Multi-Server Support for Large Scale Distributed Virtual Environments

Beatrice Ng, Rynson W.H. Lau, Antonio Si, and Frederick W.B. Li

***Abstract*— CyberWalk is a distributed virtual walkthrough system that we have developed. It allows users at different geographical locations to share information and interact within a shared virtual environment (VE) via a local network or through the Internet. In this paper, we illustrate that as the number of users exploring the VE increases, the server will quickly become the bottleneck. To enable good performance, CyberWalk utilizes multiple servers and employs an adaptive region partitioning technique to dynamically partition the whole VE into regions. All objects within each region will be managed by one server. Under normal circumstances, when a viewer is exploring a region, the server of that region will be responsible for serving all requests from the viewer. When a viewer is crossing the boundary of two or more regions, the servers of all the regions involved will be serving requests from the viewer since the viewer might be able to view objects within all these regions. This is analogous to evaluating a database query using a parallel database server, which could improve the performance of serving a viewer's request tremendously. We evaluate the performance of this multi-server architecture of CyberWalk via a detail simulation model.**

***Index Terms*—Multi-server architecture, adaptive region partitioning, distributed virtual environments.**

I. Introduction

CyberWalk [7], [8] is a distributed virtual walkthrough prototype system that we have developed. It allows users at different geographical locations to share information and interact within a shared virtual environment (VE) via a local network or through the Internet. This VE may represent a physical museum or place of interest. It may also represent a place to be constructed, such as a planned theme park. CyberWalk employs a standard client-server architecture. Information on virtual objects, including their locations and shapes, are maintained in a central database server. When a viewer (user) walks through a VE, geometry information of the virtual objects located within a visible distance from the viewer will be conveyed to the viewer's client machine. The information will then be processed and rendered into images to be viewed in a timely fashion. In general, the virtual objects could be dynamic, changing their locations and orientations within the VE. However, since the current CyberWalk prototype supports only static objects (i.e., objects cannot be changed/modified), we focus on VEs with static objects only in this paper.

Our goal in this project is to provide good performance for virtual walkthrough with Cyber-Walk, both in terms of responsiveness and image quality, under the existing constraint of relatively low Internet bandwidth. In [8], we have introduced a multi-resolution caching mechanism, which allows frequently accessed virtual objects to be cached in a client's local storage at appropriate resolutions. The caching mechanism is also complemented with a prefetching mechanism, which attempts to predict the movement of a viewer and transmits the objects to the client in advance. We have verified that the caching and prefetching mechanisms provide impressive improvement in system performance.

In this paper, we illustrate that when the number of viewers exploring the VE increases, the server will quickly become the bottleneck through serving queries from the clients, which include retrieving and transmitting the requested object models to the clients. CyberWalk addresses this problem by employing parallelism using an array of object servers. We propose an *adaptive region partitioning technique* to partition the whole VE into multiple regions. All objects within a region will be managed by one server. Requests from viewers for any object within a region will be served by the server managing that region. This reduces the number of viewers that each server needs to handle. When a viewer is crossing the boundary of two or more regions, all the servers of the relevant regions will be serving requests from the viewer since the viewer may be able to view objects within all these regions. However, when a server is overloaded by a large number of requests due to too many clients accessing its region, the managed region will be partitioned and part of it will be transferred to a lightly loaded neighbor server. We illustrate in this paper the performance improvement of CyberWalk using this parallel architecture.

The rest of the paper is organized as follows. Section II describes relevant research work. Section III introduces the parallel server architecture of CyberWalk. Section IV presents the load balancing mechanism through the adaptive region partitioning scheme. Section V discusses the

Manuscript received December 11, 2002; revised February 28, 2004; approved by IEEE Transactions on Multimedia Editor W. T. Chang. This work was supported in part by two CERG grants from the Research Grants Council of Hong Kong (RGC Reference Numbers: CityU 1080/00E and CityU 1308/03E).

Beatrice Ng and Rynson W.H. Lau are with City University of Hong Kong, Hong Kong (e-mail: rynson@cs.cityu.edu.hk).

Antonio Si is with the Oracle Corporation, Redwood Shores, CA 94065, USA (e-mail: antonio.si@oracle.com).

Frederick W.B. Li is with The Hong Kong Polytechnic University, Hong Kong (e-mail: csbor@comp.polyu.edu.hk).

performance of CyberWalk via several simulated experiments. Finally, section VI briefly concludes the paper.

II. Related Work

In this section, we introduce existing multi-server techniques for distributed virtual environments and compare local approach with global approach to load balancing. We then give an overview of our CyberWalk prototype system and discuss the unique features of CyberWalk to achieving load balancing.

A. *Multi-servers for Distributed Virtual Environments*

As the number of clients accessing a VE increases, the amount of messages needed to be sent among them increases dramatically. The server loading in managing the VE and handling the messages also increases significantly. Traditionally, there are two types of architecture for multiple clients to participate in a VE, *peer-to-peer* and *client-server*.

Earlier systems, such as NPSNET [13] and DIVE [3], are implemented in a peer-to-peer architecture. This approach has minimal communication overheads, but may not scale well to handle many simultaneous clients due to the saturation of network bandwidth in handling broadcast or multicast messages from the clients. To improve scalability, systems such as BrickNet [24], Community Place [20] and MASSIVE-3 [16], are implemented in client-server architecture. With this approach, each client sends messages to the server for further propagation to other clients and/or servers in the same VE. The advantage of this approach is that the server may perform message filtering to minimize the amount of messages needed to be handled by each client and to be propagated through the network. The major limitation, however, is that as the number of clients accessing the VE increases, the amount of messages needed to be sent among them increases dramatically. The server loading in managing the VE and handling the messages also increases significantly. Another problem is that the server may potentially become a single point of failure.

A distributed VE system with a multi-server architecture could solve these problems. The VE may be partitioned into regions, and each of which is assigned to a separate server, distributing the workload among them. This may also prevent the single point of failure problem if clients can be connected to different servers dynamically. Systems of this approach include RING [14], NetEffect [9] and CittaTron [18].

In RING [14], the VE is partitioned statically and each region is assigned to a fixed server. With this approach, some servers may still be overloaded if a large number of clients converge to some particular regions of the VE. A client in RING, however, may choose to connect to a server statically or dynamically based on its current position. In NetEffect [9], the VE is partitioned dynamically based on client density of individual communities (regions), and it is possible for one server to handle multiple communities. A master server is responsible for performing the load balancing duty. It periodically checks the client density of all communities and performs load balancing by transferring communities among the servers. In addition, a client may be migrated to a different community of another server when its community is overcrowded. However, after the client is migrated to a new community, it needs to wait for the client machine to download the scene of the new community. In CittaTron [18], the VE is partitioned into regions dynamically based on the server loading as a result of the number of clients in the region. Each region is assigned to a unique server. The size of each region may be adjusted during run-time and clients may be transferred among the servers in order to achieve load-balancing. However, factors such as object density and locality are not considered for load balancing.

Existing multiplayer online games have already implemented with distributed game servers. For example, Quake III Arena [22] and Diablo II [11] offer a list of game servers for clients to join. However, each game server maintains a unique game state, which is not shared among the servers. This is essentially a set of separated client-server systems running the same game and may not be considered as a real multi-server system. EverQuest [12], in contrast, divides the entire VE into distinct zones, and maintains these zones by individual game servers. EverQuest allows a client to travel from one zone (game server) to another freely. Ultima Online [25] and Asheron’s Call [1] adopt similar approach as EverQuest, but they divide the entire VE into visually continuous zones. The boundary of each zone is mirrored at neighbor server to reduce the lag problem and to improve interactivity when a user crosses from one zone to another. In addition, Asheron’s Call is technically more advanced in that it may dynamically transfer a portion of the zone controlled by a given game server to any other lightly loaded server. Unfortunately, object coherency is not considered.

