Voronoi State Management for Peer-to-Peer Massively Multiplayer Online Games

Shun-Yun Hu, Shao-Chen Chang, Jehn-Ruey Jiang
Department of Computer Science and Information Engineering
National Central University, Taiwan, R.O.C.

***Abstract*—State management is a basic requirement for multi-user virtual environments (VEs) such as *Massively Multiplayer Online Games* (MMOGs). Current MMOGs rely on centralized server-clusters that possess inherent scalability bottlenecks and are expensive to adopt and deploy. In this paper, we propose *Voronoi State Management* (VSM) to maintain object states for peer-to-peer-based virtual worlds. By dynamically partitioning the VE with *Voronoi diagrams* and aggregating game states of overloaded nodes onto superpeers, VSM supports existing consistency control to enable scalable, load balanced, and fault tolerant VE state management. As both client and server-side resources are utilized collaboratively, VSM also integrates both client-server and peer-to-peer VE designs in a unified approach.**

I. Introduction

Massively Multiplayer Online Games (MMOGs), where up to hundreds of thousands of players assume virtual identities known as *avatars* to interact in synthetic *virtual environments* (VEs) [1], have in recent years seen phenomenal commercial success and cultural impacts. Today's MMOGs scale by using a number of dedicated servers linked on high-speed networks to form a *server-cluster*. However, server-clusters cost millions of dollars to develop, deploy, and maintain [2], and the total amount of resources is still limited at any given moment. Some recent proposals thus suggest the use of *peer-to-peer* (P2P) architectures to support MMOGs that may be more scalable and affordable [3]–[14].

One fundamental requirement for MMOGs is the maintenance of *game states*, which are the various attributes of game objects such as the locations, possessions, and current status of avatars and computer-controlled *non-player characters* (NPCs). In a client-server architecture, game states are stored authoritatively at the servers, and are updated according to game-specific rules called *game logic*. For a P2P-based MMOG (P2P MMOG), the design goal thus is to distribute game states from centralized servers to participating clients, while ensuring consistent views and balanced workloads for all nodes. However, although proposals have been made to maintain a P2P network's *topology* based on avatar positions [3]–[9], few have yet addressed game states management comprehensively [10]–[14]. Furthermore, P2P schemes pose other practical challenges, namely: *heterogeneity*, where not all clients have the same resources to support similar behaviors; *churn*, where clients constantly join and leave the network; and *hacking*, where a client's behavior is modified to cheat or disrupt gameplay.

We propose a P2P-based game state management scheme called *Voronoi State Management* (VSM) that considers both client heterogeneity and churn. VSM partitions the VE into a number of cells via *Voronoi diagrams* [15], each of which is managed by a client using existing consistency controls. Capable clients are promoted as *aggregators* to manage multiple cells for overloaded nodes. To balance aggregators' loads, VSM dynamically adjusts their control spheres and inserts new ones as needed. Sufficient fault tolerance is achieved by replicating game states onto backup nodes.

The contributions of this paper are the analysis of current MMOG state management and the design of VSM. We show that VSM can be practical to meet the consistency, responsiveness, scalability, and reliability requirements for MMOGs [4], and can be easily integrated with server-clusters, thus providing a bridging transition from client-server to P2P MMOGs. The rest of this paper is organized as follows. Section II provides background on MMOGs and server-based state management. Section III presents our problem formulation. We describe VSM's design in Section IV, and related work in Section V. Conclusions are presented in Section VI.

II. Background

A. Networking and Consistency Models

Games can be seen as finite state machines where user inputs or game semantics (such as NPC behavior) cause *events* to be generated and game states subsequently modified according to game logic. For example, a rule may state that: "A player gains 30 experience and 1 agility points if the avatar has run for 3 minutes". State updates therefore can be understood as a *request - process - update - display* sequence.

