On Delay Adjustment for Dynamic Load Balancing in Distributed Virtual Environments

Yunhua Deng and Rynson W.H. Lau, *Senior Member, IEEE*

Abstract—Distributed virtual environments (DVEs) are becoming very popular in recent years, due to the rapid growing of applications, such as massive multiplayer online games (MMOGs). As the number of concurrent users increases, scalability becomes one of the major challenges in designing an interactive DVE system. One solution to address this scalability problem is to adopt a multi-server architecture. While some methods focus on the quality of partitioning the load among the servers, others focus on the efficiency of the partitioning process itself. However, all these methods neglect the effect of network delay among the servers on the accuracy of the load balancing solutions. As we show in this paper, the change in the load of the servers due to network delay would affect the performance of the load balancing algorithm. In this work, we conduct a formal analysis of this problem and discuss two efficient delay adjustment schemes to address the problem. Our experimental results show that our proposed schemes can significantly improve the performance of the load balancing algorithm with neglectable computation overhead.

Index Terms—Multi-server architecture, dynamic load balancing, delay adjustment, distributed virtual environments.

1 Introduction

A distributed virtual environment (DVE) allows users to interact with virtual objects or among themselves through networks. Applications of DVEs include massively multiplayer online games (e.g., World of Warcraft), virtual worlds (e.g., Second Life), military and industrial remote training, and collaborative engineering. As the number of concurrent users increases, these systems may reach their performance bottleneck and no longer provide the quality of service required, typically in terms of response time. A general approach to address this scalability problem is to distribute the workload among multiple networked servers, by partitioning the entire virtual environment (VE) into multiple partitions to be served among the servers.

Traditionally, the partitioning process is a static operation and no dynamic changes to the partitions will be allowed during runtime. For example, in [11], the VE is statically partitioned and each partition is assigned to a fixed server. The advantages of this approach are that it is simple and the computation time of the partitioning process would not affect the interactivity of the DVE system. However, as users move around in the VE, some partitions may have too many users due to some application objectives while others may have too few users, resulting in some servers getting overloaded while others underloaded. To address this limitation, dynamic load balancing schemes are introduced, which try to balance the load among the servers dynamically during runtime. Depending on whether a single server (usually referred to as a central server) is needed to perform the load balancing computation, existing dynamic load balancing methods can be generally classified into two approaches: *centralized methods* and *decentralized methods*. In centralized methods, a central server makes the load migration decisions based on the information collected from all local servers, which manage individual partitions, and then passes the decisions to local servers for them to carry out the load migrations. In decentralized methods, each local server could make load migration decisions with information collected from its neighbor servers, which manage the adjacent partitions.

Whether a load balancing method is based on the centralized or decentralized approach, it needs to collect load information from some other servers. If all the servers are located at the same site, the network latency among the servers may be neglectable. However, for large-scale DVEs, such as Second Life, where the servers may be distributed across different cities/continents, the network latency may no longer be neglected. As a result, there will be a delay from the moment that each server sends out its load status to the moment when the collected load status is used to make load migration decisions. As DVE systems are highly interactive and the load on each server may change dramatically over time, by the time when a server is to make the load migration decision, the collected load status from other servers may no longer be valid. This will affect the accuracy, and hence the performance, of the load balancing algorithm, as we demonstrate in Section 5 of this paper. To the best of our knowledge, no works in the area of DVE load balancing have formally studied this problem.

We have two main contributions in this work:

1. We formally model the load migration process across servers by considering the latency among them. Although our work is based on the centralized approach, it can also be applied to the decentralized approach.

2. We propose and compare two efficient delay adjustment schemes to address the latency problem.

We have conducted extensive experiments and our results show that the proposed schemes can significantly improve the performance of the load balancing algorithm, with relatively small overhead.

The rest of the paper is organized as follows. Section 2 summarizes related work on dynamic load balancing. Section 3 models the load migration process across servers in a DVE system. Section 4 presents our two delay adjustment methods: uniform adjustment scheme and adaptive adjustment scheme. Section 5 presents and evaluates a number of experiments.

2 Related Work

In this section, we briefly summarizes relevant load balancing works on the centralized approach and on the decentralized approach, as well as those that concern the delays.

2.1 Centralized Methods

The dynamic load balancing problem in DVEs may be considered as the problem of partitioning a finite element graph and mapping the partitions to different servers. A number of graph partitioning methods have been proposed. Some of them make use of the geometric information of nodes, e.g., the recursive coordinate bisection [1] and inertial bisection [23], which produce regular partitioning, and [24]

- *Yunhua Deng is with Department of Computer Science, City University of Hong Kong, Kowloon, Hong Kong, E-mail: goolor.goolor@gmail.com.*
- *Rynson W.H. Lau is with Department of Computer Science, City University of Hong Kong, Kowloon, Hong Kong, Email: rynson.lau@cityu.edu.hk.*

Manuscript received 15 September 2011; accepted 3 January 2012; posted online 4 March 2012; mailed on 27 February 2012.
For information on obtaining reprints of this article, please send email to: tvcg@computer.org.

[22], which produce irregular partitioning to better fit the VE or game world decomposition. Others make use of the graph connectivity information, e.g., the spectral graph theory based method [21], simulated annealing based method [25] and index based method [20]. In general these methods can produce well balanced partitions even if the load distribution becomes significantly skewed. However, repartitioning from scratch during each partitioning process can be very time consuming, as these methods are not inherently designed for interactive applications. In addition, remapping of the new partitions can cause very high data redistribution / user migration overheads, as the newly generated partitions can deviate considerably from the previous ones.

Instead of repartitioning from scratch using traditional graph partitioning methods, methods based on refining existing partitions and migrating extra loads among the processors have been proposed for finite element analysis [19] [13] [9]. These methods have lower overall overheads. They model the load migration problem as an optimization problem with the objective of minimizing both the load difference among servers and the amount of load needed to be migrated. Derived from the idea in [19] that models the incremental graph partitioning problem as a linear programming problem, an optimization-based load balancing method for multi-server DVEs is proposed in [17]. This method models the loads of a DVE as a connected graph, with each node representing a user and each edge representing the communication cost between the corresponding two nodes. It performs load migrations among adjacent partitions so as to minimize a linear combination of load imbalance among the partitions and the inter-partition communication costs of the reassigned nodes. Although this method may produce very load balancing solutions, it has a very high computational cost, as its optimization process involves all nodes of the DVE. As a result, it is not suitable for DVEs in practice.

In [5], the VE is divided into cells and several heuristic algorithms are proposed to find near optimal assignments of these cells to the servers so as to achieve load balance. However the heuristic algorithms are only practical for small-scale DVEs, due to their high computational overheads. A more efficient method based on heat diffusion is proposed recently [6]. Unlike the above methods, this method replaces the optimization step with multiple iterations of load balancing flow computations during each load balancing process.

In general, most centralized method can produce good load balancing results as they typically execute some kind of optimization algorithms. However, since the load balancing process is performed on a single server, all servers managing the partitions (referred to as the *local servers*) will have to send their load information to this central server. The network delay, together with the long processing time, may become significant. As we will demonstrate in this paper, these delays may affect the accuracy of the load balancing solutions.

