100, XILINX, etc.

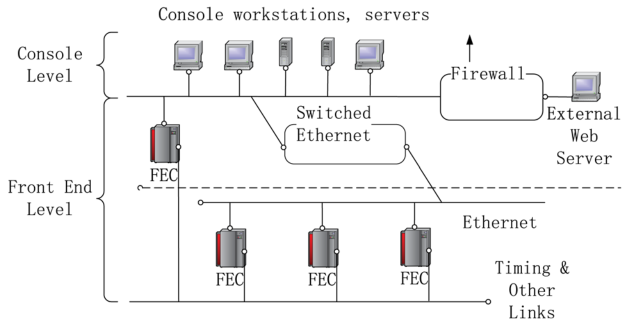


FIG. 1. RHIC system hardware architecture.

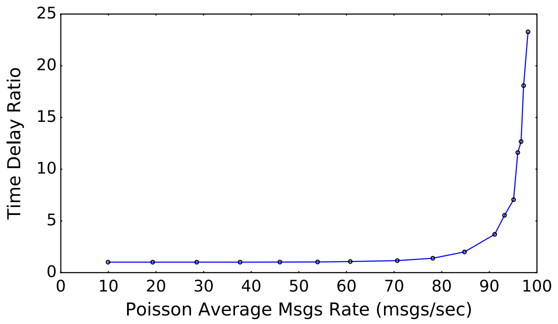


FIG. 2. Illustration of the client-server problem.

features from underlying control hardware into a collection of collider control points known as parameters, and each parameter can possess one or more properties to better describe characteristics of devices. The number of parameters and names of parameters are determined by ADO designers to meet the needs of the system. The most important ADO class methods for device control are the *set()* and *get()* methods. They are processed by the ADO that acts as the interface to device drivers in order to access control hardware. The collider is controlled by users or applications which *sets* and *gets* the parameters in instances of these classes using a suite of interface routines.

There is a fundamental performance issue in the front end system, where every FEC acts as a server which holds different kinds of ADOs, providing services to a potentially large number of clients. Depending on the traffic load on the server, the system exhibits different behaviors. For example, when the number of clients in an FEC reaches its limit, the system slows down or even crashes. When the system crashes, all current applications' communications are lost and it takes time to restore them. In other cases where the server's traffic is moderate, there may exist a working load range where the server works very efficiently.

In the current system, working load[2] is balanced[3] to prevent heavy traffic from crashing FECs. Another way of dealing with heavy traffic is to use what is called a reflected server (basically like a proxy server) that is built on a more robust and higher performance Linux system. However, those approaches do not eliminate the fundamental design limitation of the system.

In this work, we consider this practical issue from a different perspective. We analyze it quantitatively using game theory. Figure 2 shows the results of an experiment conducted on one of the FECs, which demonstrates how its performance changes with its traffic load. In this example, the arrival procedure of clients' requests is represented by a Poisson process [2] with various message rates. For each of those different arrival rates, we measure the ratio of the time spent by the server[4] to process requests between the case where the server has a certain message arrival rate and the case where the server has no arrival messages. The result indicates how well the system behaves for different server load. We can see that in the early stages, the curve is relatively flat, which means that the server's utilization rate does not grow much with the increasing message rate. On the contrary, as shown in the later part of the curve, when the number of clients approaches the server's capacity,[5] the performance of the system deteriorates dramatically.

This clearly demonstrates how the interactions between clients and the server is critical to the overall system performance. In this paper, we mainly study two possible scenarios in the system. As shown in Fig. 2, first, we focus on the later section of the curve. We study the case where the server does not have enough resources to process all the incoming requests. Specifically, the server's total capacity is some proportion of the total amount of traffic. The goal in this case is to adjust clients' behaviors in a distributed way so that the server does not crash and the server's throughput is maximized. Second, we focus on the front part of the curve, where the server has enough resources to process all the requests. The goal in this case is to guide the clients to explore around in their strategy space, so that their combination of strategies keeps the server working at an efficient operation point.[6] We propose management schemes following both centralized and distributed methodologies. Additionally, we consider the temporal dependencies in the client communications. The clients have a time constraint for when to expect responses from the server.

[2]Tasks are processed by FECs according to their priorities, higher priority tasks are processed earlier.

[3]If an FEC has many clients, the data size transmitted each time is relatively small. On the other hand, if the data size is large, then that FEC tends to have fewer clients. In other cases where the FECs already have heavy traffic, working load are split to other FECs.

[4]From now on, we will use the word "server" and "FEC" interchangeably.

[5]Here we assume one client only sends one request at each time. Depending on the size of clients' requests and hardware specifications of the server, the maximum number of clients the server can hold varies, in this example it is around 98.

[6]We will define the server's working efficiency more formally in the later section. Intuitively, it represents how much amount of traffic the server can process per unit time.

In general, we model the basic interactions as a repeated game between clients, and formulate it as an integer programming problem.[7] In our solution, we adopt a reinforcement learning procedure as an underlying routine, and propose a memory scheme to improve its learning ability. Next, depending on different scenarios, we coordinate the reward mechanisms to correctly encourage the clients, so that they can learn appropriate strategies to optimize different goals. Through extensive simulations, we prove that first, the memory scheme boosts the learning ability of the reinforcement procedure greatly, especially in a dynamic environment. Second, the activity management schemes can properly adjust clients' behaviors, so that different optimization goals are achieved.

The motivation of this work can be seen from the following point of view. First, since currently the performance limitation mentioned above does not affect the system significantly, there is no need to change the control system's infrastructure for this reason. However, analyzing it quantitatively brings insights into this issue, which also impacts how distributed control systems will be built in the future. Second, the accelerator control systems are designed to be highly reliable and should have a fast recovery in the event of some disruptions in operation, improving the dependability of the systems is crucial. This work aims to focus on methods to increase dependability of client-server communications in the systems.

The main contributions of this work can be summarized as follows. First, we take a game theory approach along with integer programming formulation to study realistic scenarios in accelerator control systems. Second, we introduce a reinforcement learning procedure as a basic routine to regulate clients' behaviors, and provide a way to enhance its performance. Third, we propose several client activity management schemes to accomplish various optimization goals in the system.

The rest of this work is organized as follows: Sec. II introduces the underlying game model, a literature review on game dynamics, as well as other basic assumptions and definitions. Section III analyzes the scenario where the server does not have enough resources to hold all clients' traffic and proposes a memory based activity management scheme. Section IV studies the scenario where the server has adequate resources to process all clients' traffic, and proposes both centralized and distributed activity management schemes. Moreover, a real-world time constraint and corresponding activity management schemes are also discussed in Sec. IV. Section V validates the correctness and evaluates the performance of the proposed schemes. Finally, Sec. VI summarizes this work.

[7]An integer programming problem is a mathematical optimization or feasibility program in which some or all of the variables are restricted to be integers.

II. PRELIMINARIES

In this section, we present some basic assumptions and definitions. First, the game theory model of the system is given, followed by a literature review about game dynamics. Next, to facilitate the illustration in the simulation section, we briefly introduce the Markov decision process (MDP) and a reinforcement learning procedure known as Q-learning [3].

A. Game theory model

Since in our system, the information on FECs are all different and FECs usually do not share information between each other, clients need to talk to a single FEC each time to get a particular kind of information. We build the model for single server serving multiple clients, then it can be applied to all the communication cases in the system.

We model the basic interactions between clients as a repeated game [4], which is played over discrete time slots among n clients[8] (players). During each time slot, a stage game is played in which all n clients simultaneously decide whether to *Send(S)* or *Hold(H)* their traffic to a server. Depending on the result of the stage game as a consequence of clients' actions, payoffs will be assigned to each client.

For the scenario where the server does not have enough resources to process all clients' requests, payoffs are assigned in the following way: At any time t, if the traffic coming from clients (L_i for client i) exceeds the server's capacity C_t (which stands for the capacity value at time t), then the server is crashed and clients get punishment payoff $-c$. Otherwise, they get payoff 1 as the profit of a successful traffic transmission. The payoff function for client i can be expressed as:

$$u_i(L_i, a_i) = \begin{cases} 1 & \text{if the server is alive} \\ -c & \text{if the server crashes} \end{cases} \tag{1}$$

where a_i is client i's action, with value 1 standing for action *Send* and 0 for *Hold*.

Table I summarizes these notations.

For the scenario where the server has enough resources to process all the incoming traffic and the time constraint case, the payoff function will be introduced in details in the later section.

B. Game dynamics

Game dynamics study how a game evolves when players interact with each other repeatedly. Learning procedures have been developed to achieve some overall optimization goals or convergence to the game's equilibrium.

[8]Since in our system, FECs do not share information between each other, this allows us to model each FEC separately by using the same game model.

TABLE I. Notations.