Networked games additionally can be understood from their *networking model* (i.e. how nodes connect and communicate) and *consistency model* (i.e. how game state updates are maintained across nodes). For networking, the two main architectures are *point-to-point* (also known as peer-to-peer, for clarity, we will refer it as point-to-point) and *client-server* (including both single servers and server-clusters). In point-to-point, all nodes are fully connected to each other and any message generated is sent to all other nodes. Point-to-point does not scale well as transmissions grow at $O(n^2)$, where n is the node size. In client-server, event messages from all *client nodes* are sent to a special *server node*, which then redistributes the messages according to the clients' individual needs, achieving $O(n)$ overall transmissions [1].

Two main consistency models also exist in today's networked games based on where game logic is executed: *event-based* and *update-based*. Event-based requires all nodes to have the full game states and perform the same game logic [16]. Any event occurred is received and processed by all nodes. As long as each node has the same states and processes events in roughly the same order (depending on consistency requirements), game states would update consistently on all nodes. As consistency depends on event ordering, synchronization both *conservative* (e.g. *lock-step*) and *optimistic* (e.g. *Time Wrap* [17], *TSS* [18], and *OOS* [19]) have been proposed. On the other hand, in update-based model, a server node retains all the game states and is the only node that executes game logic. Clients send events only to the server, and receive *state updates* relevant to their interests [20]. Consistency is achieved as long as client-side *replicas* of the game objects are roughly in sync with the server's *primaries* by having relevant and timely updates. Here, both the server and clients maintain game states, with the difference that the server's version may be global (i.e. it has all the states) and authoritative, while the clients' states are local (i.e. it has only what is relevant to current gameplay) and referential (i.e. game states could be corrected if deviated from the server's version [20]).

Event-based models often coincide with point-to-point networking, whereas update-based models are often used with client-server. Event-based is suitable for games such as *real-time strategy* (RTS), where the full set of game states is needed by each node, yet too large to update [16]. But as events from all other nodes must first be received before time can advance, time advancement could slow down when extra latency exists. Update-based is suitable if clients only need a subset of the game states, or if game logic is preferred to be centrally executed. As the server has authoritative view for a given area, logical clock can also advance at regular intervals (e.g. 100ms for each clock tick) *regardless* of the actual latency among nodes, providing faster responses. Current MMOGs thus adopt client-server with update-based consistency control, as clients only need partial game states, security is more easily guaranteed, and the server need not wait to advance its time.

B. MMOG Server-cluster Designs

One common way the game industry adopts to scale MMOGs is to provide players parallel access to duplicated worlds (called *shards*), where each world is essentially a separate environment with a limit on concurrent users (e.g. 2000 to 2500 [2]). Players then choose which shard to enter upon login. However, this approach lessens realism and limits social interactions as communications are blocked across shards [21].

To scale a single virtual world, three main designs for server-cluster exist based on how game states are distributed: 1) *replication-based*, where the servers form a point-to-point topology and possess identical, fully replicated game states (e.g. *proxy-servers* [22] and *mirror-servers* [18]); 2) *object-based* [23]–[25], where game objects are distributed evenly among servers; and 3) *zone-based* [21], [26]–[32], where game objects are partitioned spatially.

Replication schemes allow events from any player to be processed by any server, so that players can connect to the server with minimal latency. They also allow more flexible load balancing, as overloaded servers can migrate players to any server in the cluster. However, its point-to-point communication can be unscalable. Object-based approaches often attempt to split objects as evenly as possible on the servers (the most common are player objects). This allows load balancing to be simply finding ways to distribute objects evenly. However, as events may affect an unpredictable number of objects on other servers, the amount of inter-server communication can become unpredictable. A zoned approach, on the other hand, keeps most of the event processing local unless the events occur near zone borders, inter-server communication thus can be constant for a given player density, achieving better scalability. However, cross-border interactions may involve locks for zones or objects that could be time-consuming [27].

Given the better scalability of zone-based approaches, today they are more widely adopted in practice. However, three inter-related issues must be addressed: 1) how to partition the world, 2) how to balance work load among servers, as users may crowd and overload a particular zone, and 3) how to maintain consistent visibility and interactions across zone borders.

