Delaunay State Management for Large-scale Networked Virtual Environments

Chien-Hao Chien, Shun-Yun Hu and Jehn-Ruey Jiang
Department of Computer Science and Information Engineering
National Central University, Taiwan, R.O.C.
chienhao1@gmail.com, syhu@yahoo.com, jrjiang@csie.ncu.edu.tw

Abstract

Peer-to-Peer (P2P) networks have been proposed as a promising approach to create more scalable Networked Virtual Environment (NVE) systems, but P2P-NVE also increases the probability of cheating by allowing users to manage the states of objects. In this paper, we propose Delaunay State Management (DSM), a P2P-NVE state management scheme that divides the whole virtual world into many triangular regions by Delaunay triangulation. In DSM, each region is managed by three super-peers, whose collective decisions determine how states will change. Assuming that at most one of the three super-peers is malicious, effective anti-cheating can be provided. Additionally, we also describe how DSM provides essential state management functions (e.g., consistency, load-balancing, and fault-tolerance), and conclude with its potential applications.

1. Introduction

Most of today's commercial Networked Virtual Environments (NVEs) [16] such as the Massively Multiplayer Online Game (MMOG) *World of Warcraft* [1], or the social virtual world *Second Life* [15], adopt the Client/Server (C/S) architecture. These NVEs allow users to create virtual identities called *avatars* to perform jobs, and interact with other users, by changing the *states* of various virtual objects. These object states, or *game states*, often include attributes about an object such as its name, type, location, and properties. In C/S architectures, the states are managed by a server. To change these states, users need to first send *events* to the server, who would process and simulate the events to modify the game states. The server then sends *updates* to notify users who are affected. This event-update model [7] is used in almost all current commercial NVEs. As the server would incur a heavy load with an increasing number of users (e.g., Second Life has over 45,000 peak concurrent users), partitioning the world into various zones, and assigning servers from a server-cluster to manage each zones, has become a popular model for large-scale NVEs. However, as the number of users increases, maintaining a server-cluster becomes ever more complicated and costly.

Unlike C/S architectures, a peer-to-peer (P2P) architecture does not have a central bottleneck, as user-side resources can be continuously added to support the system. Several schemes for P2P-based NVEs (P2P-NVEs) [2, 3, 6, 8, 12] have thus been proposed to eliminate the central bottleneck to improve scalability at lower costs. Unfortunately, P2P also increases the probability of user cheating and the difficulty to maintain game state consistency on the user machines. Additionally, as user-supplied resources are less reliable (e.g., they can crash or leave at any time), fault tolerance also becomes an important issue.

In this paper, we propose a super-peer-based P2P state management scheme called *Delaunay State Management* (DSM). DSM partitions the virtual world via *Delaunay triangulation*, and lets three super-peers to collectively manage each triangular region. Super-peers thus oversee and co-work with each other to provide dynamic load balancing, anti-cheating via *mutual checking* [11], as well as basic consistency and fault tolerance.

The rest of this paper is organized as follows: Section 2 provides background on NVE consistency models and architectures. We present DSM's design in Section 3, discuss its properties in Sections 4, and conclude in Section 5.

2. Background

2.1. Consistency model

Ensuring two or more interacting users have the same view is an important issue in NVEs. Two main consistency models have been used for current networked games: *event-based* and *update-based* models [7]. The most popular model for current MMOGs is update-based consistency, where all clients send *events* to and receive *updates* from a server. In this model, only the server runs the *game logic* (i.e., game rules on how behaviors and actions produce outcomes) to modify states and sends the updates to relevant

users. When conflicting views exist between the server and clients, the server is considered correct and authoritative. In event-based model, all participating nodes are equal (such as in *real-time strategy*, or RTS games), and game logic is run at all nodes. Nodes exchange *event* messages only, and synchronized event ordering is the basis for consistency.

2.2. Centralized server architectures

In order to increase the number of concurrent users, two main designs for C/S architectures have been used [7]:

1) **Zone-based.** The virtual world is divided into several regions, where each region is assigned to a server. The servers are then connected by private high speed networks. For example, *Second Life* [15] consists of many *regions* of size 256x256 meters, each managed by a dedicated server. Users see and interact with other users within the same *region*, but may also go across (i.e., transfer) to a neighboring *region* by walking. Zone-based approach scales the virtual world by simply adding more server machines. However, a high density of users in a region can still degrade the quality of service, and even cause server crashes.

