Voronoi State Management for P2P MMOGs

Shun-Yun Hu, Shao-Chen Chang, Jehn-Ruey Jiang
Department of Computer Science and Information Engineering
National Central University, Taiwan, R.O.C.
syhu@yahoo.com, cscxcs@gmail.com, jrjiang@csie.ncu.edu.tw

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 concept paper, we propose *Voronoi State Management* (VSM) to maintain object states for peer-to-peer-based virtual worlds. By dynamically partitioning the virtual world with *Voronoi diagrams* and performing localized replication for game states, 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.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed applications*

General Terms

Algorithms

Keywords

peer-to-peer, virtual environment, online games, state management, Voronoi, scalability, load balancing, fault tolerance

1. INTRODUCTION

Massively Multiplayer Online Games (MMOGs), where up to hundreds of thousands of players assume virtual identities known as *avatars* to interact in computer-generated *virtual environments* (VEs) [31], have in recent years seen phenomenal commercial success and cultural impacts. The scalability of today's MMOGs is maintained by using a number of dedicated servers linked together on high-speed networks to form a *server-cluster*. However, server-cluster architectures cost millions of dollars to develop, deploy, and maintain [19], and the total amount of resources is still limited at any given moment. Some recent proposals [5, 14, 17, 18] thus suggest the use of *peer-to-peer* (P2P) architectures to support MMOGs that may be more scalable and affordable.

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 server-cluster 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 maintenance from centralized servers to participating clients, while ensuring consistent views of the world and balanced workloads for all nodes. However, although proposals have been made to maintain the *topology* of a P2P network based on avatars' positions, few has yet addressed comprehensively the management of game states. 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 regions via *Voronoi diagrams* [3], and promotes capable clients as *arbitrators* to handle state management in a given region. To balance arbitrators' loads, VSM dynamically adjust region boundaries and insert new arbitrators as needed. By replicating game states in nearby regions, sufficient fault tolerance and efficient consistency control can be achieved, which are important given the dynamics of P2P networks and the real-time requirements of MMOGs.

The contributions of this paper are the analysis of current MMOG state management schemes, and the proposal of the VSM algorithm. We show that VSM can be a practical approach fulfilling the consistency, responsiveness, scalability, and reliability requirements for MMOG state management, and can be easily integrated with server-cluster architectures, thus providing a bridging transition from client-server to P2P-based MMOGs.

The rest of this paper is organized as follows. Section 2 provides background on MMOGs and server-cluster state management. Section 3 presents a model for the state management problem. We describe VSM's design in Section 4, and conclude the paper with some discussions in Section 5.

2. BACKGROUND

2.1 Characteristics of MMOGs

MMOG is a relatively new genre of computer games that is characterized by a large number of concurrent users within the same virtual world [1, 18]. Game-play in MMOGs usually surround a fantasy setting, where players assume identities of diverse races and occupations, and may enhance their avatars' equipments and skills by carrying out quests. Computer-controlled NPCs often populate a MMOG to act as virtual citizens that provide information, trades, or fighting opportunities. Similar to other large-scale VEs [31, 32], MMOGs have the following set of requirements:

Consistency A basic requirement for multiplayer games is to allow players at different physical locations to interact within a shared space. To allow meaningful interactions, a player's action must be seen by other nearby observers in consistent manners. Consistency between players' individual views of the world is achieved through the passing and processing of *event messages*, which update each clients of the current states of the system. However, packet loss, jitter and latency all may degrade the consistency of games.

Responsiveness Games are real-time applications where players expect to see the effects of their own actions and that of other players within a reasonable amount of time. However, network latency and processing delay may prevent a game to be responsive. Latency tolerance varies for different game genres [32], and is typically between 500ms to 1000ms for MMOGs [12].