Existing partitioning schemes may be *static* such as grids [21], [27]–[29], or *dynamic* such as strips [30], quad-tree [26], or other irregular shapes [31], [32]. To balance load, load detections are first done by periodically monitoring CPU or bandwidth usage [28]. Once abnormality is found, evaluation is performed to determine if zones should be repartitioned, or if objects need to be migrated to other servers. *Global schemes* recalculate load assignments based on the loads from all servers [23]–[26], whereas *local schemes* consider boundary shifting or load migrations only with neighbors [28]–[32]. Local schemes are more efficient in terms of computation and migration costs, but migration could not occur when neighboring servers are also overloaded. On the other hand, global schemes can better utilize resources as loads can be migrated to any server. However, global information collection and optimization are time-consuming and may not be practical under MMOGs' real-time constrains. Additionally, if a server handles discontinuous zones as a result of load migration, inter-server communication could increase [28]. Neither approach thus addresses load balancing satisfactorily.

To provide visibility across zone borders, *replicas* may be created for near-border objects. To ensure update atomicity, border objects may be locked for events that update more than one object across borders [27] (e.g. when a player hands an item to another player, see also "Seamless Servers: The Case For and Against" in [33]). Consistent and transactional updates thus are possible at the cost of increased delays.

In general, server-cluster load balancing faces the tradeoff between balancing computation load and minimizing inter-server communications [24], [28]. Besides load balancing, consistency maintenance during load migration is another issue that needs to be considered [27], [34], while little published work has been done on fault tolerance [11].

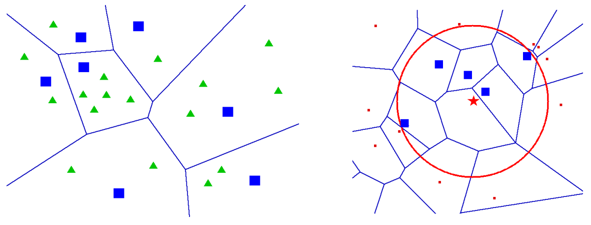
Fig. 1. (L) Voronoi partitioning (triangles are objects, squares are user nodes as *sites*) (R) Star node needs to connect to its *AOI neighbors* (squares)

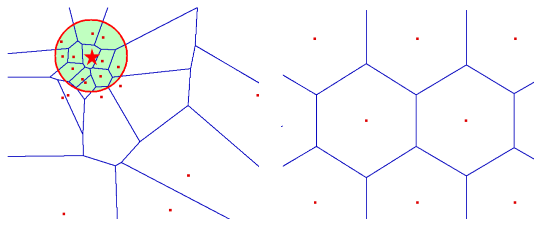
Fig. 2. (L) Shades are the aggregator's *sphere of control* (R) Virtual peers

III. Problem Formulation

We first present a general problem formulation and assumptions for state management in VE applications as follows:

1. The world is a 2D plane with fixed width and height.

2. *Attributes* are tuples of the form *(type, name, value)*, where a *type* is a basic data-type such as **int, char, float**, or **string**. They are the basic encapsulations of *game states*.

3. *Objects* consist of tuples of the form *(name, attributes, x, y)*, where x and y are the x and y coordinates of the object's location within the VE, and *attributes* is a list of attributes associated with the object. Objects are the basic units representing players, NPCs, or items and may change locations via player inputs or game logic executions.

4. Objects are created, updated, and destroyed by *events*, which are messages initiated by players or NPC algorithms, and are processed according to game-specific *game logic*.

5. Each player controls an *avatar object*, which has a fixed and game-specific *area of interest* (AOI) radius [1], within which interactions occur (i.e. an avatar is only aware of object updates in its AOI; likewise, events can only impact objects within the AOI of the initiating avatar).

IV. Voronoi State Management

As existing server-based schemes may not effectively support system scalability and load balancing, while P2P schemes have yet to manage game states sufficiently, we thus seek to address both issues with *Voronoi State Management* (VSM).

A. Design of VSM

Our aim is to allow object states be managed collaboratively by both servers and clients in a scalable and seamless manner. We also would like to utilize existing client-server-based consistency control to make state management transparent for application developers, so that applications can be developed in manners similar to existing client-server-based VEs.

