ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THCPL03

A SUCCESS-HISTORY BASED LEARNING PROCEDURE TO OPTIMIZE SERVER THROUGHPUT IN LARGE DISTRIBUTED CONTROL SYSTEMS*

Y. Gao†, J. Chen, T. Robertazzi, Stony Brook University, Stony Brook, NY 11794, USA
K. A. Brown, Brookhaven National Laboratory, Upton, NY 11973, USA

Abstract

Large distributed control systems typically can be modeled by a hierarchical structure with two physical layers: Console Level Computers (CLCs) and Front End Computers (FECs). The control system of the Relativistic Heavy Ion Collider (RHIC) at Brookhaven consists of more than 500 FECs, each acting as a server providing services to a potentially unlimited number of clients. This can lead to a bottleneck in the system, as heavy traffic can slow down or even crash a system, making it momentarily unresponsive. In this paper, we consider this problem from a game theory perspective. Specifically, we consider the case where the server has a varying capacity. First, we model this problem as an integer programming problem. Second, we adopt a regret-based procedure as a basic solution and then propose a success-history based scheme to better accommodate the dynamic server capacity. Finally, simulation results show that both algorithms perform well and lead to a significant improvement of system performance. Moreover, compared with the regret-based procedure, the proposed success-history based scheme results in a higher server throughput and lower crash probability under the dynamic environment.

INTRODUCTION

The control system of the Relativistic Heavy Ion Collider (RHIC) at Brookhaven 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) level and Front End Computers (FECs) level, 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, 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 equipment.

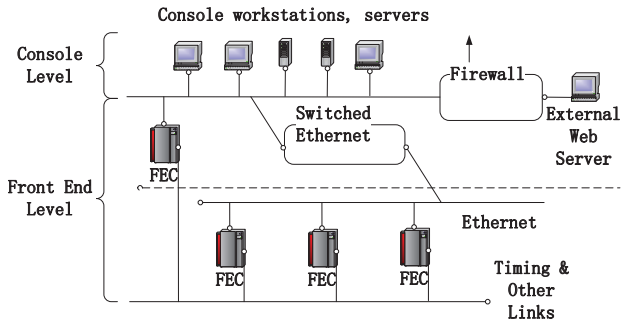


Figure 1: RHIC system hardware architecture.

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 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.

In this paper, we consider a practical performance issue in the front end system, where every FEC acts as a server which holds different kinds of ADOs, providing services to a large number of clients. When the number of clients in an FEC reaches its limit, it slows down the system or even crashes it. When the system crashes, all current applications' communication get lost and it takes time to restore them.

Fig. 2 demonstrates this performance issue. In this example, the arrival procedure of clients' requests is represented by a Poisson process with various rates. For each of those different arrival rates, we measure the ratio of the time spent by the server 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 the performance of the system deteriorates dramatically when the number of clients[1] approaches the server's capacity.

* Work supported by Brookhaven Science Associates, LLC under Contract No. DE-SC0012704 with the U.S. Department of Energy.
† ygao@bnl.gov

[1] Here we assume one client only sends one request at each time. The maximum number of clients the server can hold depends on the size of clients'

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THCPL03

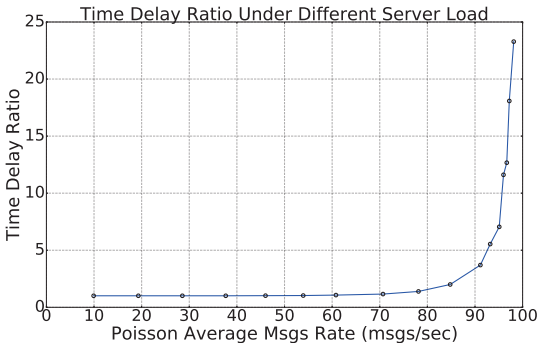


Figure 2: Illustration of the client-server problem.

Moreover, in our system, there are asynchronous processes residing on FECs. Those processes will share FECs' resources when their required information are 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.

We address this client-server problem with dynamic server capacity in a novel approach using game theory. Our goal is to manage clients' behaviors, so that the server's throughput is maximized, and at the same time the server crashes as little as possible.