Security MMOGs retain user accounts for authentication and billing purposes, such data thus must be securely stored. Additionally, as the entertainment value rests on the fair and correct execution of game rules, cheating or malicious gameplay must be prevented, detected, and stopped. As client programs may be hacked to produce illegal behaviors, most MMOGs retain the execution of game logic exclusively at the servers, and use the clients only as terminals for displaying the results of server-side game logic executions.

Scalability In a MMOG context, scalability refers to the ability of the system to sustain a large number of concurrent users within the same world. A system's scalability usually depends on factors such as the server's total bandwidth and processing capacity, the amount of activities occurring, and player densities in a region. However, the prevailing factor for scalability is whether resource usage is bounded at each system component for both clients and servers [14].

Persistency One key characteristic of MMOGs is to allow access to the virtual world 24 hours a day. Players may log in and out of the server at any time, while retaining their inventory and status (e.g. levels, experience points, and currency). Persistency is usually provided by transactional databases that keep continuous records of all important game state updates.

Reliability As a MMOG scales and persists, it must also be fault tolerant to accidental software or hardware failures, as well as live updates of software or contents, in order to provide smooth and continuous play experiences. In case of a server crash, normal operations must be resumed quickly without noticeable disturbance. Roll-back recovery of the game states from database may also be required sometimes.

In this paper we will consider all the above requirements except for security and persistency, which are separately addressed in other work [6].

2.2 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 points and 1 point in agility if the avatar has run for 3 minutes". State updates therefore can be understood as a *trigger - process - 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 traditionally known as peer-to-peer, for clarity, we will refer it as point-to-point in this paper) and *client-server* (which includes both single servers and server-clusters). In point-to-point, all nodes are fully-connected to each other where messages generated by any node are sent to all other nodes. Point-to-point does not scale well as overall transmissions grow at $O(n^2)$. In client-server, event messages from all *client nodes* are sent to a special *server node*, which would then redistribute the messages according to the clients' individual needs, making overall transmission grow at $O(n)$ [31].

Two main consistency models also exist in today's networked games based on where game logic is executed: *event-based* and *update-based* models. Event-based model requires that all nodes have the full set of game states and perform the same game logic [4]. When an event occurs, it is received and processed by all nodes. As long as each node has the same set of states and that events are processed in more or less the same order (depending on consistency requirements), game states would update consistently on all nodes. As executing events in the same order by all nodes is crucial to guarantee consistency, synchronization schemes have been proposed that include both *conservative* ones such as *lock-step*, or *optimistic* ones such as *Time Wrap* and *trailing state synchronization* (TSS) [10, 13].

On the other hand, in update-based model, a server node retains the entire set of game states and is the only node that performs game logic execution. Clients send events to the server instead of to all other clients, and receive *state updates* relevant to their current interests [33]. Consistency is achieved as long as client-side *replicas* of the game objects are more or less in sync with the server's *primaries* by receiving relevant updates in a timely manner. In this model, 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. only what is visible to current gameplay is maintained) and referential (i.e. game states could be corrected if deviated from the server's version).

Event-based models often coincide with point-to-point networking, whereas update-based models are often used with client-server. Event-based models are more responsive as messages need not be relayed, and suitable for games where the entire set of game states is needed by each node, yet too large to update (as in real-time strategy, or RTS games, where thousands of states change constantly) [4]. Update-based is suitable if clients only need a subset of game states to operate, or if game logic is preferred to be executed by a single authority. Current MMOGs adopt client-server with update-based consistency control, as clients only need partial game states and security is more easily guaranteed.

2.3 MMOG Server-cluster Designs

One common way used by the game industry to increase the number of players of a MMOG is to provide players parallel access to duplicated worlds (called *shards*), where each world is essentially a separate environment with a limit to the number of concurrent users (e.g. between 2000 and 2500 [19]). Players then choose which shard to enter upon login. However, this approach lessens realism and limits social interactions as players cannot communicate across shards [29].