2.2 Decentralized Methods

The idea of the decentralized approach is that local servers, i.e., servers managing the partitions, perform the load balancing process individually. Each server will determine the amount of load to be transferred its neighbor servers, which manage the adjacent partitions.

In [18], an adaptive partitioning method is proposed. When a server is overloaded, it uses the load information of its neighboring servers to determine the amount of load to transfer to each of the neighbor servers. This method attempts to minimize the perimeter of the partition as the overloaded server tries to determine which parts of the partition to be transferred. However, as each server only has the load information of its neighbor servers, load balancing decisions are made without a global view of loads of all servers. As a result, the load balancing solutions can only be considered as short-term and the overloaded server may quickly become overloaded again. In [16], a method is proposed to address the limitation of [18] by considering more servers for load transfer. When an overloaded server finds out that it cannot transfer all its extra load to its neighbor servers, it will consider the neighbor servers of its available neighbor servers and so on. When there are enough available servers to take up all the excessive load, the regions managed by all these selected servers will be repartitioned using a graph partitioning method similar to [17]. Although this method may produce better load balancing solutions than [18], it is slower due to the extra communication and computation overheads. In [3], each local server monitors its QoS violations measured in terms of user response time, determines whether a perceived QoS violation is due to heavy workload or high inter-server communication, and then triggers either load shedding or load aggregation, respectively. This method considers not only to migrate load from the overloaded servers, but also to merge the partitions with excessive inter-server communication. This method has a high computational cost. Recently, an efficient hybrid method is proposed to augment the load balancing process running at each local server with some global load information, which may be received infrequently, to guide the direction of load redistribution [15].

In general, decentralized methods are more efficient than centralized methods as they perform the load balancing process by considering only the local load information. Its solutions are typically short-term and the servers may quickly become overloaded again. In addition, although these methods involve a smaller number of servers in each load balancing process, it still needs to obtain the load information from nearby servers in order to determine the target servers and the amount of load to transfer. As two neighbor servers managing two adjacent partitions may not be physically located near to each other, we still cannot neglect the network delay among neighbor servers.

2.3 Delay Consideration in Load Balancing

In some traditional load balancing application domains such as adaptive mesh analysis and queuing analysis, there have been some works that consider delays in the design of the load balancing algorithms. In adaptive mesh analysis, [14] proposes a dynamic load balancing scheme for structured adaptive mesh refinement. The method is divided into global and local load balancing processes, with the objective of minimizing remote communications by reducing the number of load balancing processes among distant processors and allowing more load balancing processes among local processors. Their assumption is that distant processors have high delays while nearby processors have lower delays. In queuing analysis, [10] and [12] conduct extensive analysis and reveal that computational delays and load-transfer delays can significantly degrade the performance of load balancing algorithms that do not account for any delays. These authors subsequently propose to control the load balancing process by a so-call gain parameter and the time when the load balancing process is being executed, with the objective of optimizing the overall completion time of the given tasks in a distributed systems [8]. They try to find out the optimal choice of the load balancing process and the gain parameter empirically through extensive experiments. Further extension of this work in [7] is to consider the random arrivals of the external tasks.

Although these works consider delays, they address very different problems from ours. Both of them try to minimize the task completion time due to network and/or computational delays. On the contrary, we argue here that when the local servers have received the load balancing solutions from the central server after some network delay, the loads of the local servers may be very different and the load balancing solutions may no longer be accurate, due to the dynamic nature of DVEs. We then propose two methods to adjust the load balancing solutions obtained from the central server in order to compensate for the change in server loads.

3 The Load Balancing Algorithm

In [6], a dynamic load balancing method based on the centralized approach is proposed. The idea of this work is to consider the load of a partition, i.e., the computational cost needed to handle the users within the partition, as heat, which can be diffused from the server outwards via the neighbor servers. Due to the simplicity of this method, we discuss our work here based on this method, without loss of generality.

To formally model the load balancing process with latency consideration, we first model the VE by dividing it regularly into a large number of square cells. Each cell may contain some users (or avatars). Here, we assume that the load on each cell is computed according to the number of users within it. In order to map the VE into multiple

servers, the VE is partitioned into n partitions. By assuming that each partition is assigned to a separate server, there will be n servers denoted as $\{S_1, S_2, ...S_n\}$, with S_i serving the i^{th} partition. (For simplicity, we assume for the rest of this paper that references to the partition and the server are interchangeable if the context is clear.) Each partition contains an integer number of cells. Thus, the load of server S_i, denoted as l_i, is the summation of the loads of all the cells inside the i^{th} partition. The load distribution of the n servers of the DVE system is therefore $\{l_1, l_2, ..., l_n\}$.

To formulate the problem, we construct a server graph for the DVE system defined above as $G = (S, E)$, where S represents the set of servers in the DVE system and E represents the set of edges with each edge linking two servers managing the two adjacent partitions. The server graph is a weighted undirected graph. The weight associated with each node S_i is represented by its load l_i, and the weight associated with each edge $e_{i,j}$ is represented by a coefficient called *diffusion coefficient* denoted by $c_{i,j}$, which will be defined later. Fig. 1 shows an example of a small server graph for illustration.

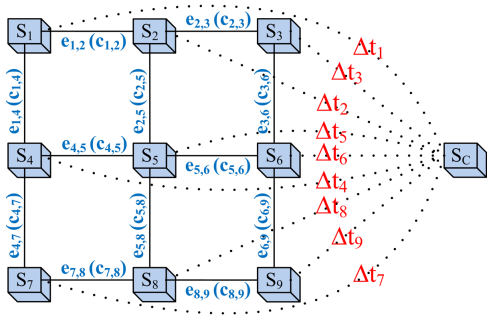

Fig. 1. A server graph with 9 nodes (from S_1 to S_9) representing 9 partitions. Edges (with associated diffusion coefficients shown in the brackets) represent the adjacency among the partitions. The communication link between each server S_i and the central server (denoted as S_C) is shown by a dotted line with the corresponding network delay indicated by Δt_i (from Δt_1 to Δt_9).

Now the load balancing problem becomes how to determine the amount of load to be transferred between any two adjacent nodes such that the load of each node becomes approximately the same. The load transferred between any two adjacent nodes, S_i and S_j, can be viewed as a flow of load (referred to as the *balancing flow*) along edge $e_{i,j}$, and is denoted as $\lambda_{i\to j}$. Note that if $\lambda_{i\to j} > 0$, the direction of the balancing flow is from S_i to S_j, and vice versa.

Given a server graph, we define the load balancing problem as how to find out the balancing flow along each edge. For simplicity, we assume that all servers have the same processing performance. The balancing flow can then be formulated as follows:

$$\bar{l} = l_i - \sum_j \lambda_{i\to j} \tag{1}$$

where $\bar{l}$ is the average load over all nodes, i.e., $\bar{l} = (\sum_k l_k)/\text{n}$. $\sum_j \lambda_{i\to j}$ represents the summation of the balancing flows between S_i and each of its adjacent nodes S_j.