Notation	Definition
$\boldsymbol{N}, \lvert \boldsymbol{N} \rvert = n$	Set of clients
$\boldsymbol{A_i} = \{S, H\}$	Action set for client i
$L_i,\ 0 < L_i \leq L_{\mathrm{MAX}}$	Amount of traffic client i possesses
1	Benefit of a successful traffic transmission
C_t	Server's capacity at time t
$-c, c > 0$	Cost of server crash
u_i	Payoff function for client i

In work [5], the authors proposed a multiagent learning algorithm through gradient ascent method, and showed that in the simple setting of two-player, two-action repeated general-sum games, it either leads the agents to play a Nash equilibrium, or leads the agents' payoffs to Nash equilibrium payoffs. Work [6] extended [5] by introducing a varying learning rate to "win or learn fast," and proved convergence in the same game settings as [5]. Work [7] generalized the convergence results of [5] to games with more than two actions, and proved all clients' regrets are bounded by a constant number. Work [8] proposed a discrete procedure which achieves the no-regret result in work [7] and part of the convergence result in work [6], and proved convergence in two-player, two-action games.

There are also many works that study game dynamics for convergence to equilibrium. Work [9] introduced a hypothesis testing procedure in which, the joint mixed strategy profiles are within distance ϵ of the set of Nash equilibria in a fraction of at least $1 - \epsilon$ of time, though almost sure convergence is not achieved. Work [10] proposed the calibrated learning dynamics leading to correlated equilibria, in which every player computes "calibrated forecasts" on the behavior of the other players, and then plays a best reply to these forecasts. Work [11] offered a class of adaptive procedures called "calibrated smooth fictitious play", which guaranteed almost sure convergence to the set of correlated approximate equilibria. Another interesting line of works [12,13] proposed a simple adaptive learning procedure: "regret-matching." It is a discrete algorithm which does not require sophisticated updating or prediction. The procedure has the property that it requires neither global information about the game nor the observation of other opponents' actions, and converges to the game's set of correlated equilibria [4].

C. Markov decision process

Markov decision process (MDP) [14] is a classical formalization of sequential decision making. It is a straightforward formulation of the problems which focus on how to learn from interactions to achieve a goal. The actions in MDPs influence not only immediate rewards, but also subsequent states through future rewards. Future rewards are usually discounted by a discount factor γ $(0 \leq \gamma \leq 1)$. In general, the learner or decision maker is called the agent, and the thing it interacts with is called the environment.

More specifically, the agent and environment interact in discrete time slots. At each time step t, the agent makes an action $a_t \in \boldsymbol{A}(s)$ based on the environment's state $S_t \in \boldsymbol{S}$. At the next time step, the agent receives a reward $R_{t+1} \in \boldsymbol{R} \subset \mathbb{R}$, and the environment moves to the next state S_{t+1}. Hence, an MDP can be expressed as a sequence or trajectory: $S_0, A_0, R_1, S_1, A_1, R_2, \cdots$.

According to the discounting concept, at each time step t, the agent tries to select actions so that the sum of the discounted rewards it receives over the future periods is maximized. In particular, it chooses a_t at time t to maximize the expected discounted return:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2}$$

where $0 \leq \gamma \leq 1$ is the discount rate.

The discount rate represents how much the agent values the future states: A reward received k steps later is only worth γ^{k-1} proportion of what it would be worth if it were received immediately. In the extreme case of $\gamma = 0$, the agent is "myopic" in the sense that it only concerns the immediate reward. In general, "myopic" behavior can reduce access to future rewards resulting in a reduced return. As γ approaches 1, the agent takes future reward into account more strongly, it becomes more "farsighted."

D. Q-learning

MDPs can be solved by reinforcement learning algorithms, such as Q-learning [3], which belongs to the family of temporal-difference (TD) learning [14]. Q-learning is a procedure that tries to optimize Q values. The Q values $Q(S_t, A_t)$ at time t are updated as:

$$Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \tag{3}$$

where $0 < \alpha \leq 1$ is the learning rate, which determines the degree to which newly learned information overrides old information.

Q values approximate the optimal values of state-action pairs.[9] Q values can be iteratively computed by Eq. (3), and the more times they are updated the more accurate their approximations are. Intuitively, Q values measure the quality of state-action pairs. The control policy derived from Q-learning will be choosing the action that maximizes the Q value for each given state, which is a direct

[9]The optimal value of a state-action pair (S, A) is the total expected return for taking action A in state S and thereafter following an optimal policy.

Algorithm 1. Q-learning (off-policy TD control) for estimating $\pi \approx \pi^*$.

```
1: Initialize $Q(s,a)$, for all $s \in \mathbf{S}, a \in \mathbf{A}$, arbitrarily, and
   $Q(terminal\text{-}state, \cdot) = 0$
2: for each episode do
3:   Initialize $S$
4:   while $S$ is not terminal-state do
5:     Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)
6:     Take action $A$, observe $R, S'$
7:     $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma \max_a Q(S',a) - Q(S,A)]$
8:     $S \leftarrow S'$
9:   end while
10: end for
```

approximation of optimal policies. A general Q-learning control scheme [14] is shown in Algorithm 1, where ϵ-greedy policy means that most of the time the actions that maximize Q values are selected, but with probability ϵ actions are selected randomly.

III. TRAFFIC THROUGHPUT OPTIMIZATION

In this section, we study the problem of traffic throughput optimization. Specially, we consider the scenario where the server does not have enough capacity to hold all the clients' traffic, and we want to regulate clients' behaviors so that the server does not crash and the server's throughput is maximized. Moreover, in our system, there are asynchronous processes residing on FECs. Those processes will share FECs' resources when their required information is updated by FECs, resulting in a varying server capacity circumstance. One difficulty with this scenario is how to regulate clients' behaviors so that they can learn the server's limitations and adjust their strategies properly.

A. Problem formulation

We use the same notations as shown in Table I. As specified by the game model in the previous section, suppose the stage game is repeatedly played among n clients for T periods. The server's capacity at time t, C_t, is a portion of the clients' total traffic. Denote as a_i^t the action played by client i at time t. Our goal is to safely and efficiently route clients' traffic so that the server's throughput is maximized:

$$\textbf{Max:}\ \sum_{t=1}^{T}\sum_{i=1}^{n} a_i^t L_i \tag{4}$$

Subject to

$$\sum_{i=1}^{n} a_i^t L_i \le C_t, \quad \forall\ t = 1,2,\ldots,T \tag{5}$$

$$C_t < \sum_{i=1}^{n} L_i, \quad \forall\ t = 1,2,\ldots,T \tag{6}$$

$$a_i^t \in \{0,1\}, \quad \forall\ i = 1,2,\ldots,n, \quad \forall\ t = 1,2,\ldots,T \tag{7}$$

$$L_i \in (0, L_{\mathrm{MAX}}], \quad \forall\ i = 1,2,\ldots,n \tag{8}$$

where constraint (5) restricts that the server does not crash during the game. Constraint (6) indicates that the server does not have enough resources to hold all clients' traffic. Constraint (7) specifies that clients' actions are binary variables, where 1 stands for action *Send*, and 0 for *Hold*. Constraint (8) states that the maximum traffic load a client can have is greater than 0 and below a threshold $L_{\mathrm{MAX}} > 0$.

Note that a special case of the optimization problem above is to keep the server's capacity constant all the time, then the problem is equivalent to the Knapsack problem, which is a well-known NP-hard problem (see, for example, [15]). This reduction along with the dynamic aspect of the problem make it very difficult to seek any optimal solution directly. In the remaining parts of this section, we propose a heuristic scheme to address it.

B. A regret-based learning procedure

In this part, we describe the discrete adaptive learning procedure proposed in work [12,13] which leads to the game's set of correlated equilibria. This procedure is used as the basis for the clients' behaviors control scheme proposed next.

The basic idea of the procedure is as follows: At each period, a player may either continue playing the same strategy as in the previous period, or switch to other strategies, with probabilities that are proportional to how much higher his accumulated payoff would have been had he always made that change in the past. Specifically, let U be his total payoff up to now. For each strategy k different from his last period strategy j, let $V(k)$ be the total payoff he would have received if he had played k every time in the past that he chose j (and everything else remained unchanged). Then only those strategies k with $V(k)$ larger than U may be switched to, with probabilities that are proportional to the differences $V(k) - U$, which is called the "regret" for having played j rather than k. These probabilities are normalized by a fixed factor, so that they add up to strictly less than 1; with the remaining probability, the same strategy j is chosen as in the last period.

It is worthwhile to point out two properties [12,13] of this procedure. First, each player only needs to know the payoffs he received in past periods. He needs not know the game he is playing—neither his own payoff function nor the other players payoffs.[10] Second, due to the discrete

[10]Which is equivalent to saying that clients may not even be aware that there is a game going on.

nature of this algorithm, all regret values are calculated based on realized information. Thus an exogenous statistical "noise" is needed to make sure every action is played with some minimal frequency.

C. Employment of extra memory

The procedure [12,13] can be seen as a "stimulus-response" procedure, in the sense that clients will choose strategies which bring them more profits in the play history with higher probability. Though the idea is straightforward and intuitive, and the procedure does not require sophisticate information structure, it leads the game to converge to its set of correlated equilibria.

Notice that in the algorithm, clients make decisions only based on their realized payoffs, which depends on the server's capacity in real time. In a dynamic environment, if the server's capacity changes over time, clients need to adjust their strategies correspondingly, otherwise it could affect the learning performance of the algorithm. The idea of the proposed memory scheme is that if the client can check the server's throughput periodically and record the parameter values corresponding to any increase of the server's throughput, and uses those parameters in later learning, then the clients can adapt better to a dynamic environment in the sense that only more profitable values get preserved into future rounds of the game. This memory scheme should give the algorithm a better learning ability.