2) **Replication-based.** The *Mirrored Server* architecture [18] is based on the idea of fully replicating all game states at several servers connected by a high-speed network. Servers exchange among themselves event messages with an event-based consistency model, and clients can connect to any server based on latency or load balancing considerations. Several servers together provide better fault tolerance and load balancing. The main problem of Mirrored Server is that each server has to synchronize game states with others, so the communication cost per server grows at $O(n)$ (n is the number of users), and the total cost can be $O(n^2)$.

2.3. P2P-NVE architectures

To address the limitations in C/S designs, recently many P2P-NVEs have been proposed. Solipsis [6] and HyperVerse [3] both aim to build a massive virtual world by providing distributed neighbor discovery and state management. SimMud [12] supports basic state management on a DHT overlay with fixed-size isolated regions. Hu et al. use Voronoi diagrams to discover neighbors on a P2P network [8] and to perform dynamic zone partitioning to balance state management loads [7]. Steiner et al. [17] propose to cluster users on a Delaunay triangulation overlay (dual of Voronoi) so that message delivery can be done more efficiently in batch. RAP [5] is a hybrid design based on static partitioning via triangulations. Each triangular vertex is a *landlord* that runs on the user's machine and handles the state updates in nearby regions. A central server acts as a *tracker* for landlord assignments. However, all these designs have not yet considered anti-cheating.

Izaiku et al. [10] propose a cheat-detection system with event-based consistency and zone-based partitioning. Each zone (called *subarea*) has one *responsible node* and several *monitor nodes* selected from user machines. Within each time *round*, the users send events to both their responsible nodes (who would relay the events to other users), and monitor nodes, who would check the hash values of events from the previous round (attached to the current event) to detect if the responsible nodes have properly relayed events to users.

Enhanced Mirrored Server (EMS) [18] extends the mirrored server architecture by allowing users to exchange messages directly to reduce transmission delay and increase scalability. It utilizes selected super-peers called *referees* for update authentication. In the Peer-to-Peer (PP) model, a user sends messages to other users directly for processing to minimize latency. But to ensure that peers execute game logic correctly, in the Peer-Referee-Peer (PRP) model, each user also sends an update to the referee. After the referee receives and checks this update with all other referees, the update is sent to relevant users. If referee messages conflict with user messages, the user's messages are dropped.

3. Delaunay State Management

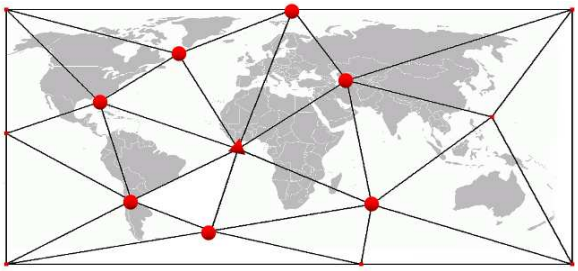

Figure 1. A virtual world partitioned by DT

As the traditional centralized architectures may be costly, here we propose *Delaunay State Management* (DSM) to manage game states more cost-effectively. The key advantages of DSM over existing P2P-NVE schemes are better anti-cheating support and more flexible load balancing. DSM is based on a *Delaunay Triangulation overlay network* [13], where nodes connect with each other to form a Delaunay Triangulation (DT). Given a set of points in the plane, DT divides the plane into a number of triangles such that no point would exist inside the circumcircle of any triangle (Fig. 1). Given a point, we also define its *enclosing neighbors* [8] as any others points reachable by an edge (e.g., circles are the enclosing neighbors of the triangle node in Fig. 1).

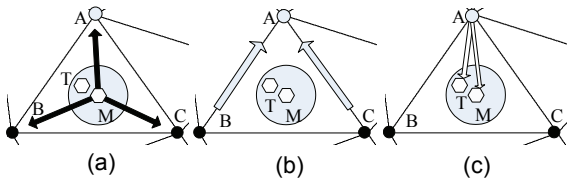

Figure 2. An example of DSM execution (*publisher*: A; *watchers*: B and C; *users*: T and M)

3.1. Design of DSM

DSM adopts the update-based consistency model, and assigns two types of roles for client machines: *user* and *manager*. Users are simply the clients in traditional C/S architectures; they generate *events* to represent actions, and receive *updates* for the affected, relevant states. Managers are the more trustworthy and powerful super-peer nodes, and act as the servers in traditional C/S architectures. Managers have to receive events from users, simulate and process the events, then send state updates to relevant users. As the managers are selected from clients, we cannot guarantee that they never cheat. So, we let multiple managers to handle a region collectively to detect potential cheating.