3.1 The Heat Diffusion Algorithm

The load balancing process is to migrate loads from the overloaded servers to the underloaded servers. This is analogous to the heat diffusion process, which is governed by a partial differential equation called the *heat equation*:

$$\frac{\partial u}{\partial t} = \alpha\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) \tag{2}$$

where $u(x, y, t)$ is a temperature function that describes the variation of temperature across spatial location (x, y) at time t, and α is the thermal conductivity. The heat equation is used to determine the change in the temperature function over time as heat spreads throughout space. Solutions of the heat equation are characterized by a gradual smoothing of the initial temperature by the flow of heat from hotter to colder partitions.

Heat diffusion was first used for load balancing in [4], which models the dynamic load balancing problem in a distributed memory multiprocessor network using a discrete form of the heat equation based on the first-order explicit scheme. For each node of the server graph $G = (S, E)$, the execution of the heat diffusion method is divided into a sequence of iteration steps. At k^{th} iteration step, node S_i computes its load using the following equation, assuming that $l_i^{(0)}$ is the initial load of S_i:

$$l_i^{(k)} = l_i^{(k-1)} - \sum_j c_{i,j}(l_i^{(k-1)} - l_j^{(k-1)}) \tag{3}$$

where $l_i^{(k-1)}$ and $l_j^{(k-1)}$ are the loads of node S_i and its adjacent node S_j, respectively, at $k-1$ iteration step. $c_{i,j}$ is the diffusion coefficient that determines the portion of the load difference between S_i and S_j to be exchanged between them. Cybenko [4] also discusses the necessary and sufficient conditions for the convergence of the diffusion method and comes up with the appropriate diffusion coefficients for those graphs with binary n-cube topology (e.g., rings) to guarantee the convergence of the iterative process. For generalized graphs, Boillat [2] proposes to compute the diffusion coefficients as follows:

$$c_{i,j} = \frac{1}{max\{deg(S_i), deg(S_j)\} + 1} \tag{4}$$

where $deg(S_i)$ is the degree of node S_i, i.e., the number of nodes adjacent to it, and $c_{i,j} = c_{j,i}$. The idea here is to increase the diffusion coefficient as the degrees of both adjacent nodes decrease.

3.2 The Global Heat Diffusion Algorithm

The global heat diffusion algorithm has two main processes:

- *Global Migration Planning (GMP)*: it is to compute the balancing flows to solve the load balancing problem using Eq. (1). This process is carried out by the central server.
- *Local Load Transfer (LLT)*: it is to transfer load between adjacent servers based on the amount indicated by the balancing flows. This process is carried out in a distributed way by each local server that manages a partition.

The GMP process is to utilize the global information, the load of each node and the adjacency relations of all nodes, to compute the balancing flows based on the heat diffusion approach. The LLT process is to actually carry out the balancing flows by means of transferring cells between adjacent partitions. Theoretically, the GMP process can be physically located at any one of the local servers, while the LLT process is executed on every local server. We describe each of the two processes in the following subsections.

3.2.1 Global Migration Planning (GMP)

The GMP process is to determine all the balancing flows of the server graph so as to satisfy Eq. (1). It typically takes several iteration steps in order to obtain the final balancing flows.

For a given state of a DVE system, we may construct the server graph $G = \{S, E\}$, with the load distribution among the servers modeled as a vector: $L = (l_1, l_2, ..., l_n)^T$, where n is the number of processors in the server graph. Hence, the initial load distribution vector at the start of the GMP process can be denoted as $L^{(0)} = (l_1^{(0)}, l_2^{(0)}, ..., l_n^{(0)})^T$, while the potential load distribution vector at k^{th} iteration step ($k > 0$) of the GMP process can be denoted as $L^{(k)} = (l_1^{(k)}, l_2^{(k)}, ..., l_n^{(k)})^T$. Here, "potential" means that $L^{(k)}$ does not represent an actual load distribution of the DVE system but just some intermediate values during the iterative GMP process. Hence, given

an initial load distribution vector, $L^{(0)}$, we may compute the potential load distribution vector at any iteration step, k, as follows:

$$L^{(k)} = DL^{(k-1)} \tag{5}$$

where $D = (d_{i,j})_{n\times n}$ is called the *diffusion matrix*. It is given by:

$$d_{i,j} = \begin{cases} c_{i,j} & \text{if } i \neq j, \text{ and } e_{i,j} \in E \\ 1 - \sum_m c_{i,m} & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$$

with $\sum_m c_{i,m}$ representing the sum of the diffusion coefficients associated with node S_i. It is a symmetric irreducible nonnegative matrix, which guarantees the iterative computation defined by Eq. (5) to converge to an equilibrium state, i.e., uniform load distribution, in polynomial time [2]. Using the server graph in Fig. 1 as an example, the diffusion matrix should be:

$$\begin{pmatrix} \frac{1}{2} & \frac{1}{4} & 0 & \frac{1}{4} & & & & & \\ \frac{1}{4} & \frac{3}{10} & \frac{1}{4} & 0 & \frac{1}{5} & & & & \\ 0 & \frac{1}{4} & \frac{1}{2} & 0 & 0 & \frac{1}{4} & & & \\ \frac{1}{4} & 0 & 0 & \frac{3}{10} & \frac{1}{5} & 0 & \frac{1}{4} & & \\ & \frac{1}{5} & 0 & \frac{1}{5} & \frac{1}{5} & \frac{1}{5} & 0 & \frac{1}{5} & \\ & & \frac{1}{4} & 0 & \frac{1}{5} & \frac{3}{10} & 0 & 0 & \frac{1}{4} \\ & & & \frac{1}{4} & 0 & 0 & \frac{1}{2} & \frac{1}{4} & 0 \\ & & & & \frac{1}{5} & 0 & 0 & \frac{3}{10} & \frac{1}{4} \\ & & & & & \frac{1}{4} & 0 & \frac{1}{4} & \frac{1}{2} \end{pmatrix}_{9\times 9}$$

Suppose that at N^{th} iteration step ($N > 0$), we have:

$$L^{(N)} = \begin{pmatrix} l_1 \\ l_2 \\ \vdots \\ l_n \end{pmatrix} = \begin{pmatrix} \bar{l} \\ \bar{l} \\ \vdots \\ \bar{l} \end{pmatrix}$$

Then, the iterative computation process of Eq. (5) would terminate. In fact, we do not really need to achieve such an exact load balance in DVEs. Besides, the loads of DVEs are in terms of number of users, which can only be divided into the nearest integer numbers. A roughly load balance is acceptable in practice so that the iterative GMP process can be terminated earlier. Hence, we have:

$$\begin{cases} L^{(0)} = (l_1^{(0)}, l_2^{(0)}, ..., l_n^{(0)})^T \\ L^{(1)} = (l_1^{(1)}, l_2^{(1)}, ..., l_n^{(1)})^T \\ \vdots \\ L^{(N)} = (l_1^{(N)}, l_2^{(N)}, ..., l_n^{(N)})^T \end{cases}$$

If we rearrange all the load values, we have the load vectors for each of the nodes (or servers) over the N iterations as follows:

$$\begin{cases} l_1 = (l_1^{(0)}, l_1^{(1)}, ..., l_1^{(N)}) \\ l_2 = (l_2^{(0)}, l_2^{(1)}, ..., l_2^{(N)}) \\ \vdots \\ l_n = (l_n^{(0)}, l_n^{(1)}, ..., l_n^{(N)}) \end{cases}$$

where $l_i^{(k)}$ is the potential load of S_i at k^{th} iteration step.