In general, the regulation of the clients' strategies in the memory scheme is carried out through adjusting the clients' server crash costs (value c, see Table I). If during a specific period the server has a smaller capacity, then at the same time each client modifies its crash cost to be larger, and vice versa. The reason is that a larger crash cost alters clients' realized payoffs, causes it to decrease whenever the server is crashed (and everything else remains unchanged). Then by applying the regret-based procedure [12,13], clients' intentions of sending traffic to the server will be suppressed, resulting in fewer server crashes, and vice versa. This should render the original algorithm a better performance in the circumstance of varying server capacity.

We allow clients to modify their own crash costs. More precisely, we leverage a parameter called *crash cost factor (CCF)*, denoted by α, to perform the server crash cost adaptation, and the server crash cost is set to be α times the benefit value from a successful traffic transmission.[11] Intuitively, the parameter describes how many times the punishment value a client gets from a server crash is as large as the benefit value it gets from a successful traffic transmission. Note that each client has an α, and they update their α separately.

In practice, each client maintains a history of profitable crash cost factor values in a memory with H entries (as shown in Fig. 3). The definition of profitable values and the steps of the procedure are explained below:

[11]Which is 1, so the server crash cost here is equal to α.

Value Weight	Memory Index k			
	1	2	$\cdots$	H
S_{CCF}	$S_{CCF,1}$	$S_{CCF,2}$	$\cdots$	$S_{CCF,H}$
S_{wt}	$S_{wt,1}$	$S_{wt,2}$	$\cdots$	$S_{wt,H}$

FIG. 3. Memory structure for each client.

In the beginning, each client randomly chooses a crash cost factor α from an unified cost factor options (denoted by C_{set}, e.g., $C_{\text{set}} = \{1, 5, 10, \ldots, 95, 100\}$) as their initial crash cost.

Then during the game, each client collects statistics about its average amount of effective traffic[12] "*eff_traffic_avg*" periodically.[13] If in any period i, compared with period $i-1$, the client's average (over the time slots in period i) amount of effective traffic increases, then the crash cost factor used in period i is recorded in the memory as a successful value S_{CCF}, and the corresponding increment in "*eff_traffic_avg*" is recorded as its weight S_{wt}. An index k ($1 \leq k \leq H$) determines the position in the memory to update. At the beginning, k is initialized to 1. Then k is increased whenever a new pair of elements, $S_{\text{CCF},k}$ and $S_{wt,k}$, are inserted. If $k > H$, k is set to 1. If there is no increment in "*eff_traffic_avg*", the memory is not updated.

Clients update their crash cost factors after they analyze the statistics. Before the memory is filled up, each client updates its crash cost factor using the random selection method. Once all H entries in the memory are filled, with probability p each client generates the next crash cost as a weighted mean of all values in the memory, as indicated in Eq. (9), and with probability $1-p$ they randomly select a value from C_{set}.

$$\alpha = \text{mean}_{\text{CCF}}(S_{\text{CCF}}) \tag{9}$$

where $\text{mean}_{\text{CCF}}(S_{\text{CCF}})$ is the weighted mean of all values in the historical memory of S_{CCF}, and defined as:

$$\text{mean}_{\text{CCF}}(S_{\text{CCF}}) = \sum_{k=1}^{H} w_k \cdot S_{\text{CCF},k} \tag{10}$$

$$w_k = \frac{S_{wt,k}}{\sum_{i=1}^{H} S_{wt,i}} \tag{11}$$

The complete adaptive procedure is shown in Algorithm 2, note that the algorithm is for per client.

[12]The amount of effective traffic of one client is defined as the amount of traffic it successfully routes to the server (note that this value is usually smaller than the total amount of traffic a client intends to route).

[13]A period is composed of a number of time slots.

Algorithm 2. Traffic throughput optimization scheme.

```
Input: Game length $T$, statistics period length $T_s$, memory size $H$, probability $p$, cost factor options $\boldsymbol{C_{set}}$.
1: Initialize average amount of effective traffic $L_{\mathrm{eff},0}$ to be 0, initialize memory updating index $k = 1$.
2: Randomly uniformly select a value $\alpha$ from $\boldsymbol{C_{set}}$ as the initial crash cost.
3: for $t = 1, 2, \ldots, T$ do
4:   Make strategies according to the regret-based procedure [12,13].
5:   if it is time to collect statistics and update crash cost factor then
6:     Calculate average amount of effective traffic $L_{\mathrm{eff},j}$ in current statistic period $j$.
7:     if $L_{\mathrm{eff},j}$ is greater than $L_{\mathrm{eff},j-1}$ then
8:       if memory updating index is $H$ then
9:         Reset memory updating index $k = 1$.
10:      end if
11:      Store statistic period $j$'s crash cost factor and corresponding weight $L_{\mathrm{eff},j} - L_{\mathrm{eff},j-1}$ into the memory location $k$.
12:      Increase memory index by 1: $k = k + 1$.
13:    end if
14:    if memory is not full then
15:      Randomly uniformly select a value from $\boldsymbol{C_{set}}$ as a new crash cost factor.
16:    else
17:      Randomly uniformly select a value $x$ in [0, 1].
18:      if $x < p$
19:        Calculate new $\alpha$ according to Eq. (9).
20:      else
21:        Randomly uniformly select a value from $\boldsymbol{C_{set}}$ as a new crash cost factor.
22:      end if
23:    end if
24:  end if
25: end for
```

We can expect an improvement of the algorithm's adaptability to the dynamic server capacity based on the following properties. First, by using the amount of effective traffic as a metric to guide the parameter adaptation process, it properly integrates server throughput with server crash probability. Only more profitable candidate values are retained, which eventually optimizes the objective in (4). Moreover, different server capacities usually need different sets of crash cost factors to cope with. The algorithm forces the clients to compare the amount of effective traffic with previous period's and keep those successful candidates, which preserves the trend of changes on the sets of crash cost factors as a result of varying server capacity. This gives the algorithm more robust adaptability. Second, at the end of each statistic period, each client has at least probability $1 - p$ to explore new options, which increases the chances for the algorithm to find new appropriate values after server capacity changes. Third, by applying the weighted mean operation, the amount of improvement is used in order to influence the parameter adaptation. The weighted mean is therefore helpful to propagate high quality candidates under the current server capacity, which in turn facilitates the progress rate.

IV. WORKING EFFICIENCY OPTIMIZATION

In this section, we study the problem of server working efficiency optimization. Specially, we consider the scenario where the server has enough capacity to hold all the clients' traffic. From Fig. 2 we can see that there exists a critical point (somewhere between 60 to 80) after which the server grows busier much more quickly. A server operating around that point gets its resources utilized more efficiently.[14] Therefore, the goal in this section is to design activity management schemes for the clients, so that by following which clients can learn the server's critical point and adjust their behaviors to let the server work around that efficient point.

A. Problem formulation

As discussed above, we leverage a parameter called *server ratio (SR)* to measure the server's working efficiency, which is defined as the ratio between the total traffic on the server and the corresponding time delay incurred by the traffic. Intuitively, the parameter represents the server's processing ability. Since in this scenario we are not focusing on the relations between clients' total traffic and the server's capacity, we use C to denote the server's capacity all the time, and $C \geq \sum_{i=1}^{n} L_i$. Our goal is to maximize the server ratio over the entire time:

[14]Compared with working around the critical point, a server operating beyond that point needs to invoke much more of its resources to process the same amount of traffic.

$$\textbf{Max}: \sum_{t=1}^{T} \frac{\sum_{i=1}^{n} a_i^t L_i}{d_t} \tag{12}$$

Subject to

$$\sum_{i=1}^{n} a_i^t L_i \le C, \quad \forall\ t = 1, 2, \dots, T \tag{13}$$

$$C \ge \sum_{i=1}^{n} L_i, \quad \forall\ t = 1, 2, \dots, T \tag{14}$$

$$d_t = \mathcal{F}\left(\sum_{i=1}^{n} a_i^t L_i\right), \quad \forall\ t = 1, 2, \dots, T \tag{15}$$

$$a_i^t \in \{0,1\}, \ \forall\ i = 1,2,\dots,n, \ \forall\ t = 1,2,\dots,T \tag{16}$$

$$L_i \in (0, L_{\mathrm{MAX}}], \quad \forall\ i = 1, 2, \dots, n \tag{17}$$

where constraint (14) states that the server has enough resources to process all clients' requests. Constraint (15) specifies the time delay d_t at time t depends on the total traffic on the server at that time, according to the server's crowdedness function $\mathcal{F}$.

Note that the NP-hardness of this problem can also be obtained by a reduction from the knapsack problem, as the same shown in the previous section. In the next part, we propose both centralized and discrete adaptive activity management schemes to address it.

B. Activity management schemes

The proposed activity management schemes are based on the reinforcement learning procedure [12,13], in which the reward function acts as an essential role.

In general, designing an appropriate reward function is crucial to make reinforcement learning work. The reward function needs to capture exactly what the goal is. One difficulty of designing such a reward function is trying to design one that encourages the desired behaviors while still being learnable [16].