We use Delaunay Triangulation to divide a virtual world into several triangular regions, each of which is maintained by three managers. All the objects and user states, such as health points and positions, are stored and simulated by these managers. Each user sends events to and receives updates from managers to refresh their game states. During bootstrapping, we first assume the existence of a *gateway server*, which provides a number of *virtual managers* are selected as the initial managers (Fig. 1). When there are few users, virtual managers take care of all state management tasks. But when the number of users increases, we can find some powerful and trustworthy clients to replace the virtual managers to share the load of the server.

The main goal for DSM is to build a robust state management system that provides anti-cheating, while still considers consistency, load balancing (i.e., scalability), and fault tolerance. Our focus on anti-cheating is to check whether each manager follows the correct game logic and sends the correct state updates to users. To do so, a user randomly selects one of the three managers as *publisher* and the other two as *watchers* before sending an event. The role of the publisher rotates among managers across different game *rounds* based on the user's choice. All three managers first process the received event to create state updates independently. The two watchers then send hash values of their updates (called *current update hash*) to the publisher, who would distribute these updates to relevant users after verifying the hash values from all three managers are identical. Managers also keep the hash values of their updates for later validation. To verify that updates received by users have not been tampered by the publisher, users attach a hash value of the previous set of updates (called *previous update hash*) to each event message sent. The hash can then be verified by the watchers to see if the last state update was indeed valid before processing new events. The event processing steps for DSM are as follows:

1) **Event Delivery:** User sends an event to managers.
2) **Event Execution:** Managers process the events according to game logic and game states to create state updates.
3) **Inconsistency Resolution:** Managers detect and resolve conflicts regarding state updates with other managers.
4) **Update Distribution:** Managers send state updates to relevant users.

Let us take the scenario in Fig. 2 as an example of DSM execution. The area of $\triangle ABC$ is maintained by managers A, B, and C, which are in the same *coalition*. All game states in $\triangle ABC$ are managed and stored by the three managers. In Fig. 2(a), at the *Event Delivery* step, user M selects manager A as the publisher, and sends an event (attached with the *previous update hash*) to the coalition. When the three managers receive the event from M, they first check the hash value. If the received hash matches with the hash of the previous update kept by the managers, it means that M and the managers have the same update for the previous round, so the managers can process this event and generate updates individually at the *Event Execution* step. During *Inconsistency Resolution*, the two watchers (B and C) send the *current update hash* to the publisher A (see Fig. 2(b)). The publisher can then check the received hash values against its own to see if the updates are independently consistent. If so, the publisher sends the updates to relevant users (user T and M) during *Update Distribution* (Fig. 2(c)). If not, steps must be taken for DSM to work properly, as shown in Fig. 3. Below we describe in details, how DSM manages consistency, anti-cheating, load balancing, and fault tolerance.

3.2. Consistency

Due to network latency, managers or users may receive the same events / updates at different times and thus process them in different orders. To maintain consistency among managers and users, we adopt a two-level approach. At the first level, consistency is ensured by assuming that the states kept by managers *collectively* are more correct and authoritative than the users'. Thus, in case of inconsistency, the users have to synchronize their own states with the managers' states. This is in essence, how update-based consistency works. At the second level, consistency among the managers is ensured by majority-voting, or majority-consensus. If inconsistency arises, we assume that the same states held by two managers are correct over any alternative

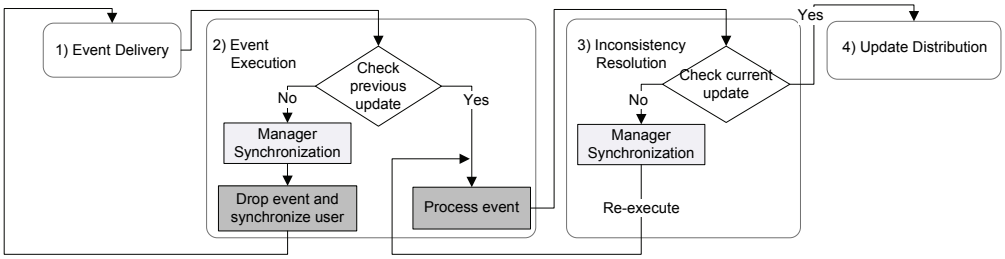

Figure 3. Event processing steps in DSM

held by the third manager. In case all managers disagree, the states should restore to a previously agreed version, and the inconsistent updates are simply discarded (i.e., the event that causes the state updates becomes ineffective).