The main contributions of this paper can be summarized as follows. First, we formulate the client-server problem with varying server capacity into an integer programming problem[2]. Second, to tackle the problem we provide a basic solution to regulate clients' behaviors by adopting a discrete regret-based learning algorithm. Third, in order to better accommodate the dynamic server capacities, we improve the basic algorithm and propose an adaptive procedure, which employs a success-history based scheme to update parameters. Finally through extensive simulations, we demonstrate that both schemes can efficiently manage clients' activities, and produce a significant improvement on both server's throughput and reliability over the case where there is no activity management. Moreover, the proposed success-history based scheme further enhances the regret-based algorithm, resulting in a notably higher server throughput and lower crash probability.

PRELIMINARIES

In this section, we present a brief literature review on game dynamics, followed by a description of the discrete adaptive algorithm [2,3] we adopted, which serves as the basic routine for our parameter adaptation scheme introduced in a later section.

requests and hardware specifications of the server. In this example, it is around 98.

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

Literature Review

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.

In work [4], the authors proposed a multi-agent 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 [5] extended [4] by introducing a varying learning rate to "Win or Learn Fast", and proved convergence in the same game settings as [4]. Work [6] generalized the convergence results of [4] to games with more than two actions, and proved all clients' regrets are bounded by a constant number. Work [7] proposed a discrete procedure which achieves the no-regret result in work [6] and part of the convergence result in work [5], and proved convergence in two-player, two-action games.

There are also many works that study game dynamics for convergence to equilibrium. Work [8] 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-ϵ of time, though almost sure convergence is not achieved. Work [9] 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 [10] offered a class of adaptive procedures called "calibrated smooth fictitious play", which guaranteed almost sure convergence to the set of correlated approximate equilibria.

However, those procedures mentioned above cannot guarantee convergence for n-player ($n > 2$) general-sum games (which is our case), and require global information about the game and observation of the opponent's actions, which is not likely to be available in practical situations. In this paper, we adopt another regret-based procedure that works for any n-player game [2, 3]. It is a discrete algorithm, which does not require sophisticated updating or prediction, and leads the game to converge to the set of correlated equilibria.

A Regret-based Learning Procedure

In this part, we describe the discrete adaptive learning procedure proposed in work [2,3] which leads to the game's correlated equilibria, and is used as the basis for our parameter adaptation scheme proposed later.

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

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THCPL03

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 we call 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 [2, 3] 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. Second, due to the discrete 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 to be played with some minimal frequency.

CLIENT-SERVER PROBLEM WITH DYNAMIC SERVER CAPACITY

In this section, we present the formulation of the client-server problem with dynamic server capacity.

We model the problem as a repeated game, which is played over discrete time slots among n clients (players) and one server[3]. 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 the server. Depending on the result of the stage game as a consequence of clients' actions, payoffs will be assigned to each client in the following way.

At any time t, if the traffic coming from clients exceeds 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 profits equal to their amount of traffic transmitted to the server (L_i for client i). The payoff function for client i can be expressed as:

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

where a_i is client i's action, with value 1 standing for action $Send$ and 0 for $Hold$. Table 1 summarizes these notations.

Suppose the game is played for T periods. 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 server's throughput is maximized:

$$\textbf{Max}: \sum_{t=1}^{T} \sum_{i=1}^{n} a_i^t L_i \quad (2)$$

[3] 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 1: Notations

Notation	Definition
$N, \lvert N\rvert = n$	Set of clients
$A_i = \{S, H\}$	Action set for client i
$L_i, 0 < L_i \leq L_{MAX}$	Amount of traffic client i possesses
L_i as above	Benefit gained for client i from 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

Subject to

$$\sum_{i=1}^{n} a_i^t L_i \leq C_t, \forall t = 1, 2, \ldots, T \quad (3)$$

$$a_i^t \in \{0, 1\}, \forall i = 1, 2, \ldots, n, \forall t = 1, 2, \ldots, T \quad (4)$$

$$L_i \in (0, L_{MAX}], \forall i = 1, 2, \ldots, n \quad (5)$$

where constraint (3) restricts that the server does not crash during the game. Constraint (4) specifies that clients' actions are binary variables, where 1 stands for action $Send$, and 0 for $Hold$. Constraint (5) states that the maximum traffic load a client can have is greater than 0 and below a threshold $L_{MAX} > 0$.