B. *Local and Global Load Balancing*

There are two major approaches to load balancing in traditional distributed systems, global and local. Global approach considers the loading information of all servers during load balancing. Usually, a single processor is responsible for collecting the loading information from all processors in the system, making the load balancing decisions, and organizing the load migration processes. Local approach, on the other hand, considers the loading information of local servers only when making the load balancing decisions for a server. In general, global approach can produce more accurate balancing decisions as the processors responsible for handling the load balancing process have the loading information of the entire system. However, it involves high overheads on network bandwidth consumption and network delay in collecting the large amount of loading information from all the servers as well as processing all the collected information. These overheads are proportional to the scale of the system. On the other hand, local approach is generally less accurate in making the load balancing decisions but it is more efficient, in particular, for large systems [26].

C. *Overview of CyberWalk*

Most existing distributed VE systems employ a complete replication approach by distributing the complete geometry database to the clients before the start of the application [2],

[3], [21]. As the geometry database is usually large in size, this approach assumes that the clients will obtain the database through a high speed network or from some other means, such as CDROM distribution. Another approach to distribute geometry data is to send them on demand to the clients at run-time [13], [23]. When the VE is large, a viewer would likely only visit a small section of it. It is more efficient and effective to transmit only the visible region of the VE to the client to reduce startup time and network traffic.

CyberWalk [7], [8] is based on the on-demand transmission approach. It has many advantages over existing systems, including minimal transmission cost through progressive model transmission and the use of viewer/object scopes, minimal rendering cost through multi-resolution object modeling, and high interactivity and system availability through caching and prefetching:

- *Viewer Scope and Object Scope*: To minimize the amount of data needed to be handled, CyberWalk generalizes the Area-Of-Interest concept [13] to both viewers and objects, referred to as the *viewer scope* and the *object scope*. A viewer scope indicates how far the viewer can see. An object scope indicates how far an object can be seen, and its size is proportional to the size of the object. An object can only be seen by a viewer when the two scopes overlap. Hence, objects which scopes do not overlap with the viewer scope do not need to be transferred to the client to save transmission, processing, and memory costs. In CyberWalk, we define a scope as a circular region characterized by a radius. This viewer scope / object scope may be considered as a restricted form of the Aura / Nimbus model [15].
- *Multi-resolution Object Modeling for Progressive Transmission and Rendering:* Sending the complete models of only the visible objects to the client machine on-demand may still cause a possibly long pause in the walkthrough. Instead, CyberWalk encodes each object model in a format similar to the *progressive mesh* [17] for progressive model transmission and reconstruction. Each object is modeled as an ordered list. The list begins with a *base mesh*, which is a minimal resolution model of the object, followed by a sequence of *progressive records*, each of which stores information that helps increase the resolution of the model by a small amount. If we apply the information stored in each of the progressive records to the base mesh of an object in order, the object model will gradually increase in resolution until all the progressive records in the list are exhausted. On the other hand, we may decrease the resolution of the object model by reversing the operation. During run-time, the object distance from the viewer determines the resolution of the progressive mesh needed to be available at the client. Since the visibility of an object usually changes slightly from frame to frame, only a small number of progressive records need to be transmitted to the client between consecutive frames.
- *Multi-resolution Caching:* In CyberWalk, a multi-resolution caching technique was developed to allow fine-granularity of object caching and replacement. A caching mechanism allows a client to utilize its memory and local storage to cache currently visible objects that are likely to be visible in the near future. A local cache also supports a certain degree of disconnected operation when the Internet is temporarily unavailable.
- *Prefetching:* A prefetching mechanism allows a client to predict objects that are likely visible in the near future and fetch them in advance to improve response time. In CyberWalk, an EWMA with residual adjustment scheme was developed to predict the viewer's movement in the VE. Considering the fact that most users use PCs equipped with 2D mice for 3D navigation, we have recently developed a hybrid motion prediction method for predicting the viewer's future movement based on the motion pattern of the 2D mouse [5].

D. Unique Features of CyberWalk to Load Balancing

Although there are similarities in the load balancing process between distributed VEs and traditional distributed systems, there are two major differences here. First, in traditional distributed systems, we may transfer any tasks to any servers [4]. This provides a greater flexibility to the load balancing process. In distributed VEs, however, virtual objects exhibit visual coherency – if an object is visible to the viewer, objects that are geographically close to this object are likely visible to the viewer too. To minimize the communication overheads, we need to cluster nearby objects into the same regions as much as possible so that they can be served by the same servers. Hence, in the load balancing process, an overloaded server may transfer its load to its adjacent servers only.

Second, in traditional distributed systems, the general objective of load balancing is to complete all the tasks within the shortest possible time. Hence, it is important to balance the loadings of all the processors as much as possible. However, in distributed VEs, our objective is to complete all the update operations within a given time (referred to as the performance threshold). Hence, if we can complete all the tasks within the given time, there is no need to perform load balancing. This can minimize the overhead of the load balancing process.

Unlike distributed systems, mobile communication systems only provide very limited degree of load balancing. A mobile system is generally constructed by a collection of cells. Each cell is served by a dedicated base station (i.e., server) located at the center of the cell and is allocated a fixed number of channels. When all the channels of a cell are taken up by the mobile users, it is possible for the cell to borrow additional channels from neighbor cells or from a central controller in order to serve additional users. Once the cell fails to get additional channels, it will have to reject the service request from new users [19]. The major difference between mobile systems and distributed VEs is that in mobile systems, the geographical locations of the base stations are basically fixed and cannot be dynamically adjusted according to the density of the mobile users. On the other hand, in distributed VEs, we may dynamically adjust the size and location of the region served by a server in order to achieve load balancing.

III. Parallel Architecture of CyberWalk

To study the impact of the number of clients on the performance of a centralized server, we have conducted a simulated experiment to measure the average *latency time* and *response time* of multiple viewers on single-server CyberWalk. Latency time measures the average time required to retrieve the base meshes of all visible objects, i.e., from the moment the viewer makes a move to the moment when there is a coarse image of all visible objects. Response time measures the average time that the viewer needs to wait from the moment the viewer makes a move to the moment when all the visible objects are available at the client at their optimal resolutions. Latency time inherently measures the interactivity of the system, while response time measures the absolute performance of the system. The number of clients that we have experimented ranges from 2, 4, 8, 16, 32, 64, and 128.

The size of the VE was set to 100×100 square units (or cells). 6,000 virtual objects are uniformly distributed among the cells. The radius of each object scope is randomly generated, ranging from 0.7% to 1% of the size of the VE, with a mean of 0.85%. Each object is modeled by a progressive mesh, containing 4,500 to 7,000 progressive records with a mean of 5,300. Each progressive record has a size of 40 bytes while each base mesh has a size of 3KB. The database is approximately 1.2GB. The viewer's viewing angle is set to 120 degrees. The radius of the viewer scope is also generated in the same way as that of the object scope. We assume that the network bandwidths of the server and of each client are 20 Mbps and 2Mbps, respectively. The client movement in the VE is determined by sampling the movement of a real user who interacts with the CyberWalk prototype using a mouse. The movement is modeled as follows. The viewer moves circularly. Each movement step includes a translation of 15 units along the viewing direction, followed by a rotation of the viewing direction of 12 degrees. At each step, the viewer rotates his/her head by ±20 degrees. In addition, the moving direction changes with an angle of 10 degrees, after every 4 movement steps. This is termed changing circular moving pattern (CCP).