We note that the central issue is to effectively partition and distribute game states among participating nodes, such that resource usage at all nodes is bounded. A simple way thus is to distribute game objects among *all* user nodes, where each object is managed by the nearest user (i.e. the node whose position is closest to the object on the virtual map). Note that the VE will then partition into a number of *Voronoi cells* [15] with user positions as the *sites* of each cell (Fig. 1L). Within a cell, the user node can govern all object updates in a client-server fashion. As some events may affect objects in nearby cells, each node also needs to connect with neighboring nodes in charge of the other objects (i.e. the *AOI neighbors* whose Voronoi cells are covered by a given AOI, see Fig. 1R). *Voronoi-based Overlay Network* (VON) [4] can be used to keep connections with AOI neighbors. However, this approach has three potential problems: 1) When users are crowded inside a small area, each node's connection would grow at $O(n^2)$, where n is the number of AOI neighbors. 2) Due to the unequal distribution of user locations, certain cells may be quite large with large number of objects beyond what a user node can handle. 3) The users are always moving, making object ownerships difficult to determine precisely.

To address the above issues, we first identify two roles for each user node: *arbitrator* and *peer*. An *arbitrator* is in charge of a given Voronoi cell, and has the authority to decide how game states should change according to event messages and game logic. A *peer* is a node that generates events and displays the results of processed events. Arbitrators are like the server, and the peers are clients. User nodes thus always have the *peer* roles, which take user commands and generate relevant events for arbitrators to process. Arbitrators, however, can be fulfilled only by capable user nodes, or by the server if needed.

All user nodes start as both peers and arbitrators where each peer submits events to its arbitrator part for processing. However, when user nodes crowd and become overloaded due to increased connections and messages, a more capable user (or server) node is selected as the only arbitrator responsible for a group of peers. This elevated superpeer, or *aggregator*, then acts as a special arbitrator with a fixed location and a circular *sphere of control*, of which any regular peer must submit its arbitrator role upon entering (Fig. 2L). In case a peer approaches multiple aggregators, it would join the closest one. This way, peers in the sphere of control connect only to the aggregator, reducing overall bandwidth usage. Meanwhile, other independent peers outside the sphere can still keep contacts with their AOI neighbors via the aggregators. To prevent an arbitrator from managing excessive area, server-provisioned *virtual peers* are deployed initially at specific locations as arbitrators (Fig. 2R). Control is released to user nodes only incrementally as they join. To avoid ambiguities in object ownerships caused by node movements, *explicit ownership transfer* via messages is required between arbitrators.

There is still a *gateway* server that acts as the initial object maintainer and the entry point for all user nodes. However, the gateway is light-weight and is required by clients only during the initial join or when finding suitable aggregators.

We describe VSM's mechanisms in more details below:

Consistency control We choose to use *update-based* consistency for VSM, as consistency can be achieved without requiring the arbitrators to wait to progress logical time. We also observe that few events in typical MMOGs are actually *correlated* [19] (e.g. a trade can occur amidst a fight, and most interactions occur between just two players), consistency thus is achievable if only one set of authoritative game states exists.

In VSM, each arbitrator is authoritative as the *managing arbitrator* within its cell. Events generated in the cell (e.g. player/NPC actions) are first sent to the managing arbitrator, then processed directly or forwarded to relevant arbitrators if some affected objects are outside the cell. An arbitrator thus is in contact with neighboring arbitrators whose objects are observable by its peers. Each arbitrator keeps internal time buckets [35] to store incoming events and processes them at regular intervals according to *game logic*. When object states have changed, other interested arbitrators whose peers can see the object will receive *state updates* from the managing arbitrator. Events and updates can be either *reliable* or *unreliable* based on their importance (e.g. movements can be unreliable, but chat or trade messages are reliable). We note that games commonly utilize *client-side predications* [20], [35] to mask latency for fast actions (e.g. movements), so even if we send the more critical events (e.g. chat, trade, player death) reliably with longer delays, it may still be acceptable.