Let us revisit Eq. (1), which formulates the load balancing problem in DVEs as a problem of finding the proper balancing flow along each edge of the server graph. Each iteration step defined by Eq. (5) computes the amount of load that each node of the server graph, S_i, should transfer to (or receive from) each of its adjacent nodes. Thus, the total balancing flow across any edge $e_{i,j}$ is obtained by accumulating the potential loads transferred across $e_{i,j}$ over the total number of iterations, N, before Eq. (5) stops:

$$\lambda_{i\to j} = \sum_{k=0}^{N-1} c_{i,j}(l_i^{(k)} - l_j^{(k)}) \tag{6}$$

Now, if we consider the effect of communication delay, we may denote $L(t) = (l_1(t), l_2(t), ..., l_n(t))$ as the load distribution of n servers at time t, with $l_i(t)$ representing the load of server S_i. We let Δt_i be the communication delay between server s_i and the central server. When the central server receives the load status from S_i at time t, this load status should correspond $l_i(t - \Delta t_i)$, instead of $l_i(t)$, of S_i. Let $L_C(t) = (l_1(t - \Delta t_1), l_2(t - \Delta t_2), ..., l_n(t - \Delta t_n))$ represent the load distribution that the central server holds at time t. We need to modify Eq. (5) as follows:

$$[L_C(t)]^{(k)} = D[L_C(t)]^{(k-1)} \tag{7}$$

where $[L_C(t)]^{(k)}$ is the potential load distribution at the k^{th} iteration step. Now, we may rewrite Eq. (6) to obtain the balancing flow as:

$$\lambda_{i\to j}(t) = \sum_{k=0}^{N-1} c_{i,j}\{[l_i(t - \Delta t_i)]^{(k)} - [l_j(t - \Delta t_j)]^{(k)}\} \tag{8}$$

where $\lambda_{i\to j}(t)$ represents the balancing flow from server S_i to S_j that the central server computes at time t, and $[l_i(t - \Delta t_i)]^{(k)}$ is the potential load of S_i at k^{th} iteration step of the revised GMP process defined in Eq. (7).

Note that the communication delay, Δt_i, for any server i can be generalized to cover the network delay as well as processing delay.

3.2.2 Local Load Transfer (LLT)

The LLT process is to perform the actual load migration, according to the computed balancing flows from the GMP process. Recall that the VE is divided into cells and each cell is considered as a basic unit for transfer among the servers. As each cell may have a different number of users, i.e., a different amount of load, there is not a direct mapping from the amount of load to transfer into the number of cells to transfer. The LLT process running on each local server is to select the boundary cells for transfer between adjacent servers. After all the balancing flows have been carried out, the load for each node should be roughly equalized.

4 The Delay Adjustment Schemes

In the global heat diffusion algorithm, the central server carries out the GMP process to compute the balancing flows for each local server based on the latest load information collected from the local servers. It then propagates the balancing flows to the local servers to carry out the LLT process there. As a result, it takes one single-trip delay for a local server to send its load status to the central server and another single-trip delay for the central server to return the local server the balancing flows. This involves a total of at least a round-trip delay. If the servers are distributed, the effect of this round-trip delay may no longer be neglected, as the load status of a local server from the time that it sends out the load status till the time that it executes the LLT process may have changed significantly. For example, if there are a lot of users moving from this local server to its neighbor servers during this period of round-trip delay, this local server may become underloaded if it carries out the balancing flows recommended by the central server, while its neighbor servers may become overloaded.

We refer to this problem as the delay adjustment problem with the objective of minimizing the effect of the delay on the performance of the load balancing algorithm. This delay adjustment process should be efficient so that it would not introduce significant overhead. Otherwise, the benefit gained from the adjustment may be offset by the additional delay. Likewise, this delay adjustment process should not require extra communication with other servers, which will also introduce additional delay. Here, we present two delay adjustment schemes. The first one is a very simple scheme, which results may serve as reference. The second scheme is more sophisticated but requires some amount of user mobility tracking.

4.1 Uniform Adjustment Scheme

We denote the load of server S_i measured by S_i itself locally at time t as $l_i(t)$, and the load status of S_i available at the central server at time t as $l'_i(t)$. To consider the communication delay, we have:

$$l'_i(t) = l_i(t - \Delta t_i) \tag{9}$$

where Δt_i represents the delay between S_i and the central server, i.e., a single-trip delay. Now supposed that at time t the central server carries out the GMP process to compute the balancing flows and immediately propagates them to the local servers. The load status of S_i used in the GMP process is as shown in Eq. (9) and then it becomes $l_i(t + \Delta t_i)$ by the time when S_i receives the balancing flows from the central server. The load variation of S_i between these two time moments is:

$$\Delta l_i(t - \Delta t_i, t + \Delta t_i) = l_i(t + \Delta t_i) - l_i(t - \Delta t_i) \tag{10}$$

Eq. (10) indicates the net increase in S_i's load between $t - \Delta t_i$ and $t + \Delta t_i$, i.e., the period of a round-trip delay. If $\Delta l_i(t - \Delta t_i, t + \Delta t_i) > 0$, it means that more users have moved into S_i from, than moved out of S_i into, its neighbor partitions.

Based on this observation, we may adjust the balancing flows according to the change in S_i's load. We first assume that S_i has only one neighbor server S_j. If the balancing flow for S_i as recommended by the central server is $\lambda_{i\to j}(t)$, with t corresponding to the time that it was computed. Due to the communication delay, the balancing flow received by S_i is:

$$\lambda'_{i\to j}(t) = \lambda_{i\to j}(t - \Delta t_i) \tag{11}$$

Now, we adjust the balancing flow as follows:

$$\lambda'_{i\to j}(t + \Delta t_i) = \lambda_{i\to j}(t) + \Delta l_i(t - \Delta t_i, t + \Delta t_i) \tag{12}$$

where $\lambda'_{i\to j}(t + \Delta t_i)$ is the adjusted balancing flow in S_i at $t + \Delta t_i$. The idea of this is that, in addition to the recommended balancing flow, S_i should send out the net increase in load obtained during the $2\Delta t_i$ period in order for its load to reach the average level, $\bar{l}$.

Typically, S_i has more than one neighbor servers. We may generalize Eq. (12) simply by uniformly distributing the net increase of S_i's load among the neighbor servers as follows:

$$\lambda'_{i\to j}(t + \Delta t_i) = \lambda_{i\to j}(t) + \Delta l_i(t - \Delta t_i, t + \Delta t_i)/deg(S_i) \tag{13}$$

Note that this is only an approximation method, which assumes that the contribution to the net increase of S_i's load is uniform among S_i's neighbor servers.

4.2 Adaptive Adjustment Scheme

The uniform adjustment scheme is simple as it only needs to capture the load variation as defined by Eq. (10). However, adjusting the balancing flows uniformly across the neighbor servers is only a coarse approximation.