Both caching and prefetching mechanisms are activated in each client. Since in our previous study [8], we have found that 1% cache size is able to achieve a cache hit of 86%, the size of the storage cache of each client is set to 1% of that of the database, i.e., 12MB. Both the server and the client programs run on PCs with a Pentium III 800 MHz CPU and 512 MB RAM. The result of the experiment is shown in Fig. 1. Figs. 1(a) and 1(b) depict the performance of the server when the number of clients ranges from 2 to 128. As the performance of the server cannot be seen clearly in these diagrams when the number of clients is less than 16, we show it in a much finer scale in Figs. 1(c) and 1(d).

From the figure, both latency and response times of the server as observed by each client increase super-linearly as the number of clients increases. For example, the latency time for 128 clients is as high as 28 seconds. This is very non-favorable for walkthrough as each client will start experiencing low interactivity when the number of clients accessing the VE reaches a certain value. Assuming that a 0.2 second latency time is the performance threshold that a client can tolerate, we observe from Fig. 1(c) that all clients will experience low interactivity when there are more than about 10 clients.

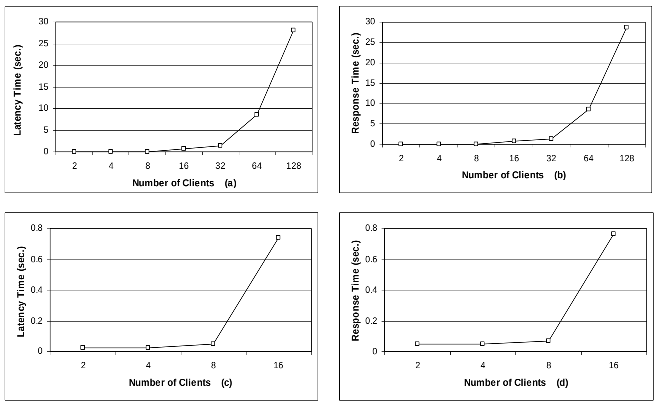

Fig. 1. Effect of number of clients on the performance of the server.

To address this problem, CyberWalk employs a parallel architecture of multiple servers. A parallel architecture is fundamentally a share-nothing architecture [10]. As illustrated in Fig. 2, each server manages its own memory and database repository of virtual objects. In traditional share-nothing architecture, one of the servers is often dedicated to be a coordinator, which has complete knowledge of the data managed in each of the other servers. When the coordinator receives a query from a client, it forwards the query to the server which manages the data required by the query. However, the coordinator could quickly become a bottleneck when the rate of query submission becomes high. Another problem with such architecture, which is regarded as very important in our application, is that any changes made by a client may have to go through many network communication steps before the client receives a reply. This extra delay can seriously affect the interactivity of the system, especially if the servers are not physically located in the same LAN.

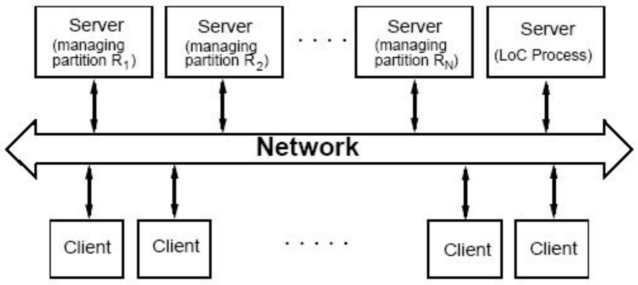
Server (managing partition R_1)
Server (managing partition R_2)
Server (managing partition R_N)
Server (LoC Process)
Network
Client
Client
Client
Client

Fig. 2. Parallel architecture of CyberWalk.

In CyberWalk, we partition the complete VE into a two-dimensional array of regions. Virtual objects within a region will be managed by a single server only. Each server will be serving only those clients who are exploring objects within the region managed by the server. In other words, each client will be communicating directly to the server which manages the virtual objects that the client is exploring. Under normal circumstances, when a viewer is exploring a region, the server managing the region will be responsible for serving all requests from the viewer. However, when the viewer moves to the region boundary, it may be able to see objects belonging to the neighbor regions. Fig. 3 shows such an example. When viewer *V* is crossing the boundary from region R_1 to region R_3, i.e., when the viewer scope of *V*, O_V, touches the boundary of R_3 and R_4, both servers S_3 and S_4, in addition to S_1, will now be serving requests from *V* since *V* is able to view objects within regions R_1, R_3 and R_4.

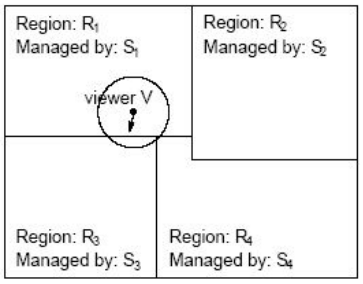

Fig. 3. Visual interaction of a client with multiple servers.

To handle this situation, when O_V first touches the boundary of R_3, server S_1 will send a message to server S_3 of the format $\langle id_V, loc_V, \vec{v}_V \rangle$, where id_V is the ID of *V*, which is inherently the IP address of the client machine, loc_V is the location of *V* in the VE, and $\vec{v}_V$ is the viewing direction of *V*. Once server S_3 has received such a message from server S_1, it determines the objects in R_3 that are visible to *V* and transmits the corresponding progressive records and/or base meshes directly to *V*. *V* will then maintain direct communication channels with both S_1 and S_3 as long as O_V overlaps with both regions. If the viewer scope of *V* now touches R_4, server S_1 will send a similar message to server S_4 and *V* will now be communicating with server S_1, S_3 and S_4. When *V* eventually moves into region R_3, *V* will stop communicating with server S_1 and S_4 and will only communicate with server S_3.

An advantage of this architecture is that each server will effectively be serving for a considerable less number of clients. When the number of clients inside a region has substantially increased to the extent which affects the performance perceived by a viewer, that region may be further partitioned and the newly partitioned region may be allocated to a less busy neighbor server.

As depicted in Fig. 2, one process, named the LoC (Loading Collector) process, is dedicated to collect the loading information of each server. Each server will periodically inform the LoC process about its load, and has the flexibility of determining how often it will inform the LoC process about its load. If a server's loading varies more frequent, the server will inform the LoC more often; otherwise, it could do it less frequent. The LoC process, thus, has the most up-to-date loading information of each server. Approaches to determine such duration has been studied in [6] and is thus beyond the scope of this paper. When a server is overloaded, it contacts the LoC process and requests for the loading information of its neighbor servers. The overloaded server will then select a neighbor server with the lowest load to absorb the newly partitioned region. The LoC process could also be piggybacked as a point of contact for each newly-jointed client. It finds the appropriate server for the user to contact to explore the relevant part of the VE. Note that the LoC process could be hosted in any one of the servers. When the server hosting the LoC process fails or is overloaded, a new server could be selected to host it.

In order to fully realize the advantage of this parallel architecture, we need to address two issues. First, we need a mechanism to monitor the servers' loading. Since the load of each server needs to be updated frequently, this mechanism must be efficient in order not to cause extra load to each server. Second, we need to properly partition the VE into regions. Since the virtual objects may not be uniformly distributed among the regions, if the VE is simply partitioned in a uniform manner, some servers will need to manage more objects than others. This will cause some servers to experience a heavier workload than others. In addition, some regions may contain more interesting objects and attract more visitors. This skewed popularity characteristic of virtual objects may further affect the load distribution among the servers. An efficient partitioning scheme which would consider the loading of the affected servers is needed. We term our partitioning scheme, *adaptive region partitioning*, as the partitioning of regions will be adapted to the load among the servers.