Game objects's ownerships in principle belong to the managing arbitrator. However, as Voronoi cell boundaries can change due to user node movements, to ensure that state updates occur authoritatively at only one node, explicit messaging between arbitrators is required for ownership transfers. In most cases the transfers are light-weight, as game states are usually already known by the neighboring arbitrator (if their peers can see the objects). Whole object states are transferred along with ownership only if the new owner does not know the object. When an arbitrator mistakenly receives an event for objects it does not own (e.g. ownership transfer is in process), the arbitrator would forward the event to the proper owner. Two main types of actions in MMOGs are handled as follows:

1) Basic update: When a user walks, equips some items, or attacks other users/NPCs, the event affects one or more objects directly. In such case the event is sent to and processed by the managing arbitrator of the affected objects (via forwarding if necessary). Updates are then sent to any interested arbitrators.

2) Transactional update: When a trade occurs between two users (e.g. A buys an item from B), the event will affect states for two or more objects *simultaneously*. In this case, A's arbitrator first requests to *lock* the affected attributes from B's arbitrator, before updating A's states (i.e. item added, money decreased). B's arbitrator is then notified to update B's states while releasing the lock (i.e. item removed, money increased). The transaction finishes when A's arbitrator receives a confirmation from B's arbitrator. During the lock, any events that might change A or B's affected attributes are queued for later processing. Unless the final confirmation is received, both arbitrators will restore the original states after a timeout.

Load balancing The goal for load balancing is to ensure that the loads of all nodes in the system do not exceed each node's capacity. Unlike existing schemes, where the load is first distributed to a fixed number of high-capacity server nodes, then migrated between them when overload occurs, we propose a somewhat opposite approach where the load is first assigned onto many low-capacity user nodes, and then aggregated to high-capacity nodes or server nodes when overload happens.

We assume that there are existing methods to detect a node's current load, and that thresholds for a node's *overload* and *underload* conditions are well-defined. In VSM, initially each user node is also the arbitrator of the Voronoi cell based on its own position. Overload occurs when there are too many AOI neighbors to connect and exchange messages, and the arbitrator would ask the *gateway* server for an *aggregator* (i.e. a superpeer with spare capacity, or a server node if no such peer exists). The aggregator then takes over the arbitrator role of the overloaded node, with a fixed location at the requestor's current position. The aggregator has a predefined radius according to its capability, within which all user nodes should submit their arbitrator roles. The aggregator then manages all the Voronoi cells of the peers within its sphere of control. It is important to note that the takeover does not change the partitioning of the cells, so the area controlled by an aggregator is often irregular but roughly conforms to a circle.

However, the aggregator itself also has a resource limit and may be overloaded if it manages too many nodes. When an aggregator overloads, it shrinks the radius of its sphere of control until the load becomes manageable. However, if there are still other overloaded user nodes outside the sphere, they would ask the gateway to initiate new aggregators. When the aggregator is underloaded (i.e. the peers it manages has decreased), it would disintegrate itself to return control for the Voronoi cells back to the user nodes.

Fault tolerance As VSM relies on user-supplied resources, which is less reliable than provisioned server resources, fault tolerance in face of node or link failure thus is important. We discuss the failure of regular arbitrators and aggregators separately as follows:

1) Regular arbitrators. Each arbitrator selects a random user node as its *backup arbitrator* to store a copy of its game states. If the user node fails, the backup arbitrator would transfer the ownerships of game objects to all the *enclosing arbitrators* of the failed node, so that state updates can continue. Of course, when a user node fails, its avatar object is no longer online and can be excluded from the transfer.

2) Aggregators. Each aggregator also has a capable user node as its *backup aggregator*, which acts similarly as a backup arbitrator and stores copies of the original aggregator's game objects. When an aggregator fails, the backup assumes the original role of the failed aggregator (i.e. re-establishing connections with neighboring nodes to handle events) and finds a new backup aggregator with the help of the gateway server. This design aims to assist a smooth transition back to normal operations when aggregators fail.