To scale a single virtual world, three main types of server-cluster exist based on how game states are distributed: 1) *replication-based*, where the servers themselves form a point-to-point topology and game states are fully replicated among servers (e.g. *proxy-servers* [23, 24] and *mirror-servers* [10]); 2) *object-based* [20, 21, 22], where game objects are distributed evenly among servers; and 3) *zone-based* [9, 11, 16, 29], where game objects are assigned via spatial partitioning.

Replication-based schemes have the advantage that events from be any player can be processed by any server, so players can connect to the server with minimal latency. They also allow more flexible load balancing, as overloaded servers can migrate players to any other server in the cluster. However, the point-to-point communication makes it unscalable. Object-based approaches usually attempt to split objects as evenly as possible on the servers (the most common are objects representing the players). This allows load balancing to be simply finding ways to distribute objects evenly. However, as any event may affect an unpredictable number of objects, inter-server communication can become unpredictable. A zoned approach, on the other hand, keep 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 [2].

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 within a particular zone and overload the server, and 3) how to maintain visibility and interactions across zone borders in consistent manners.

Existing partitioning schemes may be *static* such as grids [2, 8, 26, 29], or *dynamic* such as strips [11], quad-tree [16], or other irregular shapes [9, 28]. For balancing the load, load detections are first done by periodically monitoring CPU or bandwidth usage [8]. Once abnormality is detected, load reassignment 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 [20, 21, 22], whereas *local schemes* consider boundary shifting or load migrations only with neighbors [8, 26]. Local schemes are more efficient in terms of computation and migration costs, however, load migration would 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 available 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 [8]. 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 objects across borders [2] (e.g. when a player hands an item to another player, see also "Seamless Servers: The Case For and Against" in [1]). 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 [8, 21]. Besides load balancing, consistency maintenance during load migration is another issue that needs to be considered [2, 15], while little published work has been done on fault tolerance.

3. PROBLEM FORMULATION

In this section, we present a general problem formulation for state management in VE applications. We use the following assumptions about a VE system:

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 [31], 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 avatar that creates them).

Given the above, we define below the requirements for a P2P MMOG, based on the requirements for MMOGs. Note that persistency and security are beyond our scope.

Consistency Consistency of object states is the most basic requirement for VE applications. In a P2P setting where peers may manage different regions, consistency and visibility across regions should be guaranteed. To achieve consistency, all peers responsible to execute game logic and update object states, should do so based on consistent knowledge of the game states and event ordering.

Responsiveness As prolonged latency could render a game unplayable to users [32], the maximum number of end-to-end transmission hops should be limited, such that there is only a bounded number of message transfers between peers in any *trigger - process - display* sequence.

Scalability When game logic processing load is distributed over peers, the system scales well if the resources usage is always bounded within capacity. We thus pose two requirements: 1) the resources for processing game logic should increase with player size, and 2) the processing and transmission load for each peer should be balanced and bounded.

Reliability As node availability in P2P networks is less stable than that of a server-cluster, one important requirement is that the entire system may still function despite of frequent node joining and leaving. Fault tolerance against node failures thus is essential.

4. VORONOI STATE MANAGEMENT

As existing server-cluster approaches may not effectively deal with system scalability and load balancing, while P2P

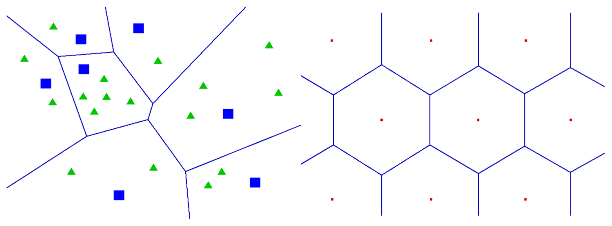
Figure 1: (left) VSM (right) virtual peers