IV. Adaptive Load Balancing

In CyberWalk, the complete VE is regularly subdivided into a large number of rectangular cells. Each cell, $c_{i,j}$, contains a set of objects, i.e., $c_{i,j} = \{o_{c_{i,j},1}, o_{c_{i,j},2} \ldots o_{c_{i,j},|c_{i,j}|}\}$. The VE is also partitioned into a set of N_R regions, i.e., VE $= \{R_1, R_2, \ldots R_{N_R}\}$, while each region contains an integer number of *cells*, i.e., $R_i = \{c_{i,1}, c_{i,2}, \ldots c_{i,|R_i|}\}$. Fig. 4 depicts a partition of 9 regions, each containing 9 cells and managed by one server.

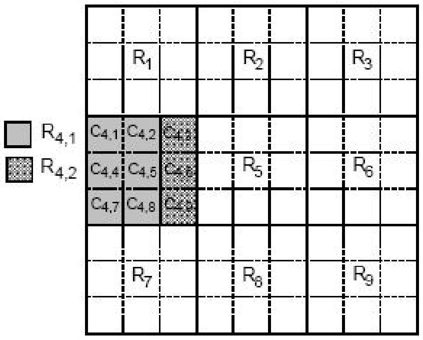

Fig. 4. Partitioning of the VE.

There are several ways to partition the VE into regions. The simplest way is to evenly partition it and we refer to it as the even scheme. Each region will cover the same geographical size of the VE and contain the same number of cells, i.e., $|R_i| = |R_j| \quad \forall R_i, R_j \in \text{VE}$. However, since the virtual objects might not be distributed uniformly within the VE, some regions may contain more objects than others. This could be addressed by partitioning the VE based on "object density" and we refer to it as the density scheme. In this scheme, each region will contain approximately the same number of objects, i.e., $\sum_{k=1}^{|R_i|}\left|c_{i,k}\right| \approx \sum_{l=1}^{|R_j|}\left|c_{j,l}\right| \quad \forall R_i, R_j \in \text{VE}$. However, each region may cover a different geographical size of the VE, and thus may contain different number of cells.

Note that when the virtual objects are distributed uniformly within the VE, the density scheme and the even scheme will result in a similar partition. The density scheme attempts to achieve a uniform load among all servers by ensuring that each server will manage the same number of objects. This is based on an assumption that each object has a similar degree of interest to the viewers and thus, a similar probability of being accessed. In practice, however, viewers may show particular interests in certain objects and explore certain regions more frequent than others. Hence, the density scheme may not necessarily result in a uniform load among the servers. We address this issue by our *adaptive region partitioning* scheme, in which the partitioning of the regions is adapted to the load among the servers.

A. The Adaptive Region Partitioning Scheme

When CyberWalk is first started, there is no information regarding to the loading of each server and the VE is partitioned into N_R regions based on the density scheme. Each server computes its own loading periodically. When server S_i is found to be overloaded, region R_i will be partitioned. The newly partitioned region will then be allocated to a neighbor server of S_i with the lowest load. This inherently directs all future viewers' requests on the newly partitioned region to the neighbor server and thus reduces the load of S_i. For example, in Fig. 4, when server S_4 of region R_4 is overloaded, R_4 is partitioned into two subregions, $R_{4,1}$ and $R_{4,2}$, such that $R_{4,2}$ may be transferred to a neighbor server.

To determine if a region needs to be partitioned, each server S_i will continuously monitor its own load by maintaining two monitor windows called the *short duration window*, $W_S(S_i)$, and *long duration window*, $W_L(S_i)$, with a window size of $|W_S(S_i)|$ and $|W_L(S_i)|$, respectively, where $|W_S(S_i)| < |W_L(S_i)|$. (Hereafter, we will leave out S_i when the context is clear.) The short duration window monitors the load of S_i within a very short duration of time. The purpose is to detect sudden bursts of load on the server either due to network congestion or a sudden increase in interest on region R_i. In order for a server to respond to the sudden bursts of load quickly, $|W_S(S_i)|$ should be set as small as possible. However, the smallest $|W_S(S_i)|$ is limited by the frequency at which the server updates its own loading. By contrast, the long duration window monitors the load of S_i within a much longer period of time. The purpose is to detect a continuous high load on the server. $|W_L(S_i)|$ cannot be set too high or the server will not be too responsive to the change in loading, and it cannot be set to small or it will increase the partitioning overhead. From our experiments, setting $|W_L(S_i)|$ to a few times higher than the value of $|W_S(S_i)|$ produces reasonably good results in most situations.

We model the server's loading by two factors: CPU loading factor (CPU_{server}) and network loading factor (Net_{server}). The CPU loading factor captures the processing overhead required to identify and retrieve the object models requested by the viewers. It is modeled by the server utilization and approximated by the number of objects processed in a second:

$$\text{CPU}_{server} = \frac{\text{number of objects processed per second}}{\text{CPU}_{\max}}$$

where $\text{CPU}_{\max}$ is the maximum number of objects a server can process in a second. This information inherently captures the CPU overhead of the server. The network loading factor captures the transmission overhead required to transmit the progressive records to the viewers and is approximated by the amount of object data sent to the client per second:

$$\text{Net}_{server} = \frac{\text{number of bytes sent per second}}{\text{effective network bandwidth}}$$

where the effective network bandwidth is the actual network bandwidth allocated to the server. Hence, each server will monitor the number of bytes sent through the network and determine the duration to transmit the data.

To determine if a region needs to be further partitioned, each server maintains two sets of partitioning thresholds, $\tau_{s,\text{CPU}}$ and $\tau_{s,\text{Net}}$ for the short duration window and $\tau_{l,\text{CPU}}$ and $\tau_{l,\text{Net}}$ for the long duration window. When CPU_{server} exceeds either $\tau_{s,\text{CPU}}$ or $\tau_{l,\text{CPU}}$, the region will be further partitioned. Likewise, when Net_{server} exceeds either $\tau_{s,\text{Net}}$ or $\tau_{l,\text{Net}}$, the region will also be partitioned. The setting of these two sets of thresholds determines the loading at which the server should activate the partitioning process. Since each server compares its loading with the thresholds for the short duration window frequently (once every $|W_S(S_i)|$), the thresholds essentially specify a maximum loading that the server should not exceed in order to maintain the system's interactivity. On the other hand, each server compares its average loading with the thresholds for the long duration window in a less frequent manner (once every $|W_L(S_i)|$), the thresholds essentially specify an average loading that the server should not exceed in order to maintain the system's interactivity. In other words, these two sets of thresholds specify the bounds on the CPU usage and Network usage of the server. In order to maximize the resource usage of the server, we may set the thresholds for the short duration window as high as possible. The thresholds for the long duration window should be set lower than those of the short duration window. However, they should not be set too low as it would waste the server's resources.

The following steps summarize the process of partitioning a region R_i:

1. When server S_i is overloaded, it contacts the LoC for the loading information of its neighbor servers. For instance, in Fig. 4, neighbor servers of S_4 are S_1, S_5, and S_7.
2. When server S_i receives the loading information of its neighbor servers, it selects the neighbor server with the lowest load as the target server S_T to offload some of its load. This load transfer is achieved by partitioning R_i and allocating the newly partitioned subregion to the target server, i.e., the newly partitioned subregion will be merged

with R_T managed by S_T. Referring to Fig. 4, assuming that server S_4 is overloaded and S_5 is selected as the target server, S_4 will partition R_4 into subregions $R_{4,1}$ and $R_{4,2}$, and transfer $R_{4,2}$ to S_5. Future requests on region $R_{4,2}$ will then be handled by S_5.