B. Discussions

VSM supports existing update-based consistency control via the use of arbitrators and can utilize existing methods for state synchronization between peers and arbitrators. It is responsive as most *request - process - update - display* sequences are within three end-to-end hops (i.e. peer → managing arbitrator → neighboring arbitrator(s) → peers in nearby cells). VSM can scale as client resources can be added, while aggregation in crowded areas helps to balance loads. Fault tolerance is supported by state replications on backup nodes, which may immediately transfer out ownerships or assume managing responsibilities for aggregators. Finally, as virtual peers and aggregators can be provided by servers, VSM allows both server and client resources to integrate in the same framework, thus provides a transition from client-server to P2P MMOGs.

V. Related Work

Few work exists on state management for P2P VEs. *SimMud* [10] supports basic state management via superpeers within fixed-size regions. VSM has gone further to allow dynamic region partitioning. *Colyseus* [12] supports first person shooter (FPS) games based on DHTs, but the $log(n)$ query in DHT may create unacceptable object discovery latency for large node size. By using VON [4], VSM discovers objects within bounded query hops. *Time Prisoner* [13] has basic state management in superpeer-managed zones, but does not consider inter-zone interactions or load balancing. *HYMS* [11] and *Hydra* [14] consider fault tolerance issues, but still rely heavily on server resources. Naor and Wieder [36] propose Voronoi partitioning for DHT overlays, but only fixed node locations are considered. Chen and Lee first suggest the use of Voronoi to partition VEs [37], but details are not given. Ohnishi et al. [8] describe a *Delaunay* (dual of Voronoi diagrams) overlay for VEs, but without considering state management. Our concept of aggregators is similar in spirit to the node clustering scheme recently proposed by Varvello et al. [9] for Delaunay overlays. However, they do not deal with superpeer overloads when node density increases, while other state management aspects such as consistency control are also not considered.

VI. Conclusion

We have presented Voronoi State Management (VSM), a P2P state management scheme that aims to utilize both client and server resources in a seamless manner. We are now evaluating VSM via simulations with regard to bandwidth usage and the actual effectiveness of the load balancing and fault tolerance mechanisms. VSM provides some interesting possibilities for additional design improvements, for example: if aggregators move with their peers, connection handoff may become fewer, or if arbitrators handle only dynamic avatar objects, then ownership transfers for the more static objects can be avoided. We believe that Voronoi partitioning and dynamic superpeer elevations are promising approaches to handle the difficult load balancing and fault tolerance issues in VE state management, and as such, they provide a unique space for various policy designs in the future.

References

[1] S. Singhal and M. Zyda, *Networked Virtual Environments*, 1999.

[2] D. Kushner, "Engineering everquest," *IEEE Spectrum*, 2005.

[3] J. Keller and G. Simon, "Solipsis: A massively multi-participant virtual world," in *PDPTA*, 2003.

[4] S.-Y. Hu, J.-F. Chen, and T.-H. Chen, "Von: A scalable peer-to-peer network for virtual environments," *IEEE Network*, vol. 20, no. 4, 2006.

[5] C. GauthierDickey *et al.*, "Using n-trees for scalable event ordering in peer-to-peer games," in *Proc. NOSSDAV*, June 2005, pp. 87–92.

[6] P. Morillo *et al.*, "Providing full awareness to distributed virtual environments based on peer-to-peer architectures," *LNCS*, vol. 4035, 2006.

[7] S. Douglas *et al.*, "Enabling massively multi-player online gaming applications on a p2p architecture," in *Proc. ICIAD*, December 2005.

[8] M. Ohnishi *et al.*, "Incremental construction of delaunay overlaid network for virtual collaborative space," in *Proc. C5*, 2005, pp. 77–84.

[9] M. Varvello, E. Biersack, and C. Diot, "Dynamic clustering in delaunay-based p2p networked virtual environments," in *Proc. NetGames*, 2007.

[10] B. Knutsson *et al.*, "Peer-to-peer support for massively multiplayer games," in *INFOCOM*, 2004, pp. 96–107.

[11] K. Kim, I. Yeom, and J. Lee, "Hyms: A hybrid mmog server architecture," *IEICE Trans. Info. and Sys.*, vol. E87-D, no. 12, 2004.