For the working efficiency optimization scenario, the reward function should encourage clients' actions that improve the server's SR, which is equivalent to optimizing the objective (12).

1. Centralized approach

In a straightforward approach, the server monitors its server ratio at each time step. Then depending on the value, the server assigns payoff 1 to all clients whenever it sees an increase in its SR, and payoff −1 when its SR drops. This reward function simply informs clients which action to take under what circumstance is profitable or otherwise bringing punishment. The issuing of profit or punishment depends on the increase or decrease of the SR values. In other words, the changings of SR values control the incentives for clients to take certain actions in certain circumstances.

Algorithm 3. Centralized SR optimization scheme.

```
Input: Game length $T$, the server's crowdedness function $\mathcal{F}$.
1: Initialize all clients' strategies randomly for time step 1 as
   set $\boldsymbol{A_1}$.
2: Calculate server ratio $SR$ according to $\mathcal{F}$, and make a copy
   $SR_{\mathrm{old}} = SR$.
3: for $t = 2, 3, \dots, T$ do
4:    All clients play the regret-based procedure [12,13].
5:    Calculate server ratio $SR$ according to $\mathcal{F}$.
6:    if $SR > SR_{\mathrm{old}}$ then
7:       Assign payoff 1 to all clients.
8:    else
9:       Assign payoff $-1$ to all clients.
10:   end if
11:   Update $SR_{\mathrm{old}} = SR$.
12: end for
```

Therefore, through time this reward function will gradually instruct clients to take more actions which improves the SR and prevent more actions which reduces the SR, and hence eventually optimize the objective (12). We call this algorithm the centralized SR optimization scheme, as shown in Algorithm 3. Though the idea is simple and intuitive, the algorithm gives a high and steady level of SR values with fast convergence, as shown in the simulation Sec. V.

2. Distributed approach

The straightforward centralized approach targets directly at the server ratio, therefore has a good optimization value. However, the centralized nature of the algorithm limits its scalability (as shown in the simulation Sec. V). In this part, we consider a distributed scheme which only uses local information.

The important part of optimizing the server's working efficiency is the optimization of the server's SR value, which is equal to the total amount of traffic divided by the time delay incurred by the traffic. The overall traffic is composed of individual traffic coming from each client. This indicates that each client can use its own amount of traffic divided by the time delay[15] as an approximation of the server's SR value. However, if the policy of how to assign reward or punishment totally resembles the centralized one, clients will hold their traffic all the time, resulting in a very inefficient server utilization rate. The reason is that by choosing to hold traffic, the total amount of traffic in the server drops and hence incurs a smaller time delay. The clients' estimated SR values (which is the ratio between their traffic load and current time delay) raise, which by the

[15]Which can be observed from clients' side.

Algorithm 4. Distributed SR optimization scheme.

```
Input: Game length $T$.
1: Initialize time step 1's strategy $a_1$ randomly.
2: Observe the server's time delay, calculate an estimated $SR$, and
   make a copy $SR_{\mathrm{old}} = SR$.
3: for $t = 2, 3, \ldots, T$ do
4:   Take an action based on the regret-based procedure [12,13].
5:   Observe the server's time delay, and calculate an estimated
     $SR$.
6:   if client chooses Hold at time $t$ then
7:     It gets payoff 0.
8:   else
9:     if $SR > SR_{\mathrm{old}}$ then
10:      It gets payoff 1.
11:    else
12:      It gets payoff $-1$.
13:    end if
14:  end if
15:  Update $SR_{\mathrm{old}} = SR$.
16: end for
```

centralized reward function, will give clients more positive payoffs.

Therefore, to compensate for this *Hold* bias trend, a payoff 0 is assigned to clients whenever they take the action *Hold*. The resulting algorithm is called the distributed SR optimization scheme, as shown in Algorithm 4 (for per client). It enjoys the distributed algorithm's good scalability while possessing a high average SR value and fast convergence, as shown in the simulation section.

C. Time window constraint

Occasionally in real communication scenarios, clients usually expect to receive responses from the server within a time window. This puts a time constraint on the problem. In this part, we study the time constraint version of the working efficiency optimization problem.

1. Time constraint formulation

Denote the length of the time window as T_w. The time constraint implicates that ideally for all clients, the time intervals between every two adjacent *Send* actions should be no longer than T_w during the entire time. Mathematically, denote as $\boldsymbol{B}_i$ the set of time steps at which client i takes action *Send*. Then ideally, the difference between every two adjacent elements in set $\boldsymbol{B}_i$ should be no longer than T_w for every client i. Hence:

$$b_{j+1} - b_j \le T_w, \quad \forall\ j = 1, 2, \ldots, |\boldsymbol{B}_i| - 1, \quad \forall\ i \in \boldsymbol{N} \tag{18}$$

where $|\boldsymbol{X}|$ denotes the size of set $\boldsymbol{X}$.

In order to ensure the above time constraint, alternatively we try to minimize the amount of time that exceeds the time window T_w between every two adjacent *Send* actions for all clients. Naturally, define the part of time that goes beyond T_w as penalty value for violating the time constraint (18). For client i, the penalty value at time t is:

$$P_i^t = t - b_j - T_w, \quad \forall\ i \in \boldsymbol{N}, \quad \forall\ t = 1, 2, \ldots, T \tag{19}$$

where $b_j \in \boldsymbol{B}_i$ satisfies:

$$b_j \le t < b_{j+1} \tag{20}$$

According to the penalty's definition, we have:

$$P_i^t \ge 0, \quad \forall\ i \in \boldsymbol{N}, \quad \forall\ t = 1, 2, \ldots, T \tag{21}$$

Therefore, the goal of the working efficiency optimization problem with time constraint is to maximize the SR value as well as minimize the total penalty:

$$\textbf{Max}: \sum_{t=1}^{T} \frac{\sum_{i=1}^{n} a_i^t L_i}{d_t}, \quad \textbf{Min}: \sum_{t=1}^{T} \sum_{i=1}^{n} P_i^t \tag{22}$$

Subject to

$$\sum_{i=1}^{n} a_i^t L_i \le C, \quad \forall\ t = 1, 2, \ldots, T \tag{23}$$

$$C \ge \sum_{i=1}^{n} L_i, \quad \forall\ t = 1, 2, \ldots, T \tag{24}$$

$$d_t = \mathcal{F}\left(\sum_{i=1}^{n} a_i^t L_i\right), \quad \forall\ t = 1, 2, \ldots, T \tag{25}$$

$$P_i^t = t - b_j - T_w, b_j \le t < b_{j+1}, b_j, b_{j+1} \in \boldsymbol{B}_i, \quad \forall\, i \in \boldsymbol{N}, \quad \forall\ t = 1, 2, \ldots, T \tag{26}$$

$$P_i^t \ge 0, \quad \forall\ i \in \boldsymbol{N}, \quad \forall\ t = 1, 2, \ldots, T \tag{27}$$

$$a_i^t \in \{0, 1\}, \ \forall\, i \in \boldsymbol{N}, \ \forall\, t = 1, 2, \ldots, T \tag{28}$$

$$L_i \in (0, L_{\mathrm{MAX}}], \quad \forall\ i \in \boldsymbol{N} \tag{29}$$

The multiobjective optimization problem (22) is NP-hard, which can be easily seen by setting $T_w = \infty$, and then we get a reduction from the single-objective optimization problem (12) (which is also a NP-hard problem by a reduction from the knapsack problem). Next, we propose both a centralized and a distributed scheme to tackle it.

2. Activity management schemes with time window constraint

The general idea is to modify the reward functions in the Algorithm 3 and 4, so that the second optimization goal in objective (22) is also incorporated into the optimization process. Therefore, the new reward functions need to control both the SR value and the total penalty value.

For a centralized approach, the server keeps track of the SR value and total penalty at each time step. In the case where the penalty value at time t is no greater than the penalty value at time $t-1$, the server uses the same reward function as in the Algorithm 3 to optimize the SR value. Otherwise, the server checks the SR value at time t. If the SR value drops, then that means the current combination of clients' actions does not optimize objective (22) at all, hence all clients get payoff -1. On the other hand, if the SR value grows, then that means that the combination optimizes the SR value. The server needs to adjust clients' strategies so that it also optimizes the total penalty value. Note that in this case, the increment of total penalty at time t only comes from the clients who hold a positive penalty value at time t, so those clients get payoff -1. The rest of the clients who have 0 penalty at time t get payoff 1.

Similarly, for a distributed approach, each client keeps track of their own penalty values. In the case where the penalty does not grow at time t, the client uses the same reward function as in the Algorithm 4. On the other hand, if the client's penalty increases at time t, it gets punishment payoff -1.

The centralized and the distributed schemes are shown in Algorithm 5 and 6, respectively. Though just adding a simple layer of time constraint to the reward function, it significantly improves the results in terms of reducing excess waiting time, as shown in the simulation section.