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 well known to be NP-hard (see, for example, [11]). This reduction along with the dynamic aspect of our problem make it very difficult to seek optimal solutions. In the next section, we propose a heuristic scheme to tackle it.

A SUCCESS-HISTORY BASED PARAMETER ADAPTATION SCHEME

In this section, we introduce the proposed parameter update scheme. As mentioned before, we adopt the discrete reinforcement learning algorithm described in [2, 3] as a basic routine, and then build upon it a success-history based adaptive scheme to better accommodate the varying server capacity in our system.

The discrete reinforcement learning algorithm adjusts clients' behaviors based on regret values, which are calculated from clients' realized payoffs. It performs well when the server's capacity does not change. However, due to the dynamic nature of our problem, clients' realized payoffs may reflect their intentions inaccurately. If so, it can cause a slower convergence rate of the algorithm, resulting in a degraded system performance.

In order to properly manage clients' behaviors to handle the dynamic factors, we propose a success-history based

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THCPL03

Index	1	2	...	$H-1$	H
S_{CCF}	$S_{CCF,1}$	$S_{CCF,2}$	...	$S_{CCF,H-1}$	$S_{CCF,H}$
S_{wt}	$S_{wt,1}$	$S_{wt,2}$	...	$S_{wt,H-1}$	$S_{wt,H}$

Figure 3: Successful historical memory of S_{CCF}, S_{wt}.

adaptive scheme to improve the algorithm's performance, which uses a different parameter adaptation mechanism based on a historical record of successful parameter settings.

The general idea of the scheme is to adapt clients' strategies in the game by adjusting their server crash costs. 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 server is crashed (and everything else remained unchanged). Then by applying the regret-based procedure [2, 3], clients' intentions of sending traffic to the server will be suppressed, resulting the server being less crashed, 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 based on the amount of traffic they possess. More precisely, we leverage a parameter called *Crash Cost Factor (CCF)*, denoted by α, to perform the server crash cost adaptation, which is defined as the ratio of server crash cost c and the amount of traffic a client possesses L, e.g. $c_i = \alpha L_i$ for client i. 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.

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

In the beginning, each client randomly chooses a crash cost factor α, denoted by M_{CCF}, from an unified cost factor options, denoted by C_{set} (e.g. $C_{set} = \{1, 5, 10, \cdots, 95, 100\}$), then generates their initial crash cost factors as following:

$$CCF = randn(M_{CCF}, var) \tag{6}$$

where M_{CCF} is the cost factor value selected from the available options C_{set}, and $randn(M_{CCF}, var)$ is the value selected randomly from normal distribution with mean M_{CCF} and variance var.

Then during the game, each client collects statistics about its average amount of effective traffic "$eff\_traffic\_avg$" periodically. 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). If period i's average amount of effective traffic (over the time slots in it) is larger than its previous period's, 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. k is increased whenever a new pair of elements, $S_{CCF,k}$ 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 in Eq. (6). Once all H entries in the memory are filled, with probability p each client generates the next M_{CCF} as indicated in Eq. (7), and with probability $1 - p$ they keep applying the same method in Eq. (6).

$$M_{CCF} = mean_{CCF}(S_{CCF}) \tag{7}$$

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

$$mean_{CCF}(S_{CCF}) = \sum_{k=1}^{H} w_k \cdot S_{CCF,k} \tag{8}$$

$$w_k = \frac{S_{wt,k}}{\sum_{i=1}^{H} S_{wt,i}} \tag{9}$$

The complete adaptive procedure is shown in Algorithm 1. The adaptability of the algorithm to dynamic server capacity is achieved through the following algorithm's 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 (2). Moreover, different server capacities usually need different sets of crash cost factors to cope with. The algorithm makes clients to compare the amount of effective traffic only 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. It 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. Fourth, the generation of actual crash cost factors by a normal distribution focuses on the primary values while adding diversity to the algorithm, which further enhances its adaptability.