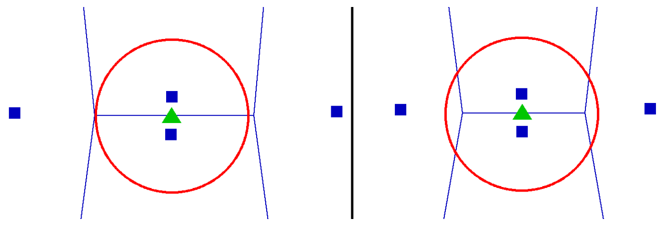
Figure 2: (left) no violation (right) violation

approaches have yet to offer practical ways to manage game states, we thus seek to address both issues by proposing *Voronoi State Management* (VSM). In this section, we first describe VSM's main components: Voronoi partitioning, localized replication, and transactional topology modification. Procedures and policies of VSM are then presented.

4.1 Design of VSM

Voronoi partitioning Our aim is to allow object states be managed collaboratively by both the server and the clients in a seamless way. We begin with the observation that the nodes in existing P2P networks are highly heterogenous [30], which suggest that designating more capable nodes as *supernodes* may be more practical than assuming equal capacities. The basic idea of our design is to partition the VE into a number of small *regions*, and promote a certain number of capable clients as the *arbitrators* for each region. An arbitrator manages a region by performing tasks similarly to a server in client-server architectures. Each client machine thus may assume two roles: as a regular *peer*, or as an *arbitrator*, with independent functionalities. Ideally, game developers only need to consider client-server-like interactions between game objects, while consistency, load balancing, and fault tolerance issues are taken care of by VSM.

Arbitrators assume fixed locations within the VE, and serve as the *sites* of a *Voronoi diagram* [3]. Voronoi diagram partitions a given 2D plane with n sites (i.e. a coordinate point) into n regions. Each site is associated with a region such that the region contains all the points closest the region's site than to any other site. We require that each arbitrator be responsible to maintain the authoritative versions of objects within its region (i.e. as object *owners*, whose ownerships are transferred by explicit messages. See Fig. 1, where squares represent *arbitrators* and triangles represent *game objects*). Localized load balancing is performed by 1) adjusting region boundaries via site movements to produce region shapes that may accommodate player clustering or 2) inserting new arbitrators within overloaded regions. Object states are stored authoritatively at the *managing arbitrator*, which is the arbitrator whose region contains the positions of objects. At any moment, a peer connects only with its managing arbitrator to learn of object states within its AOI.

To provide initial arbitrators before any clients have joined the system, a *gateway* server first sets up *virtual peers* (Fig. 1) that act as arbitrators positioned at regularly-spaced locations. As clients enter the VE, the roles of arbitrators may then be taken up by clients when virtual peers are overloaded and require new arbitrators be added. The existence of virtual peers also helps to reduce region size to be more manageable, so that game states may migrate to newly promoted arbitrators in an incremental manner.

Localized replication To support visibility across regions for clients and fault tolerance, object states are fully replicated on all the *enclosing arbitrators*, which are neighboring arbitrators whose regions have shared edges with a managing arbitrator. If an arbitrator fails, its enclosing arbitrators could immediately take over the state management tasks for the failed arbitrator upon detection. Besides the support of fault tolerance, another important function replication serves is to support existing event-based consistency models. By fully replicating game states on enclosing arbitrators, any arbitrator plus its enclosing arbitrators thus resemble a point-to-point network on a local scale. This allows any existing event-based consistency control be used by VSM (e.g. lock-step, Time Wrap [13], or TSS [10]), as long as the game states that may be affected by an event are available on all the arbitrators handling the event. Efficient consistency control can thus be supported without the need to use any time-consuming cross-region locks that could degrade the responsiveness of a game. This also allows applications be developed in a client-server style, without having to concern the actual object distribution across regions.