Algorithm 5. Centralized SR optimization scheme with time constraint.

```
Input: Game length $T$, the server's crowdedness function $\mathcal{F}$,
  time window $T_w$.
1: Initialize all clients' strategies randomly for time step 1 as
   set $\boldsymbol{A_1}$.
2: Calculate server ratio $SR$ according to $\mathcal{F}$, and make a copy
   $SR_{\text{old}} = SR$.
3: Calculate penalty values based on the time window $T_w$ for
   every client, and store them in set $\boldsymbol{P}$.
4: Make a copy of the total penalty value $P_{\text{old}} = \sum \boldsymbol{P}$.
5: for $t = 2, 3, \ldots, T$ do
6:   All clients play the regret-based procedure [12,13].
7:   Calculate server ratio $SR$ according to $\mathcal{F}$.
8:   Calculate penalty value set $\boldsymbol{P}$ based on $T_w$.
9:   if $\sum \boldsymbol{P} \le P_{\text{old}}$
10:    if $SR > SR_{\text{old}}$
11:      Assign payoff 1 to all clients.
12:    else
13:      Assign payoff $-1$ to all clients.
14:    end if
15:  else
16:    if $SR > SR_{\text{old}}$ then
17:      Assign payoff 1 to clients with $\boldsymbol{P}_i = 0,\ i \in \boldsymbol{N}$.
18:      Assign payoff $-1$ to clients with $\boldsymbol{P}_i > 0,\ i \in \boldsymbol{N}$.
19:    else
20:      Assign payoff $-1$ to all clients.
21:    end if
22:  end if
23:  Update $SR_{\text{old}} = SR$.
24   Update $P_{\text{old}} = \sum \boldsymbol{P}$.
25: end for
```

Algorithm 6. Distributed SR optimization scheme with time constraint.

```
Input: Game length $T$, time window $T_w$.
1: Initialize time step 1's strategy $a_1$ randomly.
2: Observe the server's time delay, calculate an estimated $SR$,
   and make a copy $SR_{\text{old}} = SR$.
3: Calculate penalty value $P$ based on the time window $T_w$, and
   make a copy $P_{\text{old}} = P$.
4: for $t = 2, 3, \ldots, T$ do
5:   Take an action based on the regret-based procedure [12,13].
6:   Observe the server's time delay, and calculate an estimated
     $SR$.
7:   Calculate penalty value $P$ based on $T_w$.
8:   if $P \le P_{\text{old}}$ then
9:     if client chooses Hold at time $t$ then
10:      It gets payoff 0.
11:    else
12:      if $SR > SR_{\text{old}}$ then
13:        It gets payoff 1.
14:      else
15:        It gets payoff $-1$.
16:      end if
17:    end if
18:  else
19:    It gets payoff $-1$.
20:  end if
21:  Update $SR_{\text{old}} = SR$.
22:  Update $P_{\text{old}} = P$.
23: end for
```

V. PERFORMANCE EVALUATIONS

In this section, we evaluate the performance of the algorithms for both server throughput optimization and working efficiency optimization scenarios with parameters from the accelerator control system. Specially, for the second scenario we compare the centralized algorithm 3 with the Q-learning procedure [3], and show a prominent improvement on both optimization value and convergence rate aspects.

For both scenarios, we assume there are 500 clients, they send *get()* requests to an FEC to query their interested machine parameters at a constant rate. The specific FEC used is equipped with a MVME2100 VME processor module [17]. The largest parameter size clients can get from that FEC is an array with 256 integers. All message sizes are generated randomly uniformly between 0 and the

TABLE II. Global parameter settings.

Parameter	Value
Number of clients n	500
Simulation length T	12 hours
Stage game length (one time slot)	1 second
Clients' message rate	1 msgs/ sec
Maximum traffic load L_{MAX}	$256 * 8$ bytes

maximum size.[16] We run the simulations for 12 hours, and each stage game's duration is 1 second. Table II summarizes those global parameter settings.

A. Server throughput optimization—Performance enhancement by the memory structure

For the server throughput optimization scenario, we show that the adopting of the memory structure improves the learning ability of the original procedure [12,13] significantly.

We assume the server changes capacity every 3 hours. The way the server changes capacity is as follows: Since the total amount of traffic L_{sum} from clients is known, every 3 hours a fraction between 0 and 1 is picked and multiplied with L_{sum}, then the result is the server's new capacity for the next period. Since the simulation length is 12 hours, the server has 4 capacity periods. In this simulation, we assume the server alternates between low and high capacity values. Without loss of generality, we use fraction values of 0.3,0.8,0.2,0.5.

Note that there are several parameters in the proposed scheme need to be set that will influence the overall performance in different ways. Next, we discuss the effects of the parameters in the memory scheme on the system performance, then present the simulation results. It also remains an interesting topic for future study to experiment with different combinations of those parameters, and verify their impacts on the system performance.

1. Effects of the parameters

a. Crash cost factor options C_{set} Clients use a random selection method which chooses values from a predefined cost factor set C_{set} to update their historical memory. The values in the available set will decide the overall values in their memory. Generally, a set with larger values causes more holding requests among clients, hence reducing the server's throughput and crash probability, and vice versa. According to experiments and history of the system, in this simulation we use a cost factor set of increasing values from 1 to 100 with 5 as the step size, i.e., $C_{\mathrm{set}} = \{1, 5, 10, \ldots, 95, 100\}$.

b. Memory size H and statistic period length T_s As mentioned above, clients collect statistics about their effective amount of traffic in every statistic period with length T_s, and adaptively update their memory with size H. If T_s and H are small, then only recent cost factors are used (since older values are rapidly overwritten as a result of frequent memory updates and limited memory size), which makes clients learn the current circumstance faster. However, it could also make clients short sighted. Since T_s is small, the information clients gathered from that short period of time may inaccurately reflect the real situation in the long run. Moreover, a small memory buffer can only store recent values that are only suitable for the current short period, some potentially more profitable values for the long run could get eliminated. Those factors may result in degraded system performance. On the other hand, if T_s and H are large, the system performance could get improved due to more accurate statistic data, but the convergence rate of the algorithm is expected to slow down because older parameters continue to have influence for a long time. In this simulation, clients collect statistics every 5 minutes, and each has a memory size of 20.

c. Adaptive probability p The adaptive probability decides how likely the algorithm uses the weighted mean method to update server crash cost for the next period. A large value causes clients to do adaptive updates frequently, resulting in fast learning to the current situation. However, due to the small chance of introducing new individuals, it may cause slow responses among clients to a server capacity variation. On the other hand, a small adaptive probability will impair the adaptive learning process, make it more dominated by random "noise." In this simulation, we use an adaptive probability of 0.9.

Table III summarizes the parameter settings for this scenario.

2. Simulation results

In this part, we present the simulation results and demonstrate that, first, both the adopted regret-based procedure and the proposed memory scheme can effectively manage clients' behaviors. Moreover, the proposed scheme can better handle the dynamic server capacity in our system, resulting in a higher server throughput and lower crash probability. For simplicity, we refer to the original regret-based procedure as RR scheme and the

TABLE III. Memory scheme parameter settings.

Parameter	Value
Server capacity period	3 hours
Server capacity volume fraction	0.3,0.8,0.2,0.5
Crash cost factor options C_{set}	$1, 5, 10, \ldots, 100$
Memory size H	20
Statistic period T_s	5 minutes
Adaptive probability p	0.9

[16] $256 * 8$ bytes.

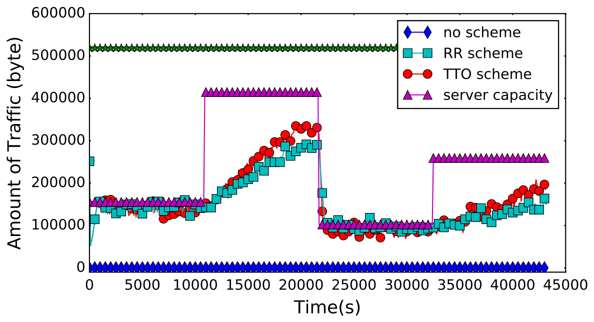


FIG. 4. Comparison results of effective amount of traffic routed by clients. The star line at the top is the sum value of all clients' traffic (this is also the minimal capacity value required for a server to hold all clients' traffic). The triangle line is the actual server's capacity, which changes every 3 hours (the exact capacity is listed in Table III). The circle line and square line indicate the actual amount of traffic clients manage to route to the server using different schemes. Without using any activity regulation, clients send their traffic all the time, hence the server crashes all the time (the star line is above the triangle line all the time), resulting in an all-0 effective amount of traffic (plotted in diamonds). On the other hand, with the activity regulations (plotted in circles and squares), clients learn to send their traffic depending on the current volume of the server's capacity, hence they successfully route their traffic to the server. Moreover, clients under the TTO scheme exhibit a better learning ability than under the RR scheme, hence route more traffic (the circle line is above the square line). The exact statistics are summarized in Table V.

proposed memory based traffic throughput optimization procedure as TTO scheme.

Figures 4–6 show the comparison results [18] of various system performances achieved by clients, when there is no regulation on their behaviors, when they apply the RR scheme, and when they apply the TTO scheme.