PERFORMANCE EVALUATIONS

In this section, we evaluate the performance of the adopted regret-based procedure as well as our proposed

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THCPL03

Algorithm 1 Success-History Based Parameter Adaptation Scheme

Input: Game length T, statistics period length T_s, memory size H, probability p, variance var, cost factor options $\boldsymbol{C_{set}}$.

Initialize average amount of effective traffic $L_{eff,0}$ to be 0, initialize memory updating index $k = 1$.
Randomly uniformly select a value M_{CCF} from $\boldsymbol{C_{set}}$.
Initialize crash cost factor as $CCF = randn(M_{CCF}, var)$.
for $t = 1, 2, \ldots, T$ **do**
  Make strategies according to the regret-based procedure [2, 3].
  if it is time to collect statistics and update crash cost factor **then**
    Calculate average amount of effective traffic $L_{eff,i}$ in current statistic period i.
    if $L_{eff,i}$ is greater than $L_{eff,i-1}$ **then**
      if memory updating index is H **then**
        Reset memory updating index $k = 1$.
      end if
      Store statistic period i's crash cost factor and corresponding weight $L_{eff,i} - L_{eff,i-1}$ into memory location k.
      Update memory updating index $k = k + 1$.
    end if
    if memory is not full **then**
      Randomly uniformly select a value M_{CCF} from $\boldsymbol{C_{set}}$.
      Generate a new crash cost factor as $CCF = randn(M_{CCF}, var)$.
    else
      Randomly uniformly select a value x in [0, 1].
      if $x < p$ **then**
        Calculate M_{CCF} according to Eq. (7).
        Generate a new crash cost factor as $CCF = randn(M_{CCF}, var)$.
      else
        Randomly uniformly select a value M_{CCF} from $\boldsymbol{C_{set}}$.
        Generate a new crash cost factor as $CCF = randn(M_{CCF}, var)$.
      end if
    end if
  end if
end for

success-history based adaptive scheme based on parameters from our control system.

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 we used is equipped with a MVME2100 VME processor module [12]. 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 maximum size[4]. We run the simulations for 12 hours, and assume the server changes capacity every 3 hours[5]. In the next part, we discuss the effects of the parameters in our scheme on the system performance. It also remains an interesting topic for future study to experiment different combinations of those parameters, and verify their impacts on the system performance.

Effects of the Parameters

Several parameters in the proposed scheme need to be set that will influence the overall performance in different ways.

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 successful 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. $C_{set} = \{1, 5, 10, \cdots, 95, 100\}$.

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.

Adaptive Probability p **and Normal Variance** var The adaptive probability decides how much likely the algorithm uses the weighted mean method to update server crash cost for the next period. A large value makes 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

[4] $256 * 8$ bytes.

[5] Since we know the total amount of traffic L_{sum} from clients, every 3 hours, we randomly pick a fraction between 0 and 1 and multiples it with L_{sum}, then the result is the server's new capacity for the next period.

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THCPL03

Table 2: Parameter Settings

Parameter	Value
Number of clients	500
Simulation length	12 hours
Stage game length (one time slot)	1 second
Clients message rate	1 msgs/sec
Maximum traffic load	$256 * 8$ bytes
Server capacity variation period	3 hours
Crash cost factor options	$1, 5, 10, \ldots, 100$
Memory size	20
Statistic period	5 minutes
Adaptive probability	0.9
Normal variance	3

small adaptive probability will impair the adaptive learning process, make it more dominated by random "noise". Similar conclusions can be drawn for normal variance. In this simulation, we use an adaptive probability of 0.9 and a normal variance of 3.

Table 2 summarizes the parameter settings in this simulation.

Simulation Results

In this part, we present the simulation results and demonstrate that, first, both the adopted regret-based procedure and the proposed success-history based 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 reason, we refer to the adopted regret-based procedure as RR scheme and the proposed success-history based procedure as SHB scheme.