Transactional topology modification The above consistency control requires that an event can never influence game objects beyond those managed by the enclosing arbitrators, otherwise an event may be processed by an arbitrator that does not have the complete set of game states necessary to produce consistent updates. In other words, a peer's AOI should never cross beyond the regions of its enclosing arbitrators. To avoid such scenarios, no edge of a given region can be less than the diameter of a peer's AOI (Fig. 2). This requires that all the edges in the Voronoi diagram to be minimally longer than the AOI diameter of a peer (e.g. a *minimal edge-length Voronoi diagram*). To ensure that all edges are longer than a certain length, any additions, updates, and removals of Voronoi sites (i.e. the arbitrator's positions) must be *transactional*, in the sense that they should not occur unless all the enclosing arbitrators around a proposed change have verified and agreed. To prevent violations due to arbitrator failures, one of the enclosing arbitrators of a failed arbitrator would assume managing responsibility immediately upon detection, so that the overlay topology may be intact. A new arbitrator should then be requested from the gateway server to replace the failed one. Note that topology changes are usually not frequent and occur only as needed for load balancing purposes.

We choose *Voronoi-based Overlay Network* (VON) [14] as the P2P overlay for arbitrators, as VON provides simple functions to allow an arbitrator to join or move within a P2P overlay, and discover enclosing arbitrators efficiently with one-hop queries.

4.2 Procedures

We now describe in details the main procedures of VSM, including peer join, event update, peer movement, load balance, and fault tolerance (node leave).

Peer Join (JOIN) When a peer first joins the VE, it contacts the *gateway server* for a unique ID and reports its capability to the gateway, so that future promotion as arbitrator is possible. The gateway then forwards a join request to the first managing arbitrator via greedy forwards. The gateway itself controls at least one arbitrator (via virtual peer), so it knows some enclosing arbitrators for the forwarding. Once a join request is received, the managing arbitrator sends the peer a list of its enclosing arbitrators and initial object states within the peer's AOI.

Event Update (UPDATE) Currently a simple lock-step protocol [13] is used to process events: a peer first sends an *event* message to its managing arbitrator, which then timestamps and forwards the event to its own event queue and those of its enclosing arbitrators whose regions overlap with the peer's AOI. This ensures that the managing arbitrators of any objects affected by the event can receive the event properly. As the enclosing arbitrators already have complete object replicas within the peer's AOI, they could update the states consistently given the same event execution order. To guarantee consistent event ordering, arbitrators only process events with timestamps before the minimal time among the latest timestamped events received from their enclosing arbitrators. To prevent stalling when an arbitrator has no events to send, *tick events* are sent as heartbeats to notify enclosing arbitrators to progress the time.

Once an event is executed, the managing arbitrator (i.e. owner) of any affected objects forward updated states to 1) all interested peers whose AOI cover the object, and 2) its own enclosing arbitrators, so that any enclosing arbitrators that did not process relevant events may still keep their object replicas up-to-date. Every time an object changes states due to an event, its *version number* would increment. Arbitrators thus accept updates from a neighbor only if the version is more current than its own.

Peer Movement (MOVE) Peers may enter new regions as they move. To ensure proper region-switching, each peer maintains a list of the enclosing arbitrators of its current managing arbitrator, which is learned every time when entering a new region. Whenever a peer receive new position updates from its managing arbitrator (as a result of *movement events*), it checks whether the new position is now in a new region. If so, it connects to the new managing arbitrator and disconnects the previous one.