We note that in a highly dynamic distributed system such as NVEs, consistency is better defined as having the same *update ordering* for each objects than the same *event ordering* as traditionally viewed. The main reason is that while ensuring the same event ordering at different nodes will guarantee consistency, often the costs to maintain strict event ordering is high, while many events are in fact independent (i.e., different executing orders actually would produce the same state updates). Thus, we strive to provide consistent update ordering rather than consistent event ordering. One main method to ensure consistent update ordering is to use hash value comparisons. When users send events to managers, the hash of a collection of previous state updates received by the user is attached to the current event message. Managers then compare this hash value against their own versions to verify that the user has indeed received the proper set of updates for the previous round. This way, we can make sure that updates for the same game object are always applied *consistently* at different user nodes.

There are two check points for state consistency, which we will describe next (see Fig. 3). For now, we only consider the inconsistency caused by network latencies; cheating-related inconsistency is described afterwards.

1) The first check point is when the *previous update hash* is checked during the *Event Execution* step. Such a check is performed by all three managers. When any manager detects inconsistency in hash values, it would stall this event and performs the *Manager Synchronization* procedure. The goal for the procedure is to make sure that all the managers have the same set of updates regarding the state changes for a particular user before they process the events from this user. It is done as follows: the first manager triggering the procedure sends the other two managers a sync message (SYN) that contains the previous update for the user. Upon receival, a manager also sends to the other two managers a SYN message attached with its previous updates for the user in question. At this point, each manager should know about the other two managers' previous updates for the user, and thus could resolve inconsistency based on the majority version from these updates. The managers would then drop this event, while the triggering manager notifies the user of the correct previous update. Afterwards, the user may resend this event based on a corrected previous update.

2) The second check point is when publisher compares *current update hash* provided by the other two watchers at the *Inconsistency Resolution* step. If the publisher detects inconsistent hash values, it means that all three managers do not have the same set of updates. For such a case, the *Manager Synchronization* procedure is run again with the SYN message attached with managers' *current updates* instead of the previous updates of the user. The manager whose update differs from the other two thus can be corrected. Afterwards, the event should be re-executed at all three managers based on the now-consistent states at each managers.

3.3. Anti-Cheating

Anti-cheating becomes especially difficult in P2P environments, as the game states are updated by clients, malicious users may thus manipulate the states easily. Although we might try to find the more trustworthy users to take the manager roles [9, 14], there is still no guarantee that malicious acts would never occur. So besides identifying trustworthy clients, safety measures must still be taken in case of malicious behaviors.

For anti-cheating, we first assume that a sufficient number of trustworthy managers exist and that malicious acts are rare. If at most one of the three managers is malicious at any given time, then its actions can be masked by the other two managers. One important observation is that the effects of cheating are, in fact, inconsistent states between malicious and non-malicious users. So, *cheating can simply be seen as a form of inconsistency, and can be corrected by following the consistency maintenance algorithms*. The intuitive idea behind DSM's anti-cheating mechanism thus is: managers are assumed to be more trustworthy than users (to address user cheating), and the majority is more trustworthy than the minority among the managers (to address manager cheating). Below we discuss both user- and manager-cheating, and how they are respectively handled.

3.3.1 User Cheating

Users in DSM cannot send events or updates to other users directly; they can only exchange messages with managers. Malicious behaviors thus can be detected by managers under the assumption that the managers act properly.

Inconsistent events Users may send different events to different managers to cause inconsistency. However, as all events are processed by all three managers to create updates, the publisher can detect the inconsistency in updates by checking the hash values of the updates from the two watchers, and thus discard the event.

Suppressed updates Users may deliberately ignore updates from managers or stop sending events. However, a user gains no advantage by suppressing updates. As the managers are ultimately authoritative regarding game states, the user's locally modified states will not affect other users'.