Supposed that S_i has two neighbor servers, S_j and S_k. Then, S_i will receive two balancing flows, $\lambda_{i\to j}(t)$ and $\lambda_{i\to k}(t)$, computed by the central server at time t. Let us denote $\beta_{i\to j}(t - \Delta t_i, t + \Delta t_i)$, $\beta_{i\to k}(t - \Delta t_i, t + \Delta t_i)$, $\beta_{j\to i}(t - \Delta t_i, t + \Delta t_i)$, and $\beta_{k\to i}(t - \Delta t_i, t + \Delta t_i)$ as the number of users having moved from S_i to S_j, S_i to S_k, S_j to S_i, and S_k to S_i during the $2\Delta t_i$ period, respectively. Then the load variation of S_i can be computed as:

$$\begin{aligned}\Delta l_i(t - \Delta t_i, t + \Delta t_i) = {} & \beta_{j\to i}(t - \Delta t_i, t + \Delta t_i) - \beta_{i\to j}(t - \Delta t_i, t + \Delta t_i) \\ & + \beta_{k\to i}(t - \Delta t_i, t + \Delta t_i) - \beta_{i\to k}(t - \Delta t_i, t + \Delta t_i)\end{aligned} \tag{14}$$

where $(\beta_{j\to i}(t - \Delta t_i, t + \Delta t_i) - \beta_{i\to j}(t - \Delta t_i, t + \Delta t_i))$ and $(\beta_{k\to i}(t - \Delta t_i, t + \Delta t_i) - \beta_{i\to k}(t - \Delta t_i, t + \Delta t_i))$ indicate the net increase in load of S_i contributed by S_j and S_k, respectively. If we can capture the exact amount contributed by each neighbor server, we may adjust each balancing flow accurately.

The difficulty for a server to capture the contribution of the net increase in load from each of its neighboring servers is that the server does not have the latest load information of its neighbor servers, unless it sends requests to them to obtain such information. This means that a server has to communicate with its neighboring servers in order to capture the net increase in load contributed by each of them. However, this will introduce additional delay, which itself could cause the same problem. Here, we introduce an adjustment scheme that does not require server-to-server communication.

We notice that the users' maximum moving speed determines how far they can move during the $2\Delta t_i$ period, i.e., between $t - \Delta t_i$ and $t + \Delta t_i$. Hence, the maximum number of cells, N_{cell}, that a user may cross during this period is:

$$N_{cell} = \frac{v_{max} * 2\Delta t_i}{W_{cell}} \tag{15}$$

where v_{max} is the user's maximum moving velocity, and W_{cell} is the width (or height) of each square cell. In most settings, N_{cell} is a small number of 1 or 2. This means that only those users located at the boundary cells of S_i adjacent to S_j may have moved to S_j during the $2\Delta t_i$ period, and vice versa. We propose to track the movement of the users at the boundary cells with a simple method.

To present our method, we assume here that the maximum number of cells that a user may cross is 1 during the $2\Delta t_i$ period. In order to capture the net increase of S_i's load contributed by S_j, we need to determine the amount of users having moved from S_i to S_j, i.e., $\beta_{i\to j}(t - \Delta t_i, t + \Delta t_i)$, and the amount of users having moved from S_j to S_i, i.e., $\beta_{j\to i}(t - \Delta t_i, t + \Delta t_i)$, without any server-to-server communication. For $\beta_{i\to j}(t - \Delta t_i, t + \Delta t_i)$, S_i first marks down all the users who are located at the boundary cells next to S_j at time $t - \Delta t_i$. For each of these users, S_i marks its data structure with a combined ID: Server ID of S_i + Timestamp. We record the total number of marked users as $\omega(t - \Delta t_i)$. At time $t + \Delta t_i$, S_i checks the number of marked users in these boundary cells and their adjacent cells. We record the total number of marked users found as $\omega(t + \Delta t_i)$. Now we have:

$$\beta_{i\to j}(t - \Delta t_i, t + \Delta t_i) \approx \omega(t - \Delta t_i) - \omega(t + \Delta t_i) \tag{16}$$

Fig. 2 illustrate our method to estimate $\beta_{i\to j}(t - \Delta t_i, t + \Delta t_i)$. Here, our assumption is that if the marked users in the shadow area shown in Fig. 2(a) do not appear in the shadow area shown in Fig. 2(b), they should have moved into S_j. We understand that this is only an estimation, as there may be some users that have moved out of the shadow area shown in Fig. 2(a) through the upper or lower boundary into the upper or lower neighbor partition. However, we believe that this is a much better approximation than that of the uniform adjustment scheme, as we will demonstrate in Section 5.

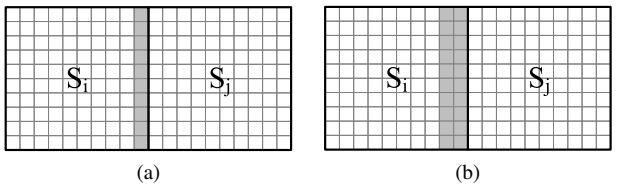

Fig. 2. (a) S_i marks the users located at the boundary cells (shadow area) at time $t - \Delta t_i$; (b) S_i counts the number of marked users at the boundary cells and their inner adjacent cells (shadow area) at time $t + \Delta t_i$.

To estimate $\beta_{j\to i}(t - \Delta t_i, t + \Delta t_i)$, S_i marks down all the users who are located at the boundary cells next to S_j as well as their inner adjacent cells at time $t - \Delta t_i$, i.e., the shadow area shown in Fig. 3(a). For each of these users, S_i marks in a separate location of its data structure with a combined ID: Server ID of S_j + Timestamp. At time $t + \Delta t_i$, S_i checks the number of users in the boundary cells, i.e., the shadow area shown in Fig. 3(b), who do not have such a marking. Based on these two numbers, we may estimate the number of users having moved

from S_j into S_i, i.e., the $\beta_{j\to i}(t-\Delta t_i, t+\Delta t_i)$. Here, our assumption is that all the unmarked users found in the shadow area shown in Fig. 3(b) should have come from S_j. Again, this is only an estimation, but it should be a good estimation.

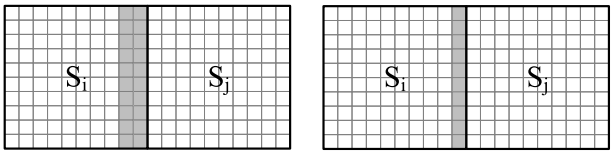

Fig. 3. (a) S_i marks the users located at the boundary cells and their inner adjacent cells (shadow area) at time $t-\Delta t_i$; (b) S_i counts the number of unmarked users at the boundary cells (shadow area) at time $t+\Delta t_i$.

After finding $\beta_{i\to j}(t-\Delta t_i, t+\Delta t_i)$ and $\beta_{j\to i}(t-\Delta t_i, t+\Delta t_i)$, we can compute the net increase of S_i's load contributed by S_j according to Eq. (14). Hence, the adjusted balancing flow to S_j is:

$$\lambda'_{i\to j}(t+\Delta t_i) = \lambda_{i\to j}(t) + (\beta_{j\to i}(t-\Delta t_i, t+\Delta t_i) - \beta_{i\to j}(t-\Delta t_i, t+\Delta t_i)) \quad (17)$$

5 Simulations and Evaluations

We have implemented a simulator in C++ to model a multi-server DVE system for evaluations. We have integrated the dynamic load balancing algorithm used here and the two proposed delay adjustment schemes into this simulator. All these algorithms are also written in C++. We have conducted a number of experiments to verify the effectiveness of the proposed methods. All the experiments were conducted on a PC with an Intel Core i7 2.80GHz CPU and 6GB RAM.