[12] A. Bharambe *et al.*, "Colyseus: A distributed architecture for multiplayer games," in *NSDI*, 2006.

[13] A. E. Rhalibi and M. Merabti, "Agents-based modeling for a peer-to-peer mmog architecture," *ACM CIE*, vol. 3, no. 2, pp. 1–19, April 2005.

[14] L. Chan *et al.*, "Hydra - a massively-multiplayer peer-to-peer architecture for the game developer," in *Proc. NetGames*, 2007.

[15] F. Aurenhammer, "Voronoi diagrams-a survey of a fundamental geometric data structure," *ACM CSUR*, vol. 23, no. 3, pp. 345–405, 1991.

[16] P. Bettner and M. Terrano, "1500 archers on a 28.8: Network programming in age of empires and beyond," Proc. GDC, 2001.

[17] R. M. Fujimoto, "Parallel discrete event simulation," *CACM*, vol. 33, no. 10, pp. 30–53, October 1990.

[18] E. Cronin *et al.*, "An efficient synchronization mechanism for mirrored game architectures," *MT & A*, vol. 23, no. 1, pp. 7–30, May 2004.

[19] S. Ferretti and M. Roccetti, "Fast delivery of game events with an optimistic synchronization mechanism in mmog," in *Proc. ACE*, 2005.

[20] T. Sweeney, "Unreal networking architecture," http://unreal.epicgames.com/network.htm, 1999.

[21] P. Rosedale and C. Ondrejka, "Enabling player-created online worlds with grid computing and streaming," Gamasutra Resource Guide, 2003.

[22] J. Muller and S. Gorlatch, "Rokkatan: scaling an rts game design to the massively multiplayer realm," *ACM CIE*, vol. 4, no. 3, 2006.

[23] F. Lu *et al.*, "Load balancing for massively multiplayer online games," in *Proc. Netgames*, 2006.

[24] J. C. S. Lui and M. F. Chan, "An efficient partitioning algorithm for distributed virtual environment systems," *TPDS*, vol. 13, no. 3, 2002.

[25] P. Morillo *et al.*, "An adaptive load balancing technique for distributed virtual environment systems," in *Proc. ICPDCS*, 2003, pp. 256–261.

[26] J.-Y. Huang and M.-Y. Tsai, "The study of dynamic scene management for massive-players virtual environment," in *Proc. CVGIP*, August 2005.

[27] M. Assiotis and V. Tzanov, "A distributed architecture for mmorpg," in *Proc. Netgames*, 2006.

[28] J. Chen *et al.*, "Locality aware dynamic load management for massively multiplayer games," in *Proc. PPoPP*, 2005, pp. 289–300.

[29] B. Ng *et al.*, "Multi-server support for large scale distributed virtual environments," *IEEE TMM*, vol. 7, no. 6, pp. 1054–1065, 2005.

[30] E. Deelman *et al.*, "Dynamic load balancing in parallel discrete event simulation for spatially explicit problems," in *Proc. PADS*, 1998.

[31] R. Chertov and S. Fahmy, "Optimistic load balancing in a distributed virtual environment," in *Proc. NOSSDAV*, 2006.

[32] S. Pekkola *et al.*, "Collaborative virtual environments in the year of the dragon," in *CVE*, 2000, pp. 11–18.

[33] T. Alexander, *Massively Multiplayer Game Development*. Charles River Media, 2003.

[34] J.-Y. Huang *et al.*, "Design of the server cluster to support avatar migration," in *Proc. VR*, 2003.

[35] C. Diot and L. Gautier, "A distributed architecture for multiplayer interactive applications on the internet," *IEEE Network*, 1999.

[36] M. Naor and U. Wieder, "Novel architectures for p2p applications: the continuous-discrete approach," in *Proc. ACM SPAA*, 2003, pp. 50–59.

[37] C.-C. Chen and C.-J. Lee, "A dynamic load balancing model for the multi-server online game systems," in *HPC Asia, Poster*, 2004.