3. Server S_i will determine the total amount of load needed to be transferred to neighbor servers. We refer to such goal as the *target load*. In our prototype, each server will reduce its load to 10% below the threshold value.
4. The partitioning is performed on a per cell basis. Each server will maintain a workload indicator for each cell. The partitioning of a region R_i is achieved by de-allocating one or more of its boundary cells, $c_{i,j}$, from R_i and transferring them to the target server, i.e., $R_i = R_i - \{c_{i,j}\}$ and $R_T = R_T + \{c_{i,j}\}$. For example, cells $c_{4,3}$, $c_{4,6}$, and $c_{4,9}$ in Fig. 4 are the boundary cells of R_4 to be merged with R_5.
5. Sometimes, it is possible for the target load to be higher than the amount of load that the target server can accept. To prevent overloading the target server, S_i only transfers a portion of the target load to the target server. It then selects another target server and transfers the remaining load to it.

As discussed in Section 2.4 that virtual objects in the VE exhibit visual coherency, S_i should offload its load to only its neighbor servers instead of any arbitrary servers, to reduce the communication overheads of the system. In Fig. 4, if server S_4 offloads cells $c_{4,3}$, $c_{4,6}$, and $c_{4,9}$ at three different load balancing operations to arbitrary servers, these three cells may potentially be offloaded to three different servers. When the viewer scope of a viewer overlaps with these cells, the client will have to communicate with multiple servers in order to receive object models within these cells. On the other hand, when offloading is performed only to the neighbor servers, all these three cells may well be offloaded to server S_5 and the client will only need to communicate with two servers.

B. *Identifying Target Servers and Target Cells*

When an overloaded server is to transfer part of its load to a neighbor server, it needs to identify some boundary cells as the target cells to be transferred and a neighbor server as the target server to accept the cells. We use Fig. 5 as an example to show how we identify the target server and the target cells. We assume that server S_2 is overloaded and needs to transfer some of its load to a neighbor server.

We define a *break point* as a cell that indicates a potential location for finding boundary cells to be transferred when the server is overloaded. All boundary cells that have at least two of their four sides connected to cells managed by other servers are identified as break points. To minimize the cost of identifying break points during the load balancing process, each server will maintain a *break point list* to indicate all the break points along its boundary. At the same time, the server will also maintain a *boundary edge list*. Each edge is either a horizontal line or a vertical line that joins two aligned break points. For example, in Fig. 5, B_1 and B_2 will form an edge because they are aligned horizontally; however, B_7 and B_1 will not because they are not aligned vertically or horizontally. Hence, region R_2 has four edges (highlighted as thick lines) in its boundary edge list. We further maintain a *neighbor server list* to indicate the neighbor servers that are adjacent to edges found in the boundary edge list. Once we have constructed the neighbor server list, we may select a server with the lightest load from the neighbor server list as the target server of S_2.

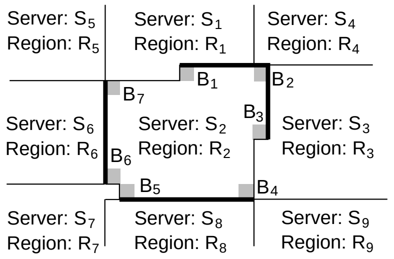

Fig. 5. Break points and boundary edges of a region.

In Fig. 5, if we assume that server S_3 is selected as the target server, the boundary cells between B_2 to B_3 will be selected as the target cells to be transferred to S_3, until the target load is reached. After transferring the boundary cells between B_2 and B_3 to S_3, break points B_2 and B_3 will be removed from the break point list. The cell to the left of B_2 will become a new break point to be inserted to the break point list. A new boundary edge will then be generated which is formed by B_4 and the new break point. Hence, we need to remove the edge between B_2 to B_3 from the boundary edge list and add the new edge to the list. If the total load of all the boundary cells between B_2 and B_3 is lower than the target load, we will select the cells along the new edge as the target cells to be transferred to S_3, until the target load is reached. However, if S_3 cannot accept the complete target load, S_2 will transfer the remaining load to S_1, if S_1 is the next lightest loaded server in the neighbor server list.

The basic idea of the adaptive region partitioning scheme is that the overloaded server always looks for a lightly loaded neighbor server and "peels" its own boundary cells to that server. Note that we do not include edges that are not found in the boundary edge list for cell transfer, because if we do so, it will likely increase the perimeters of the regions. For example, transferring the boundary cells between B_2 to B_3 will likely decrease the perimeters of regions S_2 and S_3, while transferring the boundary cells between B_3 to B_4 will likely increase the perimeters of the regions and hence increase the chance that a viewer needs to be served by more than one server.

V. Results and Discussions

We have developed a simulation model and conducted experimental studies to quantify and evaluate the performance of the parallel architecture of CyberWalk. The parameters used in our experiments are listed in Table I. Our experimental settings are similar to those described in Section 3. In our simulation model, the VE is regularly subdivided into 100×100 cells. The complete VE is partitioned into N_R regions, with $N_R << 100 \times 100$. Since each region is managed by one server, N_R servers will be required.

There are *n* virtual objects in the VE distributed among the cells. The modeling of each virtual object has been described in Section 3. There are N_V clients accessing the VE. Both the server and the client programs run on a Pentium III PC with a 800 MHz CPU and 512 MB RAM. The network settings are essentially the same as described earlier. We model two object distribution patterns, D_O, in our experiments: uniform and skew. In uniform distribution, the objects are distributed uniformly among all regions. In skew distribution, each region contains a random percentage of objects, ranging from 30% to 200% of the mean value. We study three different region partitioning schemes, P_R: even, density, and adaptive. In even and density, each region will not be further partitioned into subregions regardless of the workload of the server. In adaptive, a region might be further partitioned into subregions using the adaptive partitioning scheme.

TABLE I
PARAMETER VALUES FOR THE EXPERIMENTS

Para.	Description	Exp. #1	Exp. #2	Exp. #3	Exp. #4
n	number of objects	6K	1K, 2K, 4K, 6K	6K	6K
N_V	number of clients	128	16, 32, 64. 128	16, 32, 64, 128	128
N_R	number of regions	1, 3, 6, 9	9	9	9
P_R	region partitioning scheme	even	even	even, density	density adaptive
D_O	object distribution	uniform	uniform	skew	skew

To monitor the load, the window size of the short duration window of each server is set to 1 second, as we update the server loading once a second. The window size of the long duration window of each server is set to 5 seconds in our experiments. We assume the effective bandwidth ratio to be 0.8. Since the maximum network bandwidth of each server is 20Mbps, the effective network bandwidth is thus 16Mbps. In addition, we set $\text{CPU}_{\max}$ to 1.4×10^6, which is the maximum number of objects that the server can process in a second. This number is obtained through experiments.

To determine at what loading to initiate the partitioning process, we set both CPU and Network partitioning thresholds for the short duration window $\tau_{s,\text{CPU}}$ and $\tau_{s,\text{Net}}$, respectively, to 1. As mentioned in Section 4.1, these two thresholds determine the maximum loading of the server. We also set the two thresholds for the long duration window $\tau_{l,\text{CPU}}$ and $\tau_{l,\text{Net}}$ to 0.9, except for experiment #5, where we attempt to study how the setting of the two thresholds may affect the performance of the system.