5.1 Simulation Settings

We have simulated a VE of $1,000m \times 1,000m$ in size. It is subdivided into a set of 100×100 cells. Hence, the size of each cell is $10m \times 10m$. The cells are then divided into 100 partitions and managed by 100 servers. We model a total of 10,000 users and divide them into two groups. One group of 8,000 users move randomly in the VE. The other group of 2,000 users move as a crowd in the VE, causing different parts of the VE to get overloaded as they pass by. This is to evaluate the effectiveness of our delay adjustment schemes under such a demanding situation. The moving speed of each user is randomly chosen from the range of [0, $10m/s$], with an average moving speed of $5m/s$, which is roughly equal to the human running speed.

The single-trip communication delay between each local server and the central server is randomly selected from the range of [$100ms$, $300ms$] at each time frame. The system frame rate is set as 10 frames per second, which is also the frequency that local servers send their load information to the central server. The duration of each experiment is 2 minutes. We typically run the simulation for a short while so that the DVE system is stabilized before we start capturing the statistical and performance information. We have experimented two load balancing intervals: 1 second and 3 seconds. This refers to the duration between two consecutive load balancing processes.

To determine the maximum number of cells, N_{cell}, that a user may cross during the $2\Delta t_i$ period stated in Eq. (15), we have $v_{max} = 10m/s$, maximum $\Delta t_i = 300ms$, and $W_{cell} = 10m$. Hence, N_{cell} should be 1. This is used in the adaptive adjustment scheme for object tracking, as discussed in Section 4.2.

5.2 Performance Metrics

We define two performance metrics to measure the performance of the load balancing algorithm. First, we define the *ratio of overloaded servers* (R_{OS}) as:

$$R_{OS} = \frac{number\ of\ overloaded\ servers}{total\ number\ of\ servers} \times 100(\%)$$

which indicates the percentage of servers being overloaded. It is measured in a 1-second interval, i.e., we count the average number of overloaded servers over the number of frames in 1 second. (Hence, if a server is overloaded in 3 out of 10 frames, it is counted as 0.3 overloaded servers.) Here, a server S_i is regarded as overloaded if $l_i > l_{max}$, where l_{max} is the maximum load that a server can handle. As we have 10,000 users served by 100 servers, the average load, $\bar{l}$, is 100. In order to observe how the load balancing performance in a heavy loaded VE, we set $l_{max} = 120$. Since overloaded servers would affect the system interactivity, this metric reflects the ability of the load balancing algorithm to avoid server overloading. The value of R_{OS} should be the lower the better.

Second, we define the *ratio of migrated users* (R_{MU}) as:

$$R_{MU} = \frac{number\ of\ migrated\ users}{total\ number\ of\ users} \times 100(\%)$$

which indicates the percentage of users being migrated in a single load balancing process. This metric reflects the overhead induced by the load balancing algorithm. The value of R_{MU} should be the lower the better.

In addition to the above two metrics, the computational overheads are also important to the load balancing algorithm. Since the global heat diffusion method has two main processes, global migration planning (GMP) and local load transfer (LLT), we would measure the computation time of each process separately. While the computation time for the GMP process is determined by the central server, the computation time for the LLT process is determined by the longest time among all the local servers in executing the LLT process.

We show three sets of experiments in the following subsections In these experiments, we compare four methods: 1) ***original*** - the original global heat diffusion algorithm without delay adjustment; 2) ***uniform*** - the original algorithm with the uniform adjustment scheme; 3) ***adaptive*** - the original algorithm with the adaptive adjustment scheme; 4) ***ideal*** - the original algorithm but assuming that there is no server-to-server delay.

5.3 Experiment #1

This experiment studies the effects of the proposed adjustment schemes on the performance of the load balancing algorithm based on the ratio of the overloaded servers, R_{OS}. We compare the four methods mentioned in Section 5.2 and the results are shown in Fig. 4, when the load balancing interval is set to 1 second.

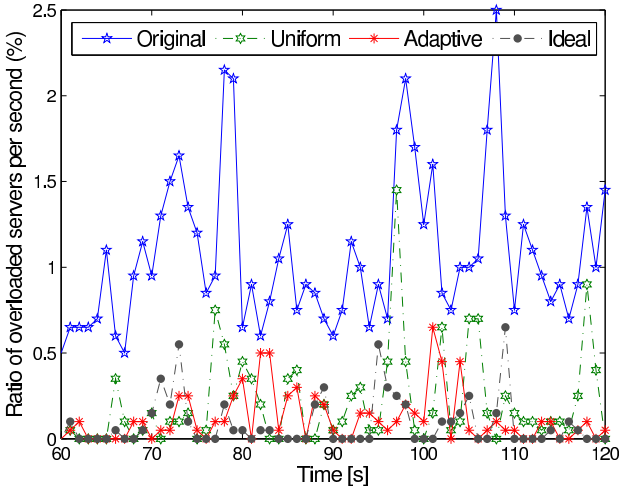

Fig. 4. R_{OS} performance at load balancing interval = 1 second.

We can see that the ideal method has the lowest R_{OS}, as we assume no communication delay among the servers. This represents the best case. On the other hand, the original method suffers from a delay of $2\Delta t_i$. We can see that R_{OS} increases significantly. With the uniform

adjustment scheme, we can see that R_{OS} is much lower than that of the original method, although it is higher than that of the ideal method. With the adaptive adjustment scheme, the performance is even better. In fact, it is very close to ideal method in some places. This shows that our proposed two delay adjustment schemes are effective, with the adaptive adjustment scheme being better due to the additional tracking of user movement. This shows that by improving the accuracy of the balancing flows, the load distribution among the local servers becomes more even. As a result, the local servers are less likely to get overloaded.

Fig. 5 shows the R_{OS} performance of the four methods when the load balancing interval is set to 3 seconds. We can see that R_{OS}'s of all methods are now higher, as a result of a longer interval before each load balancing process. However, the performance improvements of the two adjustment schemes are still obvious. Again, the adaptive adjustment scheme performs better than the uniform adjustment scheme.

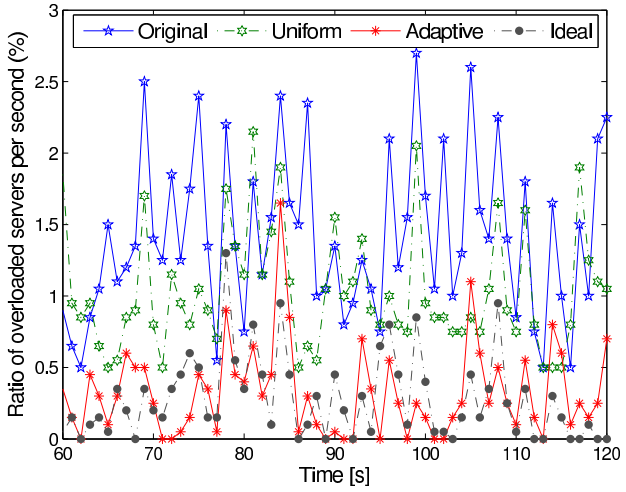

Fig. 5. R_{OS} performance at load balancing interval = 3 seconds.