Figure 4 shows the comparison results on effective amount of traffic routed by clients to the server. As we can see, first, both the adopted regret-based scheme and our proposed memory based scheme achieve a significant improvement over the naive case (plotted in diamonds), where there is no regulation on clients' behaviors.[17] Second, for both cases clients are able to adapt their strategies properly to the capacity changes so that the server's extent of usage is consistent with the amount of resources it possesses. Third, with the memory based scheme (plotted in circles), clients adapt their strategies better to the capacity changes compared with the case where they use the regret-based scheme (plotted in squares), which demonstrates that our proposed scheme can more effectively adjust clients' behaviors so that the server's resources are utilized in a more efficient way.

Figure 5 shows the comparison results on the server alive time during the entire simulation. The downward spikes represent the server is crashed at the corresponding time periods. We can see that, most of the time both cases achieve a robust server performance, which is a huge improvement over the no regulation case where the server crashes all the time (plotted in diamonds). Furthermore, Fig. 5(b) exhibits a smaller server crash probability, which proves that the proposed memory based scheme can better accommodate the dynamic server capacity scenario and hence provide a more reliable server operation.

Figure 6 shows the comparison results of the statistic plot on the counts of "*Send*" for every client. The count of "*Send*" for a client is the number of times a client chooses to route its traffic to the server. It implies how many times a client has been serviced by the server during the entire time (if the server is not crashed by a heavy traffic). As shown in the plot, Fig. 6(b) reveals a less intense variation across the whole population, which indicates that the proposed memory based scheme helps to promote a more equitable user experience. From Table IV, we can see that the standard deviation for the counts of "*Send*" among all clients is reduced by 45.5% on average.

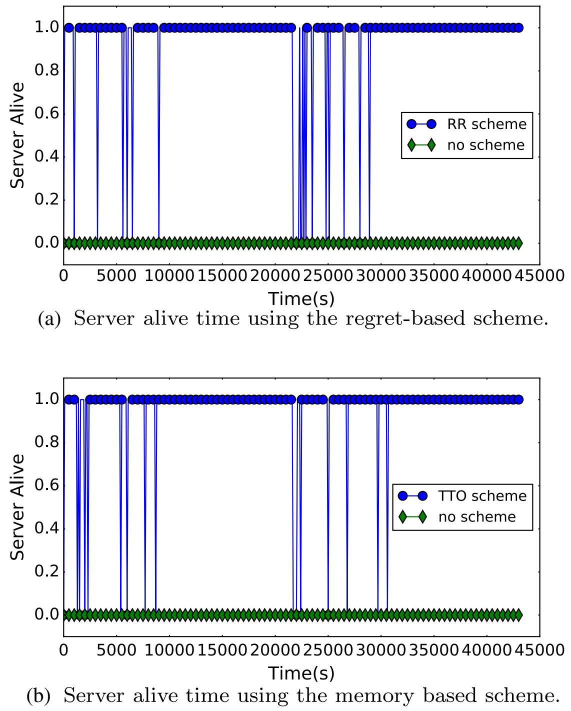
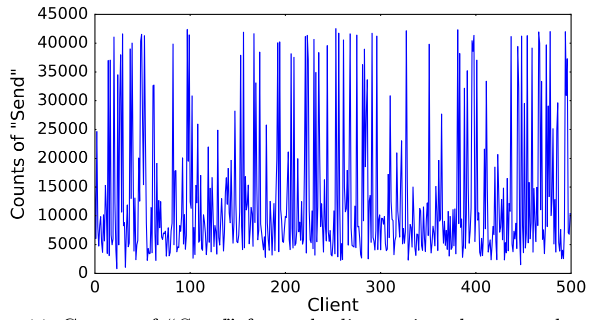
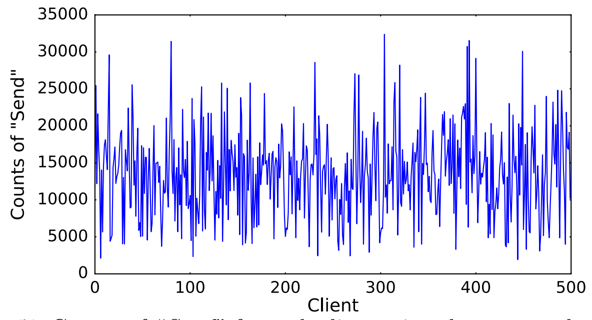


FIG. 5. Comparison results of server alive time. The diamond line at the bottom indicates that if there is no activity management, the server crashes all the time. Furthermore, the TTO scheme [Fig. (b)] results in a clearly less crashes than the RR scheme [Fig. (a)]. The exact crash probabilities are summarized in Table V.

[17]As shown by the star line on top, the actual amount of traffic clients route to the server will always be greater than the server's capacity, since clients tend to send their traffic all the time when there is no regulation, resulting in an all-0 effective amount of traffic.



(a) Counts of “*Send*” for each client using the regret-based scheme.



(b) Counts of “*Send*” for each client using the memory based scheme.

FIG. 6. Comparison results of counts of “*Send*.” The TTO scheme [Fig. (b) [produces a much less variation than the RR scheme [Fig. (a)], which means it gives clients more equal opportunities to send their traffic.

Table V summarizes the performance statistics during each server capacity period and the whole simulation time from both the cases where the proposed TTO scheme is applied and where the RR scheme is applied. In periods 2 and 4 where the server has higher capacities, neither algorithm causes any crash. Besides, the TTO scheme helps clients adapt faster to the capacity changes than the RR scheme, hence has 18.8% and 22% improvement on the server throughput, respectively. In periods 1 and 3 where the server has lower capacities, both schemes utilize the server to its full capacity, and the TTO scheme results in a 30.6% and 52% less crash probability than the RR scheme, respectively. On average, the TTO scheme reduces server crash probability by 46%, while at the meantime increasing the server's throughput by 12.7%. It validates that the TTO scheme can more effectively accommodate the dynamic server capacity circumstance in our system than the RR scheme.

TABLE IV. Variation of “Send.”

Scheme	“Send” Count standard deviation
TTO	6326.699
RR	11599.990

TABLE V. Comparison results of system metrics.

	Crash probability period #				
Scheme	1	2	3	4	All
TTO	0.0446	0.0	0.0815	0.0	0.0315
RR	0.0643	0.0	0.1693	0.0	0.0584
	Effective traffic average ($\times 10^5$) period #				
Scheme	1	2	3	4	All
TTO	1.295	2.700	0.795	1.495	1.569
RR	1.285	2.273	0.778	1.225	1.393

B. Server working efficiency optimization

In this part, we evaluate the Algorithms 3 and 4 in terms of optimization value and scalability. First, we demonstrate that both schemes are very effective on optimizing the server's working efficiency, and the scheme 4 has a better scalability than the scheme 3, especially when the number of clients in the system is large. Second, we compare the optimization results from the scheme 3 with the results obtained from the Q-learning procedure [3], and show that the scheme 3 has a much higher optimization value and faster convergence rate.

1. The centralized scheme and the distributed scheme

The performances of Algorithms 3 and 4 are evaluated and compared. First, we show the effective learning ability of both algorithms. Next, we analyze their optimization procedures on SR values in details. Then, we compare their scalability through considering them in cases with different number of clients.

a. Effective optimization area Figure 7 shows the effective working area[18] (plotted in circles) of the Algorithm 3 (compare with Fig. 2). As we can see the algorithm successfully avoids the high crowdedness area and devotes its optimization efforts to the middle part, where high SR values are more likely to appear. It implies that with a proper reward function the algorithm has the learning ability to understand what the server's working efficiency means and how to optimize it.

b. Optimized SR values Figure 8 shows the comparison results of the SR values produced by the centralized scheme (plotted in circles) and the distributed scheme (plotted in squares) during the whole simulation time. As we can see, first, both schemes are very effective on optimizing the server's working efficiency. Specially, the average SR value in the centralized scheme is 0.7% less than the optimal value.[19] The difference comes from the variations in the resulting SR values which comes from the nature

[18]Algorithm 4 has a similar property.

[19]The optimal value can be calculated by Monte Carlo simulation [14].

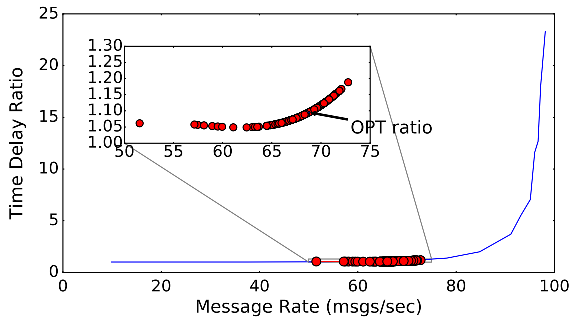


FIG. 7. The effective area of the working efficiency optimization algorithm.

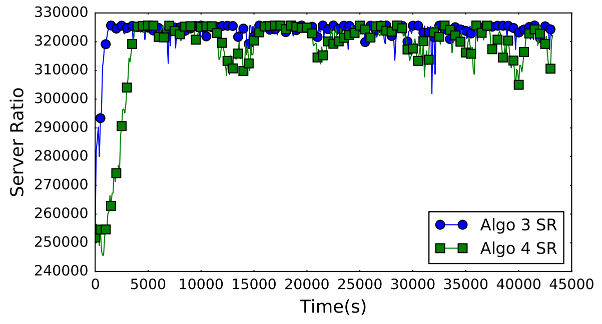


FIG. 8. Comparison result of the SR values between the centralized scheme and the distributed scheme.