Load Balance (BALANCE) One difficult issue with any partitioning scheme is the balance of workload within each region, as workload may change as players move or the activity levels within the region vary. To address this issue, we adopt two strategies: 1) reshaping the region, and 2) adding new arbitrators to the overloaded region. When a given arbitrator is overloaded (according to certain criteria), help requests are sent to its enclosing arbitrators, which would move closer towards the overloaded node if they have spare capacities. Changing arbitrator coordinates towards the overloaded node reduces the overloaded arbitrator's region. Transfer of object states from the overloaded node to its neighbors could then start. If the first approach does not relieve the load after some time, the gateway is asked to insert an additional arbitrator. The gateway picks and promotes the best candidate from a list of potential peers as the new arbitrator. Inserting a new arbitrator to an overloaded region could then relive state management responsibility from the overloaded node. In case no suitable peers are found, the server may insert a virtual peer. In a similar spirit, if an arbitrator is underloaded, it first notifies enclosing arbitrators to move away. If the underload persists, it would depart from the network as an arbitrator. Object ownerships are transferred to enclosing arbitrators before the departure. When a newly elected arbitrator joins the VE, it receives both the relevant states and pending events from its enclosing arbitrators before becoming operational. Game states are transferred reliably before events, to ensure that event processing will be based on the full set of game states. As arbitrators process only "safe" events in our consistency model, the new arbitrator is not able to process any events until all pending events (and therefore states) have been received from its enclosing arbitrators, thus ensuring the correctness and continuity of event processing.

As changes to the Voronoi diagram should not violate the minimal edge-length requirement, arbitrator insertions or movements must be checked and agreed by all enclosing arbitrators. To ensure that object ownerships are correct, an arbitrator constantly checks if the objects it owns is in another region, and if so, it transfers ownerships accordingly.

Fault Tolerance (LEAVE) When a peer leaves the VE, it simply does so, as its managing arbitrator can detect the departure via time-out, remove its avatar object, and notify other peers. For arbitrator failure, one of its enclosing arbitrators (e.g. the closest based on positions) immediately assumes object ownerships after detection as a *transient arbitrator*, and calls the gateway for a *replacement arbitrator*. The replacement joins the network similarly as outlined in the BALANCE procedure, and assumes ownerships once all object states and events are received. Meanwhile, peers would connect first to the transient arbitrator, and then the replacement arbitrator, to send events and receive updates.

4.3 Policies

Certain operations in VSM are open to various strategies and are best governed by policies that could change with requirements. Two such polices are described below.

Boundary Adjustments As object migrations should occur after adjustments, boundaries closer to clustered objects should be adjusted with priority. Adjustments should also be incremental so that disruptions are minimized. VSM currently involves all enclosing arbitrators where each arbitrator would move towards or away from a requesting arbitrator a fixed distance between them. To avoid boundary *threshing*, an arbitrator may temporarily ignore a request if its own boundaries also require adjustments.

Arbitrator Insertions Excessive load migrations may result if a new arbitrator is placed at a crowded region, while placement at a sparse region will not help to relieve load. Choosing the right position therefore can be delicate. We observe that the intersection of Voronoi edges represents an equidistant point to all the surrounding arbitrators, and may equalize object transfer costs among neighboring arbitrators. Our current policy thus is to insert new arbitrators at one of such intersections. In case no such point exists (for regions near the boundary of the map), the projections of an overloaded arbitrator's x and y coordinates to the boundaries are then selected.

5. DISCUSSIONS AND CONCLUSION

VSM supports existing event-based consistency control via localized replication of game states. It is responsive as any *trigger - process - display* sequence takes at most three end-to-end hops (i.e. peer → managing arbitrator → enclosing arbitrator → peers in nearby region). With event-based consistency control, no object locking is necessary even for cross-border interactions. VSM can scale as client resources can be added to the system, while dynamic region adjustments help to balance loads. Fault tolerance is supported by state replications on enclosing arbitrators, which may immediately assume managing responsibilities if nodes fail. Finally, virtual peers allow resources from both servers and clients to integrate seamlessly in the same framework, thus providing a bridging transition towards P2P-based MMOGs.

Few work exists on state management for P2P VEs. *SimMud* [18] supports basic state management via supernodes within fixed-size rectangular regions. VSM has gone further to allow dynamic region partitioning, and proposes a simple method to utilize existing consistency schemes. *Colyseus* [5] supports first person shooter (FPS) games based on DHTs. However, the $log(n)$ query in DHT prevents continuous object discoveries and may create unacceptable latency for large node size, whereas VSM performs responsive object discovery with bounded query hops by using VON [14]. Voronoi partitioning for DHT overlays was proposed by Naor and Wieder [25], although adjustments for node locations were not considered. Chen and Lee first suggested the use of Voronoi for VE partitioning [7], but without detailing actual designs. Ohnishi et al. [27] describe a Delaunay (dual of Voronoi diagrams) overlay for VEs, but without considering state management.