Fig. 6 shows the average R_{OS}'s of the four methods. We can see that at both load balancing intervals, the average performance of the adaptive adjustment scheme is very close to that of the ideal method. However, we can see that the average performance of the uniform adjustment scheme is much worst when the load balancing interval is high. This is because as the load balancing interval is high, the loads among the local servers become more uneven. If we simply distribute the load variation evenly among the neighbor servers during the load balancing process, it may further upset the balance of load for some of the local servers, causing them easier to get overloaded.

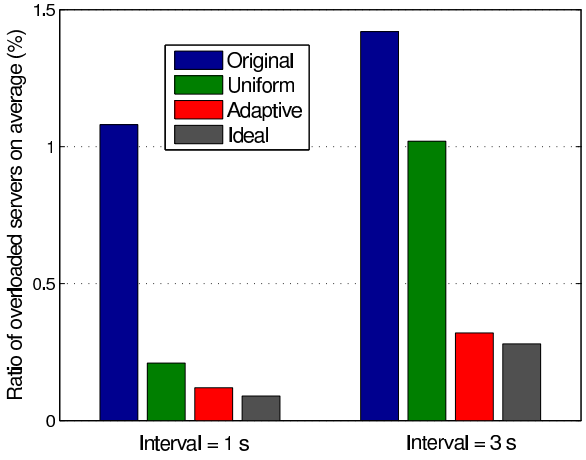

Fig. 6. Average R_{OS} performance at two load balancing intervals.

5.4 Experiment #2

This experiment studies the effects of the proposed adjustment schemes on the performance of the load balancing algorithm based on the ratio of migrated users, R_{MU}.

Fig. 7 shows the ratio of migrated users of the four methods when the load balancing interval is set to 1 second. Again, we can see that the ideal method has the lowest R_{MU}, the original method has the highest, while the two adjustment schemes are in the middle, with the adaptive adjustment scheme being slightly better than the uniform adjustment scheme. This shows that by improving the accuracy of the balancing flows, we can avoid unnecessary migration of users among the local servers. As a result, the overall amount of users needed to be migrated is reduced.

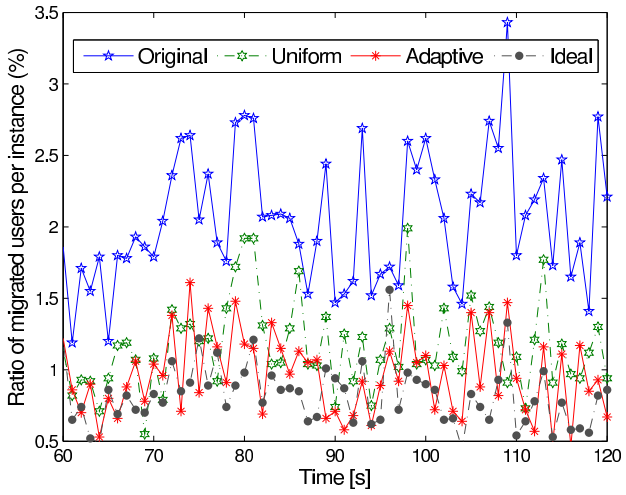

Fig. 7. R_{MU} performance at load balancing interval = 1 second.

Fig. 8 shows the R_{MU} performance of the four methods when the load balancing interval is set to 3 seconds. We can see that R_{MU}'s of all methods are now higher, as a result of a longer interval before each load balancing process. The performance improvements of the two adjustment schemes are still obvious, with the adaptive adjustment scheme performs better than the uniform adjustment scheme.

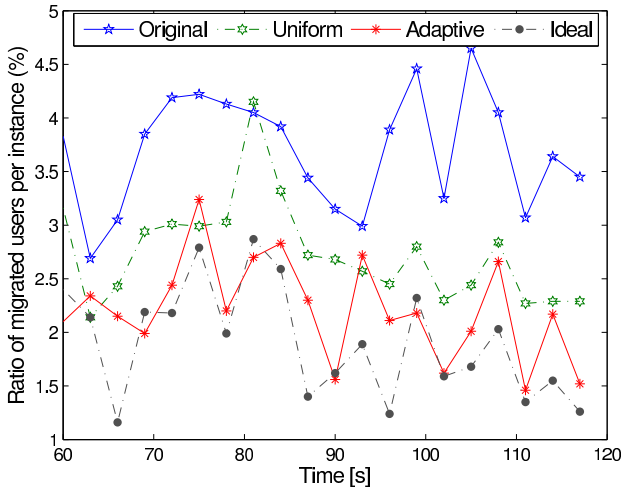

Fig. 8. R_{MU} performance at load balancing interval = 3 seconds.

Fig. 9 shows the average R_{MU}'s of the four methods. We can see that the average R_{MU} of the idea method and the two adjustment schemes are very close when the load balancing interval is set to 1 second. When the load balancing interval increase to 3 seconds, the average R_{MU} of all four methods roughly increases proportionally.

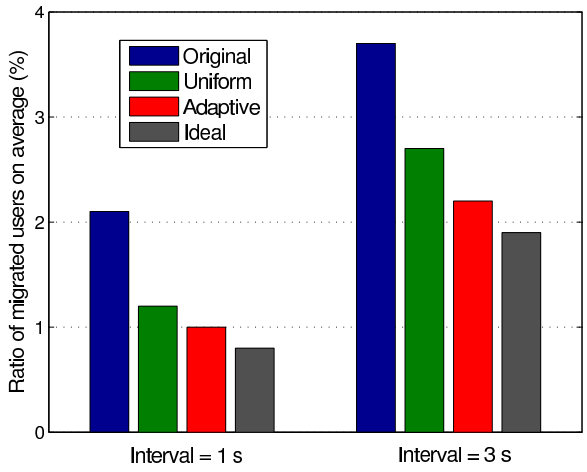

Fig. 9. The average R_{MU} with two load balancing intervals.

5.5 Experiment #3

This experiment studies the computational overhead of the proposed adjustment schemes. The two adjustment schemes are added to the beginning of the LLT process and are executed at the local servers. Table 1 shows the computation time of the four methods. We can see that the computation time of the LLT process for the four methods is roughly similar (all below 2*ms*). This means that the two adjustment schemes do not add obvious computational costs on the load balancing algorithm. On the other hand, we notice that with the adjustment schemes, there is an obvious decrease in the computation time of the GMP process. This is mainly because the two adjustment schemes help improve the load distribution among the local servers. As a result, it generally takes fewer iteration steps for the GMP process to terminate.

When the load balancing interval is set to 3 seconds, the load distribution among the local servers are now less even, as a result of a longer interval before each load balancing process. We can see that for the GMP process, the computation time of the original method is much higher now. However, the computation time of the uniform adjustment scheme is also very high compared with that of the adaptive adjustment scheme. As we have explained in Experiment #1, if we simply distribute the load variation evenly among the neighbor servers during the load balancing process, it may further upset the balance of load for some of the local servers. This in turn will cause the GMP process to execute more iteration steps before it terminates.