Each viewer will be residing at a random position within the VE when the simulation starts, and move within the VE according to the changing circular pattern (CCP) described in section 3. Our simulation program ensures that the path of each viewer is different. We measure *latency time* and *response time*, as defined in section 3, experienced by each client. In each experiment, each metric is determined by averaging the metrics obtained from all movement steps of all clients. The standard deviations are found to be small. Here, we present six of the experiments to quantify the performance of the parallel architecture of CyberWalk.

A. *Experiment #1*

In the first experiment, we study the effect of number of servers on the performance of CyberWalk. In this experiment, n is fixed at 6,000 objects. We depict the overhead when there are 128 clients. The region partitioning scheme is fixed at even. (The effect of various region partitioning schemes are studied in Experiments #3 and #4.) The object distribution pattern is fixed at uniform. (The effect of various object distribution patterns will be studied in Experiment #3.) The number of servers, N_R, ranges from 1, 3, 6, and 9. The measurements of the two metrics are depicted in Figs. 6(a) and 6(b). Since the values of the metrics for 6 and 9 servers are too small to be noticeable, we zoom in the scale for these two data points in Figs. 6(c) and 6(d). We observe from the diagrams that both latency time and response time decrease exponentially with the number of servers. Using the 0.2 second performance threshold for acceptable visual perception, we can see from the figure that 9 (or more than 6) servers are sufficient to serve for a population of 128 clients.

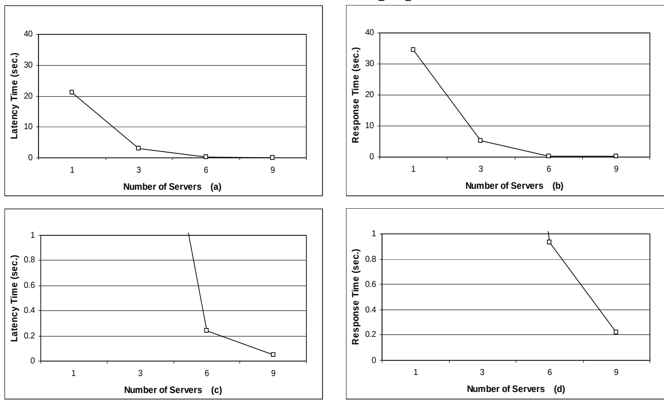

Fig. 6. Effect of number of servers on system performance.

B. *Experiment #2*

In the second experiment, we study the effect of number of objects on the performance of CyberWalk. In this experiment, n ranges from 1,000 to 6,000 objects. The number of clients ranges from 16 to 128. The number of servers is fixed at 9. The region partitioning scheme is still fixed at even, and the object distribution pattern is fixed at uniform. Fig. 7 depicts the measurements of the two metrics. The first row shows the metrics versus the number of objects, while the second row shows the metrics versus the number of clients. We can see that both latency time and response time increase with the number of objects. It is very encouraging that all the latency times fall below the 0.2 second performance threshold, meaning that users is able to experience a good visual perception on the system even with a lot of clients and objects.

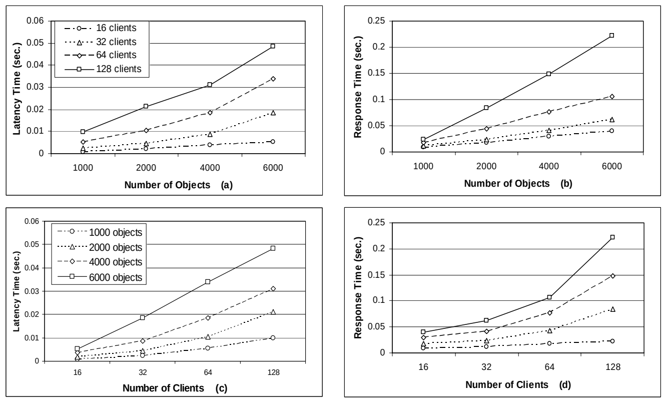

Fig. 7. Effect of number of objects/clients on system performance.

When comparing Figs. 7(b) and 7(d), we also observe that the number of objects does not have as big effect on the performance as the number of clients. This is because the amount of data, or details, transmitted depends on the distance of an object from the viewer. In a typical walkthrough, only a few objects are really rendered at high detail at any time. Increasing the number of objects will only moderately increase the amount of data needed to be sent. In contrast, increasing the number of clients will contribute to a greater amount of data transmitted from the server since each client will need to request for its own copy of all the visible object models.

C. *Experiment #3*

In the third experiment, we compare the performance between even and density region partitioning schemes. In this experiment, *n* is fixed at 6,000 objects. The number of clients ranges from 16 to 128. The number of servers is again fixed at 9. Since two partitioning schemes differ only when the objects are not distributed uniformly across the VE, we will only look at the skew object distribution pattern here.

In Fig. 8, we show three sets of data in each graph, depicting the lowest, the average, and the highest overheads among the 9 servers. We observe that in even partitioning, there is a big difference in performance among the servers. This indicates that some servers might be overloaded, while others might be under utilized. This is because in even partitioning, the VE is divided into equal-sized regions and hence each server will need to handle different number of objects, resulting in a non-uniform workload among the servers. We also observe that when there are 128 clients, the latency time experienced by the clients will be higher than the 0.2 second performance threshold. In contrast, in density partitioning, the difference in overhead among the servers is minimal. This indicates that density partitioning is able to evenly distribute the workload among the servers. In addition, all clients will experience a good visual perception as the latency times of all clients are under the 0.2 second performance threshold.

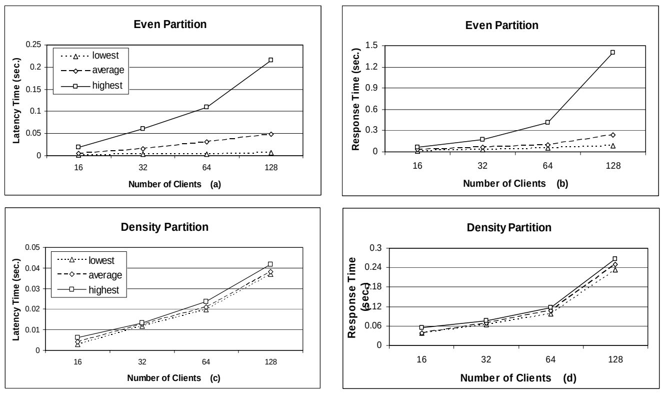

Fig. 8. Performance of even/density region partitioning schemes.

D. *Experiment #4*

In Experiment #3, we have indicated that by partitioning the regions based on object density, all servers will have relatively uniform load and thus, similar performance. This is based on the assumption that each viewer has a similar degree of interest in each object. Therefore, if all regions have a similar number of objects, they will also have a similar access probability. However, in practice, viewers usually have different degrees of interest in different objects. Hence, a region may have a higher access probability if the viewers have a high interest in the objects within it, disregarding of the number of objects that the region contains. Consequently, a uniform object density among various regions does not necessarily result in a uniform workload among the servers.

In the fourth experiment, we study the effect of our adaptive region partitioning scheme in balancing the load among the servers. We fix the number of clients to 128 and the number of objects to 6,000. We only look at the skew object distribution pattern in this experiment, as the even object distribution pattern has a similar behavior. For comparison purpose, we show results of both density and adaptive partitioning schemes since the density scheme is shown to be able to achieve certain degree of uniform load among the servers in Experiment #3. To model different degrees of interest in objects, the center region is dedicated to be the hottest region in which *V*% of the viewers will visit.