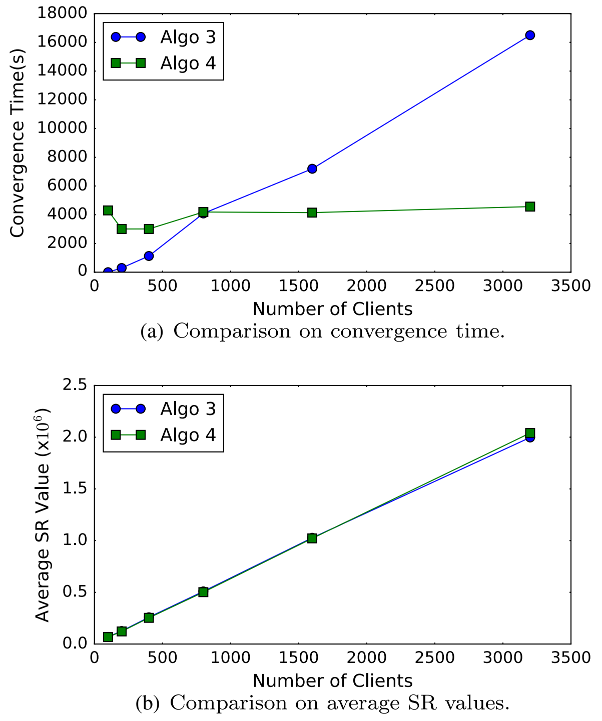
of the underlying procedure.[20] Second, compared with the centralized scheme, the distributed scheme maintains a competitively high level of SR values (1.8% less than the centralized value and 2.5% less than the optimal value on average) with a fast convergence rate, which proves that although lacking of global information, the distributed scheme is still able to optimize the server's working efficiency to a satisfied extent.

c. Scalability For comparing the scalability between the two schemes, we compare the convergence time of SR values in both schemes among cases with different number of clients.

Specifically, the number of clients n in each case is doubled from case to case starting[21] from 100 to 3200. Then for each scheme in each case, we record the time steps at which the SR value is no less than the optimized SR value,[22] and store them in a set $\boldsymbol{T_c}$. Then the 100th element of $\boldsymbol{T_c}$ is considered as an estimation of the scheme's convergence time.[23] We also record the corresponding optimized SR values in each case for both algorithms.

The results are shown in Fig. 9, as we can see from 9(a), the convergence time of the centralized scheme grows linearly with the number of clients, and becomes much higher than the distributed scheme's after $n = 800$. On the other hand, the distributed scheme manages to maintain its convergence time almost in a same level regardless of the number of clients.

As shown in 9(b), the optimized SR value in the distributed scheme is on average 1.92% less than the value produced by the centralized scheme for the first 5 cases, and 2.16% higher for the last case. The reason for the higher optimized SR value in the last case is that for cases with a large number of clients (say, 3200), the centralized scheme takes a much longer time to converge, which in turn impacts its resulting overall average value (optimized SR value).



(a) Comparison on convergence time.

(b) Comparison on average SR values.

FIG. 9. Comparison results between two schemes.

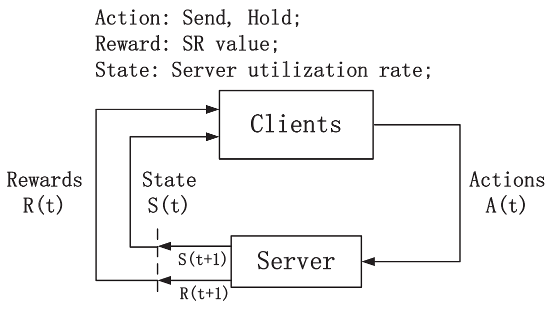


FIG. 10. The client-server interaction in the MDP model.

[20]The procedure [12,13] requires an exogenous statistical "noise" to "tremble" over every action to make sure the contingent payoffs can be estimated using only the realized payoffs.

[21]So $n = 100, 200, 400, 800, 1600, 3200$. The simulation result also applies to cases with client number higher than 3200. Here just an illustration.

[22]The optimized SR value is the average SR value over the entire simulation time.

[23]Since SR values are fluctuated during the procedure, hitting the optimized SR value a few times does not necessarily guarantee the convergence of the scheme.

TABLE VI. Q-learning parameter settings.

Parameter	Value
State space	$0, 10, \ldots, 100\%$
Learning rate α	0.3
Discount factor γ	0.9
Exploration probability ϵ	0.05

This provides us a guideline of how to choose between schemes, which depends on the number of clients in the system. In general, for a system with relatively small number of clients (say, less than 800), the centralized scheme is preferred for a faster convergence rate and higher optimized SR value. However, for a system with large number of clients (say, greater than 800), the distributed scheme wins out for its better scalability.

2. Comparison with Q-learning

To enable the employment of Q-learning, the working efficiency optimization problem can alternatively be formulated as an Markov decision process (MDP) [14]. Specially, the MDP is denoted as a 3-tuple $(\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R})$. The set of states $\boldsymbol{S}$ is the server's utilization degree, the set of actions $\boldsymbol{A}$ is $\{Send(S), Hold(H)\}$, and the set of rewards $\boldsymbol{R}$ is the server ratio assigned by the server. The process goes as follows: At each time step, each client chooses an action S or H, and depending on the total traffic being routed to the server, the server calculates the server ratio and sends it back to the clients as the reward.[24] Then the MDP moves to the next state represented by a different utilization rate of the server.[25] The interactions between the clients and the server in the MDP model is shown in Fig. 10.

Note that there are several control paradigms that can solve MDPs, such as dynamic programming, Monte Carlo Method, or temporal-difference learning (including some well-known reinforcement learning algorithms, such as Sarsa [14], Q-learning [3], or double Q-learning [19], etc.). Those methods will converge to the optimum policy with the condition that all state-action pairs will be visited an infinite number of times. However in our game, the total number of states is 2^N, where N is the number of clients.[26] It will be impractical to apply any of those methods directly to acquire an optimum control policy within a reasonable amount of time when the number of clients is relatively big (say, 500).

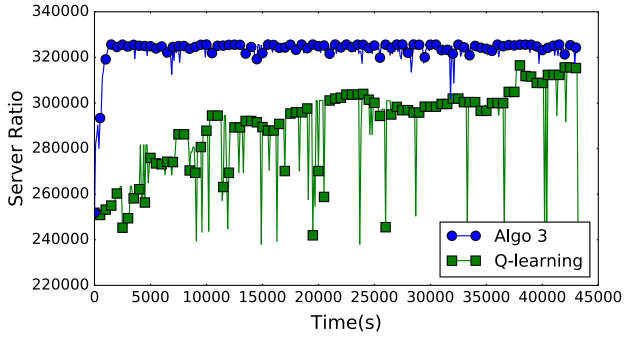


FIG. 11. Comparison result of the SR values between the centralized scheme and the Q-learning scheme.

In order to make the Q-learning feasible to run for the comparison, the state space is simplified. Specially, the server's possible utilization range 0–100% is divided[27] into 10 equal intervals $(0, 10, \ldots, 100\%)$. At each time step, the actual server's utilization rate is calculated and rounded to the nearest state in the simplified state space. Then we can apply the Algorithm 1 to get the Q-learning control policy. The parameter settings[28] for this part of the simulation is shown in Table VI.

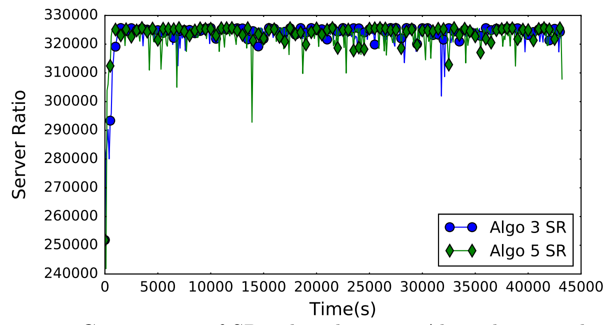
Since the Q-learning control scheme used here is a centralized scheme, we compare it with the Algorithm 3. Even now the MDP has only 11 states, it still takes time for the Q-learning to get good enough approximations of the optimal values of the 22 state-action pairs[29] for all 500 clients. Consequently, as shown in Fig. 11, the Q-learning procedure does not get enough time to do the updates, so it gets a degraded performance (but we can still see that it gradually converges to the optimal value). On the other hand, the centralized algorithm 3 converges much faster and produces a much higher and steadier level of SR values—11.6% larger on average, which further demonstrates its strong learning ability.

C. Working efficiency optimization with time window constraint

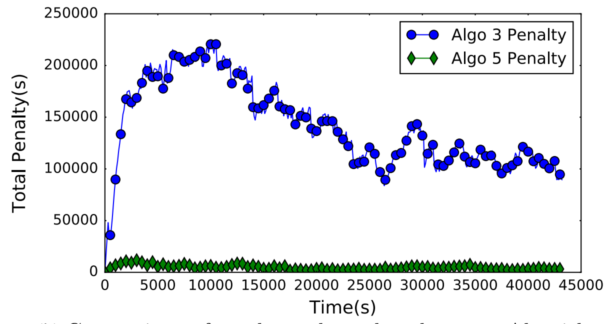
In this part, we evaluate the Algorithms 5 and 6, and compare them with the Algorithms 3 and 4 to demonstrate their ability to control the average total penalty value. Here, we use a time window[30] value of $T_w = 20$ s.