Table 1. Computational runtime (ms).

Interval	Scheme	GMP Process	LLT Process	Total
1 s	***Original***	4.03	1.69	5.72
	Uniform	2.44	1.75	4.19
	Adaptive	2.51	1.93	4.44
	Ideal	3.00	1.88	4.88
3 s	***Original***	10.37	1.74	12.11
	Uniform	9.95	1.58	11.53
	Adaptive	6.58	1.89	8.47
	Ideal	6.21	1.63	7.84

Through the above experiments, we can see that the proposed adjustment schemes have obvious improvement on the overall performance of the load balancing method when the server-to-server latency cannot be neglected. Although the uniform adjustment scheme is much simpler, the adaptive adjustment scheme outperforms the uniform adjustment schemes in all three experiments.

6 Conclusion

In this paper, we have investigated the delay problem on dynamic load balancing for DVEs. Due to communication delays among servers, the load balancing process may be using outdated load information from local servers to compute the balancing flows, while the local servers may be using outdated balancing flows to conduct load migration. As we have shown, this would significantly affect the performance of the load balancing algorithm. To address this problem, we have presented two methods here: uniform adjustment scheme and adaptive adjustment scheme. The first method performs a uniform distribution of the load variation among the neighbor servers, which is a coarse approximation but is very simple to implement. We use it more as a reference method for comparison here. The second method performs limited degree of user tracking but without the need to communicate with neighbor servers. It is a better approximation method. Our experimental results show that both adjustment schemes can greatly improve the performance of the load balancing algorithm in most situation, with the adaptive adjustment scheme performs even better on average in the experiments. To the best of our knowledge, this is the first work that formally addresses the delay problem in DVE load balancing.

Acknowledgments

The authors would like to thank the anonymous reviewers for their helpful and constructive comments on this paper. The work described in this paper was partially supported by a GRF grant from the Research Grants Council of Hong Kong (RGC Ref.: CityU 116010) and a SRG grant from City University of Hong Kong (Project Ref.: 7002664).

References

[1] M. Berger and S. Bokhari. A partitioning strategy for nonuniform problems on multiprocessors. *IEEE Trans. on Computers*, 36(5):570–580, 1987.

[2] J. Boillat. Load balancing and poisson equation in a graph. *Concurrency: Practice and Experience*, 2(4):289–313, 1990.

[3] J. Chen, B. Wu, M. Delap, B. Knutsson, H. Lu, and C. Amza. Locality aware dynamic load management for massively multiplayer games. In *Proc. ACM SIGPLAN Symp. on PPoPP*, pages 289–300, 2005.

[4] G. Cybenko. Dynamic load balancing for distributed memory multiprocessors. *Journal of Parallel and Distributed Computing*, 7(2):279–301, 1989.

[5] B. De Vleeschauwer, B. Van Den Bossche, T. Verdickt, F. De Turck, B. Dhoedt, and P. Demeester. Dynamic microcell assignment for massively multiplayer online gaming. In *Proc. ACM SIGCOMM Workshop on Network and System Support for Games*, pages 1–7, 2005.

[6] Y. Deng and R. Lau. Heat diffusion based dynamic load balancing for distributed virtual environments. In *Proc. ACM VRST*, pages 203–210, 2010.

[7] S. Dhakal, M. Hayat, J. Pezoa, C. Yang, and D. Bader. Dynamic load balancing in distributed systems in the presence of delays: A regeneration-theory approach. *IEEE Trans. on Parallel and Distributed Systems*, 18(4):485–497, 2007.

[8] S. Dhakal, B. Paskaleva, M. Hayat, E. Schamiloglu, and C. Abdallah. Dynamical discrete-time load balancing in distributed systems in the presence of time delays. In *Proc. IEEE Conference on Decision and Control*, volume 5, pages 5128–5134, 2003.

[9] R. Diekmann, R. Preis, F. Schlimbach, and C. Walshaw. Shape-optimized mesh partitioning and load balancing for parallel adaptive FEM. *Parallel Computing*, 26:1555–1581, 2000.

[10] J. Douglas Birdwell, J. Chiasson, Z. Tang, C. Abdallah, M. Hayat, and T. Wang. Dynamic time delay models for load balancing. Part I: Deterministic models. In *Proc. CNRS-NSF Workshop: Advances in Control of Time-Delay System*, 2003.

[11] T. Funkhouser. Ring: A client-server system for multi-user virtual environments. In *Proc. ACM I3D*, pages 85–92, 1995.

[12] M. Hayat, S. Dhakal, C. Abdallah, J. Douglas Birdwell, and J. Chiasson. Dynamic time delay models for load balancing. Part II: A stochastic analysis of the effect of delay uncertainty. In *Proc. CNRS-NSF Workshop: Advances in Control of Time-Delay System*, 2003.

[13] Y. Hu, R. Blake, and D. Emerson. An optimal migration algorithm for dynamic load balancing. *Concurrency: Practice and Experience*, 10(6):467–483, 1998.

[14] Z. Lan, V. Taylor, and G. Bryan. Dynamic load balancing of SAMR applications on distributed systems. In *Proc. ACM/IEEE Conference on Supercomputing*, pages 24–24, 2001.

[15] R. Lau. Hybrid load balancing for online games. In *Proc. ACM Multimedia*, pages 1231–1234, 2010.

[16] K. Lee and D. Lee. A scalable dynamic load distribution scheme for multi-server distributed virtual environment systems with highly-skewed user distribution. In *Proc. ACM VRST*, pages 160–168, 2003.

[17] J. Lui and M. Chan. An efficient partitioning algorithm for distributed virtual environment systems. *IEEE Trans. on Parallel and Distributed Systems*, 13(3):193–211, 2002.

[18] B. Ng, A. Si, R. Lau, and F. Li. A multi-server architecture for distributed virtual walkthrough. In *Proc. ACM VRST*, pages 163–170, 2002.

[19] C. Ou and S. Ranka. Parallel incremental graph partitioning. *IEEE Trans. on Parallel and Distributed Systems*, 8(8):884–896, 1997.

[20] C. Ou, S. Ranka, and G. Fox. Fast and parallel mapping algorithms for irregular problems. *The Journal of Supercomputing*, 10(2):119–140, 1996.

[21] A. Pothen, H. Simon, and K. Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM Journal on Matrix Analysis and Applications*, 11(3):430–452, 1990.

[22] K. Prasetya and Z. Wu. Performance analysis of game world partitioning methods for multiplayer mobile gaming. In *Proc. ACM SIGCOMM Workshop on Network and System Support for Games*, pages 72–77, 2008.

[23] H. Simon. Partitioning of unstructured problems for parallel processing. *Computing Systems in Engineering*, 2(2-3):135–148, 1991.

[24] A. Steed and R. Abou-Haidar. Partitioning crowded virtual environments. In *Proc. ACM VRST*, pages 7–14, 2003.

[25] R. Williams. Performance of dynamic load balancing algorithms for unstructured mesh calculations. *Concurrency: Practice and Experience*, 3(5):457–481, 1991.