To collect the loading information, each server notifies the LoC process of its load once a second in our experiment. We measure the performance of the hottest region by varying *V*% from 10% to 80%. Fig. 9 shows the performance of the center server, i.e., the hottest server, and the average performance of its affected neighbor servers. We can see that both latency and response times increase with the number of viewers. With the density partitioning scheme, the center server experiences a much higher overhead than the neighbor servers. We can see from the figure that the neighbor servers are under utilized, while the center server is overloaded. Using the 0.2 second performance threshold, we observe that the clients will start experiencing a poor visual perception when more than 40% of the viewers are visiting the center region. By contrast, with the adaptive partitioning scheme, the overhead experienced by the center server is similar to that experienced by the neighbor servers. This indicates that the center server could offload its workload to its neighbor servers properly. Hence, each server is able to support a much larger percentage of viewers. From Fig. 9(a), we can see that even with 80% of the viewers visiting the center region, the latency time of the center server (the second curve from the top) is only just above the 0.2 second performance threshold with our proposed method.

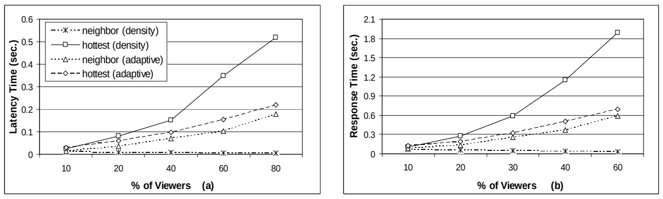

Fig. 9. Performance of density/adaptive region partitioning scheme.

E. Experiment #5

In this experiment, we attempt to study how the setting of the partitioning thresholds would affect the performance of the system. All the other settings of this experiment are the same as those in experiment #4, except that here, we vary the percentage of viewers visiting the hottest region from 10% to 100%. We would like to see how the system would behavior when the loading of the hottest server reaches its thresholds. Since the thresholds for the short duration window specify the maximum loading of the server, we keep them at 1 throughout the experiment. We vary the thresholds for the long duration window to see how they affect the load distribution.

Fig. 10 shows results of this experiment. The two numbers associated with each curve indicate the thresholds for the short/long duration windows. At low viewer percentages, we can see from Fig. 10(a) that the setting of the thresholds has very little effect on the load distribution. The difference between the loading of the hottest server and the average loading of the affected neighbor servers is roughly the same for all threshold settings. However, as the percentage of viewers visiting the hottest region increases, the difference in loading decreases when the thresholds are set lower (e.g., 0.5). This is because at lower threshold values, the hottest server will become overloaded at lower loading and activate the partitioning process more often. As the number of viewers increases, all curves will approach to their maximum latency times. We can see that with low threshold setting, difference in loading between the hottest server and the affected neighbor servers becomes small. This is because at higher loading, the hottest server will initiate the partitioning process more frequently to keep its own load to within the threshold. Concurrently, some of the affected neighbor servers may become overloaded themselves and transfer some of their loads to their own neighbor servers, which in effect increases the number of affected neighbor servers to share the load. We may note that there is a wider gap between the loading of the hottest server and the average loading of the affected server when the thresholds are set at 0.9. This indicates that the loading of the servers are not fully saturated yet.

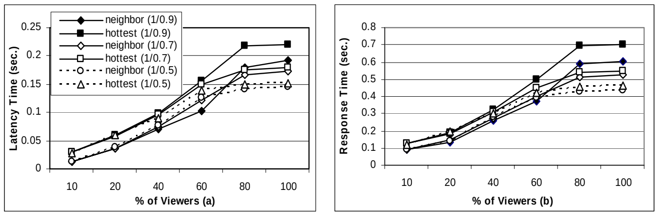

Fig. 10. Effect of setting the partitioning thresholds on system performance.

Judging from the results, setting lower threshold values for the long duration window allows the system to complete the task well before the 0.2 second performance threshold. However, since the servers would now initiate the partitioning process at a lower threshold, it can only serve fewer clients. On the other hand, by setting higher threshold values for the long duration window, the server will initiate the partitioning process at higher loading. This will increase the number of clients that the system can serve but at the same time increase the latency time. From Fig. 10(a), the hottest server has exceeded the 0.2 second performance threshold when the thresholds of the long duration window are set at 0.9.

F. Experiment #6

Fig. 11 demonstrates how the shape of the regions changes as we introduce viewers into the VE. Fig. 11(a) shows the initial partitioning of the VE, based on object density, into 9 regions managed by 9 servers. (Grey dots represent objects.) Hence, regions with high object density have smaller size. Fig. 11(b) shows the new partitioning after we have introduced a large number of clients (shown as black dots) and let them walk around for a while. We can see that regions with a lot of clients are significantly smaller in size now. We can also see that the size of some regions becomes larger even though they also have some clients inside, for example, the left middle region. This is because they have taken up some of the loading from their neighbor regions. As they are not overloaded yet, they do not distribute any of their own loads to other regions. Overall, our adaptive region partitioning scheme can effectively distribute the load among all the servers while minimizing the communication costs.

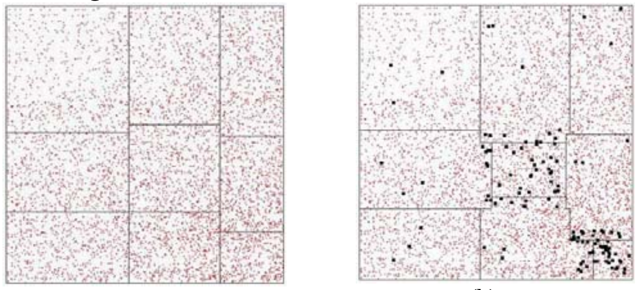
(a) (b)

Fig. 11. The partitioning of the VE when there is a large number of clients.

G. Evaluations: Local Approach vs. Global Approach

Although the global approach to load balancing can achieve a more accurate load balancing in general, there are two reasons for us to adopt the local approach here. First, the local approach, in particular our proposed method, in general has a much lower computation overhead as it only needs to process loading information of the neighbor servers. This is important as the load balancing process runs on the overloaded processor. Second, unlike the global approach that generally aims at achieving even load distribution among the processor, in our application, we only need to solve the overloading problem as it occurs. This can reduce overall network consumption and computation overhead. As long as the system can complete all the tasks within the performance threshold, the user will be able to enjoy a satisfactory performance even if some servers may have a high loading than the others.

Although it is difficult to compare the complexity of the local approach with the global approach under all possible situations, a general comparison can be found in [26], which describes a detailed comparison among various load balancing methods in terms of information dependency, load balancing overheads and aging of information. The conclusion is that diffusive methods in general perform better than other methods including the global methods. Since our method is also a diffusive method, which propagates the excessive load out from the overloaded server, our method also performs better than the global methods in general. Further, due to visual coherency of virtual objects, global methods cannot simply pass any cells from an overloaded server to any lightly loaded server. The load still has to propagate from the overloaded server through all the intermediate servers to the lightly loaded server. Hence, the performance of the global methods is expected to be even worst when applied here.

To explain the above point, we may consider a situation where all the neighbor servers of an overloaded server are nearly overloaded. With our method, the overloaded server, S_o, will first indicate to a neighbor server, S_{n1}, that S_o will need to transfer certain amount of its load to S_{n1}. If S_{n1} determines that it will be overloaded after receiving the load, it will initiate a partitioning process to transfer some of its load to its own neighbor server, S_{n2}, before accepting the load from S_o. With a global method, even though if we can identify that S_{n2} is the nearest lightly loaded server of S_o, we cannot simply pass the boundary cells of S_o to S_{n2}. We need to pass the boundary cells of S_o to S_{n1}, which in turn passes its own boundary cells to S_{n2}.