3.3.2 Manager Cheating

We assume at most one out of three managers for a given region is malicious. Malicious acts thus can be masked out by the two benign managers. In case of continuous cheats, benign managers can request replacement for the offender. We discuss the two main manager cheats below.

Publisher cheating Publishers are responsible to validate the state updates with the other watchers and distribute updates to users. To prevent publishers from sending incorrect updates to users or from ignoring some state updates, the following two actions must be taken: 1) Each user attaches a hash value of the previous set of received updates to each event message sent. If either watcher detects inconsistency between its hash and the user's hash, it performs the Manager Synchronization procedure to correct the inconsistency. 2) To prevent a publisher from sending incorrect messages continuously. Users would rotate their choices for publishers before sending events. 3) When the cheating user and publisher are the same node, the cheater may select itself as the publisher to avoid detection. However, the malicious publisher will produce inconsistent states from the other two managers, and would be detected by the two managers in the next round. *Manager Synchronization* then would be triggered, nullifying the impact of the cheat.

Watcher cheating Watchers process events to generate updates, and verify if the last update received by each user is valid. Watchers thus may produce incorrect updates or improperly trigger *Manager Synchronization*. Inconsistent updates of the watchers is detected by the publisher, which checks the updates' hash values of all three managers. When a publisher detects inconsistency, a majority decision via *Manager Synchronization* among managers is used to determine the correct update. In case all three managers have different views, the states have to rollback to some common version.

3.4. Load Balancing

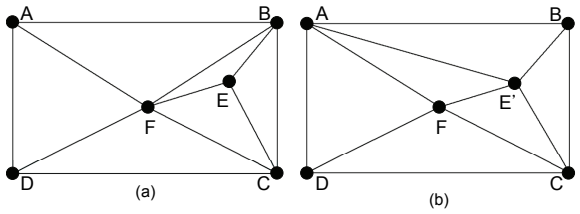

Figure 4. An example of the DT flip operation

The purpose of load balancing is to adjust the managers' handling regions in order to reduce the workloads of overloaded managers. When a manager finds itself overloaded, it would send an *overload message* to all of its enclosing neighbors, some of which will then adjust their positions to move closer to the requesting manager while avoiding *flip operations* of the DT overlay. A flip operation occurs when an existing DT edge is suddenly replaced by another edge due to a changed node position. As shown in Fig. 4, when manager at position E moves to position E', the movement would cause a flip operation. Because flip operations can cause drastic changes in object ownerships for nearby managers, managers should change positions in ways that would not cause flip operations. As such, the movement of a manager should be approved by the requesting manager to avoid simultaneous manager movements, which may cause unexpected flip operations.

If all the enclosing neighbors of the requesting manager cannot move without causing flip operations, the requesting manager should invite a new manager to join the DT overlay at a nearby location to share some load. Ideally, load balancing should ensure that the load of each manager does not exceed the manager's capacity. As shown in Fig. 5(a), if the circular area is a *hotspot* (i.e., an area crowded with many objects and users), and if the manager handling the area, A, finds itself overloaded, it will send overload messages to all its enclosing neighbors. Some responding managers, e.g., manager B in Fig. 5, will move closer to A to help reduce A's load. Note that B's own load is not changed after the movement; however, some enclosing neighbors of B may now be responsible for areas originally handled by A.

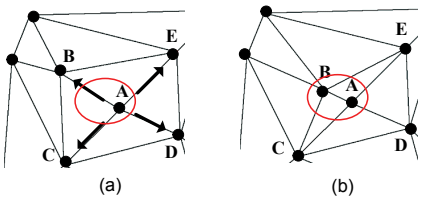

Figure 5. Load balancing model

3.5 Fault Tolerance

Fault tolerance is naturally supported in DSM, where all game states are stored by three managers. As we let three managers to handle a given region, different managers naturally act as backups for each other. When one or two of the managers fails, the game states stored by the other manager(s) are still available. Managers monitor one another for detecting manager failures. Once a manager failure is detected, DSM can recover the game states of the failed manager immediately. Take the scenario in Fig. 5(a) for example. Manager A has four enclosing neighbors, and the states within the triangular region $\triangle ABC$ are saved by A, B, and C individually. As a Mirrored Server-like architecture is used by DSM, when A fails, the states of A are already replicated on A's enclosing neighbors, and thus can be restored to a replacement manager.