[24]From the model, we can see that the control policy is a centralized scheme.
[25]We can clearly see that the reward function here captures the optimization goal accurately—to maximize the SR value (reward), which is equivalent to optimize the server's working efficiency.
[26]Each client's choice of sending or holding its traffic can change the utilization degree of the server. In general, for n clients there will be 2^n different resulting utilization rates of the server.

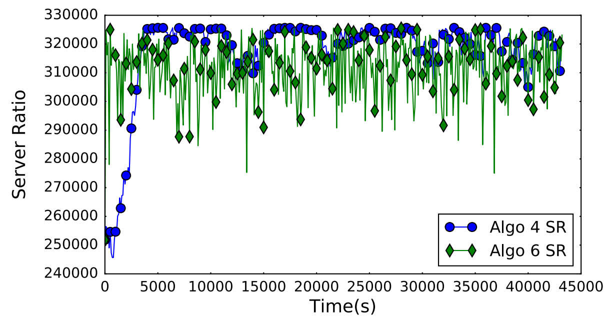
[27]We can divide the range into smaller intervals to be more accurate, but it renders the algorithm a lower optimization value and slower convergence rate. The reason is explained below.
[28]Other parameters (i.e., α, γ, ϵ) are decided by conducting experiments and picking the value returns the best results.
[29]Since each client only has 2 actions, *Send* or *Hold*.
[30]This is the GPIB bus protocol's communication timeout value [20], which is used to conduct the experiment shown in Fig. 2.



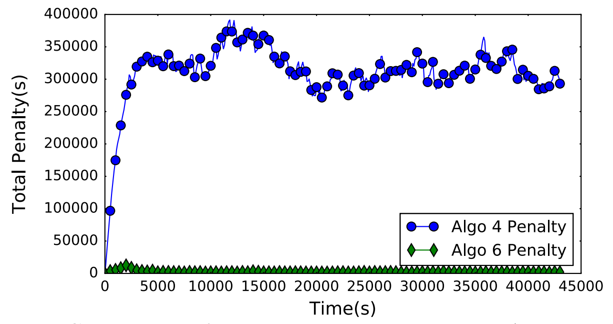
(a) Comparison of SR values between Algorithm 3 and 5.



(b) Comparison of total penalty values between Algorithm 3 and 5.



(c) Comparison of SR values between Algorithm 4 and 6.



(d) Comparison of total penalty values between Algorithm 4 and 6.

FIG. 12. Comparison results of SR values and total penalty values.

Figure 12 presents the comparison results of SR values and total penalty values between the schemes without the time constraint (the Algorithms 3, 4 plotted in circles) and the schemes with the time constraint (the Algorithms 5, 6 plotted in diamonds). As shown in Fig. 12(a) and 12(c), the schemes 3, 4 maintain a slightly higher level of SR values than the schemes 5, 6. The reason is that the schemes 5, 6 also spend some optimization efforts on minimizing the total penalty values. Consequently, the total penalty values in the schemes 5, 6 are much lower than the values in the schemes 3, 4, as shown in Fig. 12(b) and 12(d).

The exact performance statistics (in terms of the average total penalty values and SR values) are summarized in Table VII. As we can see, by adding the time constraint to the algorithms, it only affects the average SR value a little while reducing the average total penalty value greatly. For the centralized and the distributed scheme, by considering the time window constraint, the average SR values drop by 0.16% and 1.9%, while the average total penalty values are reduced by 30.1 and 130.1 times, respectively. This illustrates the big benefits of introducing the time window constraint to the algorithms. Specially, the penalty value for the scheme 6 indicates that on average for every client, the time intervals between each of its adjacent *Send* actions exceed the time window (20 s) by 4.78 s (compared with more than 10 minutes excess waiting time in the scheme 4) over the entire simulation time (12 hours), which demonstrates the algorithm's strong effectiveness on handling the time window constraint.

TABLE VII. Comparison results among schemes.

	Algorithm			
	3	4	5	6
Average penalty ($\times 10^4$ sec)	14.1988	31.1312	0.4719	0.2393
SR ($\times 10^5$ byte)	3.23322	3.17526	3.22819	3.11532

VI. CONCLUSIONS

In this work, we study a practical scenario related to server performance in the RHIC control system using game theory and reinforcement learning. We divide the scenario into two cases. In the first case where the server does not have enough resources to hold all clients' traffic, our goal is to safely and efficiently route clients' traffic so that the server does not crash and the server's throughput is maximized. We adopt a regret-based learning procedure as a base routine to regulate clients' behaviors, and then propose a memory structure to improve the procedure's learning ability, especially in a dynamic environment. In the second case where the server has enough resources to hold all clients' traffic, our goal is to manage clients' activities so that the server's resources get utilized efficiently. We propose both centralized and distributed schemes to address it. Additionally, we handle the real-world time constraints existing in the system. Through extensive simulations, we demonstrate that, for the first case the proposed memory structure significantly improves the base procedure's performance, leading to a higher server throughput and lower crash probability. For the second case, both the centralized and the distributed schemes can manage clients' activities effectively so that the server's working efficiency is maximized. Moreover, the time window constraint can be effectively handled by the schemes, which in turn brings the system a promising performance improvement.

ACKNOWLEDGMENTS

This work is supported by Brookhaven Science Associates, LLC under Contract No. DE-SC0012704 with the U.S. Department of Energy. J. C. is supported by NSF CAREER under Award No. 1553385. The first author would like to thank Robert Olsen for his expert advice to help the author understand the control system. Thanks to John Morris and Peter Zimmerman for their efforts to set up an experimental machine.

[1] D. S. Barton *et al.*, RHIC control system, Nucl. Instrum. Methods Phys. Res., Sect. A **499**, 356 (2003).

[2] T. G. Robertazzi, *Networks and Grids: Technology and Theory* (Springer, New York, 2007).

[3] C. J. C. H. Watkins and P. Dayan, Q-learning, in *Machine Learning* (Springer, Netherlands, 1992), pp. 279–292.

[4] M. J. Osborne, *An Introduction to Game Theory* (Oxford University Press, New York, 2004).

[5] S. Singh, M. Kearns, and Y. Mansour, Nash convergence of gradient dynamics in general-sum games, in *Uncertainty in Artificial Intelligence* (2000), pp. 541–548.

[6] M. Bowling and M. Veloso, Multiagent learning using a variable learning rate, Artif. Intell. **136**, 215 (2002).

[7] M. Zinkevich, Online convex programming and generalized infinitesimal gradient ascent, in *Proceedings of the Twentieth International Conference on Machine Learning (ICML-2003), Washington DC, 2003*, 2003, http://www.aaai.org/Papers/ICML/2003/ICML03-120.pdf.

[8] M. Bowling, Convergence and no-regret in multiagent learning, *Advances in Neural Information Processing Systems* (2005), pp. 209–216.

[9] D. P. Foster and P. H. Young, Learning, hypothesis testing, and Nash equilibrium, Games Econ. Behav. **45**, 73 (2003).

[10] D. P. Foster and R. V. Vohra, Calibrated learning and correlated equilibrium, Games Econ. Behav. **21**, 40 (1997).

[11] D. Fudenberg and D. K. Levine, Conditional universal consistency, Games Econ. Behav. **29**, 104 (1999).

[12] S. Hart and A. Mas-Colell, A simple adaptive procedure leading to correlated equilibrium, Econometrica **68**, 1127 (2000).

[13] S. Hart and A. Mas-Colell, A Reinforcement Procedure Leading to Correlated Equilibrium, in *Economic Essays* edited by, G. Debreu, W. Neuefeind, W. Trockel (Springer, Berlin, Heidelberg, 2001), pp. 181–200, https://link.springer.com/chapter/10.1007/978-3-662-04623-4_12.

[14] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction* (The MIT Press, Cambridge, MA, 2017).

[15] M. Garey and D. S. Johnson, *Computers and Intractability* (W. H. Freeman and Company, San Francisco, 1979).

[16] A. Irpan, Deep reinforcement learning doesn't work yet, in *Sorta Insightful*, 2018, https://www.alexirpan.com/2018/02/14/rl-hard.html.

[17] MVME2100 series VME processor modules data sheet, http://www-csr.bessy.de/control/Hard/IOC/ds0055.pdf.

[18] Y. Gao, J. Chen, T. G. Robertazzi, and K. A. Brown, A success-history based learning procedure to optimize server throughput in large distributed control systems, in 16th International Conference on Accelera- tor and Large Experimental Physics Control Systems (ICALEPCS 2017) Barcelona, Spain, 2017, 2017, https://www.bnl.gov/isd/documents/95405.pdf.

[19] H. V. Hasselt, Double Q-learning, in *Advances in Neural Information Processing Systems 23* (Curran Associates, Inc., Hyatt Regency, Vancouver, Canada, 2010), pp. 2613–2621.

[20] GPIB-488 Programming Reference Manual, https://www.l-com.com/multimedia/manuals/M_USB-488.PDF.