VI. Conclusion

In this paper, we have described a parallel architecture to support virtual walkthrough and illustrated its implementation in our CyberWalk prototype. We have pointed out that in traditional client/server architecture, a single server has limitation in serving multiple clients. As one alternative to improving the performance, we propose a parallel architecture and employ multiple servers in serving the clients. We have discussed several ways of partitioning the virtual environment into multiple regions and studied their behaviors under various object distribution patterns. We have also introduced the adaptive region partitioning scheme to address the problem of non-uniform access among the regions. This scheme, in effect, partitions the virtual environment based on the viewers' degrees of interest on the regions.

We have studied our parallel architecture via simulation and experiments. Our studies show that the adaptive region partitioning scheme is very effective in maintaining a uniform load among the servers. Although we have not conducted experiments on other moving patterns, we believe the CCP moving pattern studied in this paper is able to present an overall performance behavior of our approach.

References

[1] Asheron's Call, Available at http://www.microsoft.com/games/zone/asheronscall/.

[2] J. Calvin, A. Dicken, B. Gaines et al., "The SIMNET Virtual World Architecture," *Proc. IEEE VRAIS*, pp. 450–455, 1993.

[3] C. Carlsson and O. Hagsand, "DIVE - a Multi-User Virtual Reality System," *Proc. IEEE VRAIS*, pp. 394–400, 1993.

[4] T. Casavant and J. Kuhl, "A Taxonomy of Scheduling in General-purpose Distributed Computing Systems," *IEEE Trans. on Software Engineering*, **14**(2):141-154, Feb. 1988.

[5] A. Chan, R.W.H. Lau, and B. Ng, "Motion Prediction for Caching and Prefetching in Distributed Virtual Environments," To appear in *ACM Trans. on Internet Technology*.

[6] B. Chan, A. Si, and H.V. Leong, "A Framework for Cache Management for Mobile Databases: Design and Evaluation," *Journal of Distributed and Parallel Databases*, **10**(1):23–57, 2001.

[7] J. Chim, M. Green, R.W.H. Lau, H.V. Leong, and A. Si, "On Caching and Prefetching of Virtual Objects in Distributed Virtual Environments," *Proc. of ACM Multimedia*, pp. 171-180, Sept. 1998.

[8] J. Chim, R.W.H. Lau, H.V. Leong, and A. Si, "CyberWalk: A Web-based Distributed Virtual Walkthrough Environment," *IEEE Trans. on Multimedia*, **5**(4):503-515, Dec. 2003.

[9] T. Das, G. Singh et al., "NetEffect: A Network Architecture for Large-scale Multi-user Virtual World," *Proc. ACM VRST*, pp. 157–163, 1997.

[10] D. DeWitt, S. Ghandeharizadeh, and D. Schneider, "A Performance Evaluation of the Gamma Database Machine," *Proc. ACM SIGMOD*, 1988.

[11] Diablo II, Starcraft, Available at http://www.blizzard.com/.

[12] EverQuest, Available at http://everquest.station.sony.com/.

[13] J. Falby, M. Zyda, D. Pratt, and R. Mackey, "NPSNET: Hierarchical Data Structures for Real-Time Three-Dimensional Visual Simulation," *Computers & Graphics*, **17**(1):65–69, 1993.

[14] T. Funkhouser, "RING: A Client-Server System for Multi-User Virtual Environments," *Proc. Symp. on Interactive 3D Graphics*, 1995.

[15] C. Greenhalgh and S. Benford, "MASSIVE: A Distributed Virtual Reality System Incorporating Spatial Trading," *Proc. Int'l Conf. on Distributed Computing Systems*, pp. 27-34, 1995.

[16] C. Greenhalgh, J. Purbrick, and D. Snowdon, "Inside MASSIVE-3: Flexible Support for Data Consistency and World Structuring," *Proc. Int'l Conf. on Collaborative Virtual Environments*, pp. 119–127, 2000.

[17] H. Hoppe, "Progressive Meshes," *Proc. ACM SIGGRAPH*, 1996.

[18] M. Hori, T. Iseri, K. Fujikawa et al., "Scalability Issues of Dynamic Space Management for Multiple-Server Networked Virtual Environments," *Proc. IEEE Pacific Rim Conf. on Communications, Computers and Signal Processing*, pp. 200–203, 2001.

[19] I. Katzela and M. Naghshineh, "Channel Assignment Schemes for Cellular Mobile Telecommunication Systems: a Comprehensive Survey," IEEE Personal Communications, **3**(3):10-31, June 1996.

[20] R. Lea, Y. Honda, K. Matsuda, and S. Matsuda, "Community Place: Architecture and Performance," *Proc. VRML'97*, pp. 4l–50, Feb. 1997.

[21] I. Pandzic, T. Capin, E. Lee, N. Thalmann, and D. Thalmann, "A Flexible Architecture for Virtual Humans in Networked Collaborative Virtual Environments," *Proc. Eurographics*, pp. 177–188, 1997.

[22] Quake, Available at http://www.idsoftware.com/.

[23] G. Singh, L. Serra, W. Png, and H. Ng, "BrickNet: A Software Toolkit for Network-Based Virtual Worlds," *Presence*, **3**(1):19–34, 1994.

[24] G. Singh, L. Serra, W. Png, A. Wong, and H. Ng, "BrickNet: Sharing Object Behaviors on the Net," *Proc. IEEE VRAIS*, pp. 19–27, 1995.

[25] Ultima Online, Available at http://www.uo.com/.

[26] M. Willebeek-LeMair and A. Reeves, "Strategies for Dynamic Load Balancing on Highly Parallel Computers," *IEEE Trans. on Parallel and Distributed Systems*, **4**(9):979-993, Sept. 1993.

Beatrice Ng received a B.Sc. degree in Computer Studies from the City University of Hong Kong in 2000. She is currently a research student in the Department of Computer Science, City University of Hong Kong. Her research interests include computer graphics and distributed virtual environments.

Rynson W.H. Lau received a (top) first class honors degree in Computer Systems Engineering in 1988 from the University of Kent at Canterbury, England, and a Ph.D. degree in Computer Graphics in 1992 from the University of Cambridge, England. He is currently an associate professor at the City University of Hong Kong. Prior to joining the university in 1998, he taught at the Hong Kong Polytechnic University. From 1992 to 1993, he worked at the University of York, England, on a defense project on image processing. Rynson Lau's research interests include computer graphics, virtual reality and multimedia systems.

Antonio Si received his Ph.D. from the University of Southern California, and is currently with Oracle Corporation. Before joining Oracle, he worked as an assistant professor at the Hong Kong Polytechnic University and as a software engineer at Sun Microsystems Inc.. He has served on the program committees for several international conferences and as external reviewers for a number of international conferences and journals. His research interests are in mobile computing, internet computing, and digital libraries. He is a member of the ACM and IEEE Computer Society.

Frederick W.B. Li received a B.A. (Honors) in Computing Studies in 1994 and a MPhil. in Computer Graphics in 1998, both from the Hong Kong Polytechnic University, and a Ph.D. degree in Computer Graphics in 2001 from the City University of Hong Kong. He is currently an assistant professor at the Hong Kong Polytechnic University. He was the research manger of an R&D project funded by the Hong Kong Government Innovation and Technology Commission, PCCW, and Sun Microsystems from 2001 to 2003. Frederick Li's research interests include surface modeling, virtual reality, computer animation and computer networking.