Figure 4 to 6 show the comparison results 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 SHB 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 success-history based scheme achieve a significant improvement over the naive case (magenta plot), where there is no regulation on clients' behaviors[6]. 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 success-history based scheme (green plot),

[6] As shown by the dotted 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.

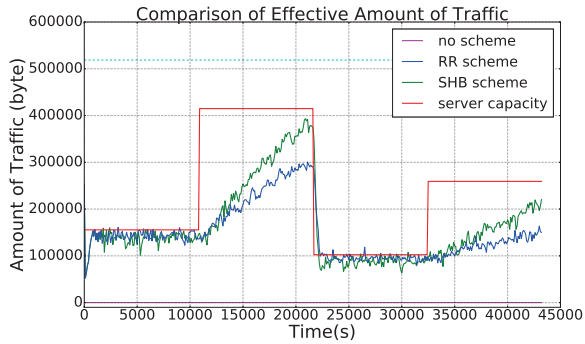
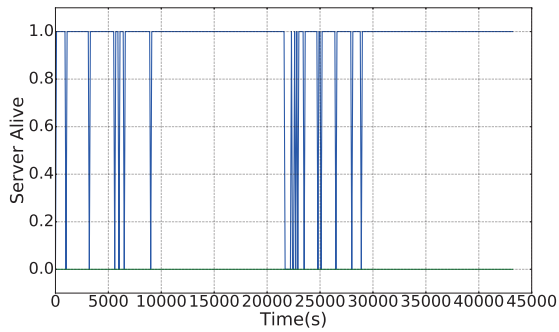
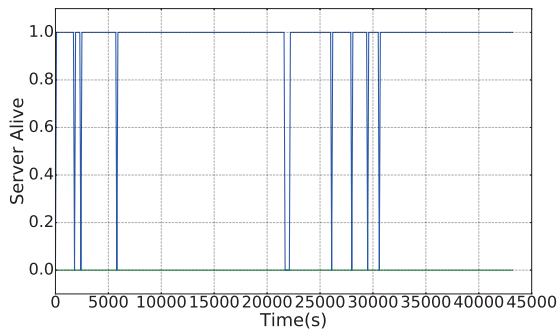


Figure 4: Comparison results of effective amount of traffic routed by clients.



(a) Server alive time using the regret-based scheme.



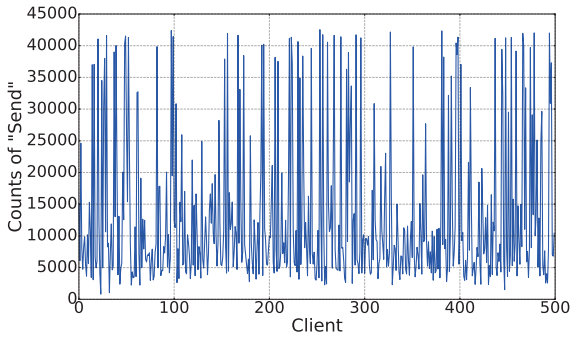
(b) Server alive time using the success-history based scheme.

Figure 5: Comparison results of server alive time.

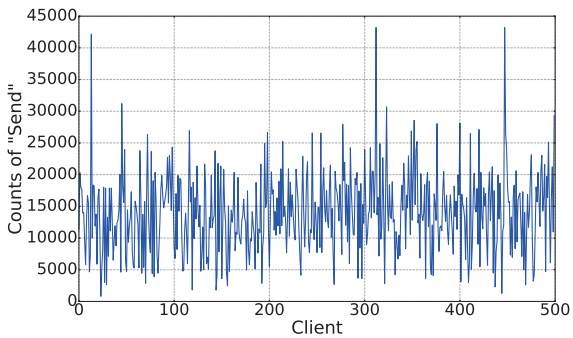
clients adapt their strategies better to the capacity changes compared with the case where they use the regret-based scheme (blue plot), which demonstrates that our proposed scheme can more effectively adjust clients' behaviors so that the server' resources are utilized in a more efficient way.