4. Discussions

Objects in a virtual world may be classified as either *dynamic* or *static*, according to their mobility patterns. Dynamic objects such as user avatars, have high mobility and generate events frequently, while having more relaxed consistency requirement. Static objects, such as treasure boxes, are stationary, and may update states less frequently, while requiring stricter consistency. As DSM provides stronger consistency and anti-cheating, at a potentially higher message and latency costs, it may be more suitable to manage static objects or serve as a storage layer for P2P-NVEs.

As managers can be any selected super-peers, the manager overlay may be best constructed by considering network latency and super-peer trustworthiness. For example, we could select managers with smaller latencies with their enclosing neighbors to minimize communication or ownership transfer costs. If we arrange managers such that the relatively less trustworthy ones are surrounded by more trustworthy ones, then our assumption – at most one of the three managers is malicious – may be achieved with high probability. How to best select and place the managers so that both the physical topology and client trustworthiness are considered will be an interesting future topic.

One final note about the design of DSM is that we provide anti-cheating by addressing state inconsistency. This provides a new way to think about anti-cheating such that if faults can be detected, recovered, and contained, then cheating may also be rendered harmless. The behaviors of malicious clients can thus be masked out by DSM naturally.

5. Conclusion

In this paper, we present DSM, a P2P-NVE state management system with strong consistency and anti-cheating support. Our system is good at managing objects with strict consistency requirements, at the expense of additional latency. DSM thus is suitable for managing important objects or events with strict consistency requirements, such as trading. Considering more dynamic objects, other state management schemes such as Colyseus [2] or VSM [7] may complement DSM in managing those objects. The mutual checking mechanism in DSM is general, so application to other types of triangulations [4] may also be possible. For future work, we will evaluate DSM's design through implementations and/or simulation experements, and try to bring better supports for dynamic objects into DSM.

References

[1] World of warcraft. http://www.worldofwarcraft.com, 2008.

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

[3] J. Botev et al. The hyperverse - concepts for a federated and torrent-based "3d web". In *Proc. MMVE*, 2008.

[4] E. Buyukkaya and M. Abdallah. Efficient triangulation for p2p networked virtual environments. In *NetGames*, 2008.

[5] K. Endo, Y. Yang, and Z. Zhang. Rap: Reliable peer to peer network gaming environment using overlapped map tessellation. In *MIT Course Project 6.824 Report*, 2006.

[6] D. Frey et al. Solipsis: A decentralized architecture for virtual environments. In *Proc. MMVE*, 2008.

[7] S.-Y. Hu, S.-C. Chang, and J.-R. Jiang. Voronoi state management for peer-to-peer massively multiplayer online games. In *Proc. NIME*, 2008.

[8] S.-Y. Hu et al. Von: A scalable peer-to-peer network for virtual environments. *IEEE Network*, 20(4):22–31, 2006.

[9] G.-Y. Huang, S.-Y. Hu, and J.-R. Jiang. Scalable reputation management for p2p mmogs. In *Proc. MMVE*, 2008.

[10] T. Izaiku et al. Cheat detection for mmorpg on p2p environments. In *Proc. NetGames'06*, 2006.

[11] P. Kabus et al. Addressing cheating in distributed massively multiplayer online games. In *Proc. NetGames*, 2005.

[12] B. Knutsson et al. Peer-to-peer support for massively multiplayer games. In *Proc. INFOCOM*, 2004.

[13] J. Liebeherr, M. Nahas, and W. Si. Application-layer multicasting with delaunay triangulation overlays. *IEEE JSAC*, 20(8):1472–1488, Oct 2002.

[14] V. Lo, D. Zhou, Y. Liu, C. GauthierDickey, and J. Li. Scalable supernode selection in peer-to-peer overlay networks. In *Proc. HOT-P2P'05*, 2005.

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

[16] S. Singhal and M. Zyda. *Networked Virtual Environments: Design and Implementation*. ACM Press, 1999.

[17] M. Steiner and E. W. Biersack. Ddc: a dynamic and distributed clustering algorithm for networked virtual environments based on p2p networks. In *Proc. CoNEXT '05*, 2005.

[18] S. D. Webb, S. Soh, W. Lau, and W. Lau. Enhanced mirrored servers for network games. In *Proc. NetGames '07*, 2007.