We are now evaluating VSM via simulations with regard to bandwidth usage and arbitrator placement policies. Minimal edge-length Voronoi is also a topic under investigations.

6. REFERENCES

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

[2] M. Assiotis and V. Tzanov. A distributed architecture for mmorpg. In *Proc. Netgames*, 2006.

[3] F. Aurenhammer. Voronoi diagrams-a survey of a fundamental geometric data structure. *ACM CSUR*, 23(3):345–405, 1991.

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

[5] A. Bharambe et al. Colyseus: A distributed architecture for multiplayer games. In *NSDI*, 2006.

[6] M.-C. Chan et al. An efficient and secure event signature (eases) protocol for peer-to-peer massively multiplayer online games. In *Proc. Networking*, 2007.

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

[8] J. Chen et al. Locality aware dynamic load management for massively multiplayer games. In *Proc. PPoPP*, pages 289–300, 2005.

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

[10] E. Cronin et al. An efficient synchronization mechanism for mirrored game architectures. *MT & A*, 23(1):7–30, May 2004.

[11] E. Deelman and B. K. Szymanski. Dynamic load balancing in parallel discrete event simulation for spatially explicit problems. 1998.

[12] T. Fritsch et al. The effect of latency and network limitations on mmorpgs: a field study of everquest2. In *Proc. Netgames*, pages 1–9, 2005.

[13] R. M. Fujimoto. Parallel discrete event simulation. *CACM*, 33(10):30–53, October 1990.

[14] S.-Y. Hu, J.-F. Chen, and T.-H. Chen. Von: A scalable peer-to-peer network for virtual environments. *IEEE Network*, 20(4):22–31, 2006.

[15] J.-Y. Huang et al. Design of the server cluster to support avatar migration. In *Prof. VR*, 2003.

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

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

[18] B. Knutsson et al. Peer-to-peer support for massively multiplayer games. In *INFOCOM*, pages 96–107, 2004.

[19] D. Kushner. Engineering everquest. *IEEE Spectrum*, 42(7):34–39, July 2005.

[20] F. Lu et al. Load balancing for massively multiplayer online games. In *Proc. Netgames*, 2006.

[21] J. C. S. Lui and M. F. Chan. An efficient partitioning algorithm for distributed virtual environment systems. *IEEE TPDS*, 13(3):193–211, March 2002.

[22] P. Morillo et al. An adaptive load balancing technique for distributed virtual environment systems. In *Proc. IASTED ICPDCS*, pages 256–261, November 2003.

[23] J. Muller et al. A proxy server-network for real-time computer games. *LNCS*, 3149:606–613, 2004.

[24] J. Muller and S. Gorlatch. Rokkatan: scaling an rts game design to the massively multiplayer realm. *ACM CIE*, 4(3), 2006.

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

[26] B. Ng et al. Multi-server support for large scale distributed virtual environments. *IEEE TMM*, 7(6):1054–1065, December 2005.

[27] M. Ohnishi et al. Incremental construction of delaunay overlaid network for virtual collaborative space. In *Proc. C5*, pages 77–84, 2005.

[28] S. Pekkola et al. Collaborative virtual environments in the year of the dragon. In *CVE*, pages 11–18, 2000.

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

[30] S. Saroiu, P. K. Gummadi, and S. D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proc. MMCN*, January 2002.

[31] S. Singhal and M. Zyda. *Networked Virtual Environments.* ACM Press, 1999.

[32] J. Smed et al. Aspects of networking in multiplayer computer games. In *Proc. ADCOG*, pages 74–81, 2001.

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