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, in most of the time both cases achieve a robust server performance, which is a huge improvement than the no regulation case where server crashes all the time (green plots). Furthermore, Fig. 5(b) exhibits a smaller server crash probability, which proves that the proposed success-history based scheme can better accommo-

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THCPL03



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



(b) Counts of "*Send*" for each client using the success-history based scheme.

Figure 6: Comparison results of counts of "*Send*".

date 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 success-history based scheme helps to promote a more equal user experience. From the following chart, we can see that the standard deviation for the counts of "*Send*" among all clients is reduced by 45.5% on average.

Scheme	"*Send*" Count Std.
SHB	6326.699
RR	11599.990

Table 3 summarizes the performance statistics during each server capacity period and the whole simulation time from both the cases where the proposed SHB 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 SHB 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

Table 3: Comparison Results of System Metrics

Scheme	Crash Probability Period #				
	1	2	3	4	All
SHB	0.0446	0.0	0.0815	0.0	0.0315
RR	0.0643	0.0	0.1693	0.0	0.0584
Scheme	Effective Traffic Avg. ($\times 10^5$) Period #				
	1	2	3	4	All
SHB	1.295	2.700	0.795	1.495	1.569
RR	1.285	2.273	0.778	1.225	1.393

server to its full capacity, and the SHB scheme results in a 30.6% and 52% less crash probability than the RR scheme, respectively. On average, the SHB scheme reduces server crash probability by 46%, while at the meantime increasing the server's throughput by 12.7%. It validates that the SHB scheme can more effectively accommodate the dynamic server capacity circumstance in our system than the RR scheme.

CONCLUSIONS

In this paper, we apply game theory approach to analyze a practical performance bottleneck in the RHIC control system, which is the client-server problem with varying server capacity. We formulate it as an integer programming problem and model it as a repeated game. To tackle it, we adopt a discrete regret-based learning procedure as a baseline, then propose a success-history based parameter adaptation scheme to improve the algorithm to better deal with the specialty in our system. Simulation results show that both schemes are efficient on managing clients' behaviors and produce a significant system improvement over the case where there is no activity management. Furthermore, compared with the regret-based scheme the proposed success-history based scheme can more effectively handle the dynamic aspect of the system, and provide a promising server performance improvement.

ACKNOWLEDGEMENT

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.

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THCPL03

REFERENCES

[1] D. S. Barton *et al.*, "RHIC control system", in *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, March 2003, vol. 499, issues 2-3, pp. 356-371.

[2] S. Hart, A. Mas-Colell, "A Simple Adaptive Procedure Leading to Correlated Equilibrium", in *Econometrica*, 2000, vol. 68, No.5, pp. 1127-1150.

[3] S. Hart, A. Mas-Colell, "A Reinforcement Procedure Leading to Correlated Equilibrium", in *Economic Essays*, 2001, pp. 181-200.

[4] 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.

[5] M. Bowling, M. Veloso, "Multiagent Learning Using a Variable Learning Rate", in *Artificial Intelligence*, 2002, pp. 215-250.

[6] M. Zinkevich, "Online Convex Programming and Generalized Infinitesimal Gradient Ascent", in *International Conference on Machine Learning*, 2003.

[7] M. Bowling, "Convergence and No-Regret in Multiagent Learning", in *Advances in Neural Information Processing Systems*, 2005, pp. 209-216.

[8] D. P. Foster, P. H. Young, "Learning, hypothesis testing, and Nash equilibrium", in *Games and Economic Behavior*, 2003, pp. 73-96.

[9] D. P. Foster, R. V. Vohra, "Calibrated Learning and Correlated Equilibrium", in *Games and Economic Behavior*, 1997, pp. 40-55.

[10] D. Fudenberg, D. K. Levine, "Conditional Universal Consistency", in *Games and Economic Behavior*, 1999, pp. 104-130.

[11] M. Garey, D. S. Johnson, *Computers and Intractability*. San Francisco, USA: W. H. Freeman and Company, 1979.

[12] "MVME2100 Series VME Processor Modules Data Sheet", *http://www-csr.bessy.de/control/Hard/IOC/ds0055.pdf*