國　立　中　央　大　學

資 訊 工 程 研 究 所

碩　士　論　文

Voronoi Diagram Based State Management for Peer-to-Peer Virtual Environments

基於范諾圖之同儕式網路虛擬環境狀態管理

研 究 生：張少榛

指導教授：江振瑞　博士

中　華　民　國　九　十　七　年　七　月

國立中央大學圖書館
碩博士論文電子檔授權書

(95 年 7 月最新修正版)

本授權書所授權之論文全文電子檔(不包含紙本、詳備註 1 說明)，爲本人於國立中央大學，撰寫之碩/博士學位論文。(以下請擇一勾選)

(v)**同意** (立即開放)

( )**同意** (一年後開放)，原因是：_____

( )**同意** (二年後開放)，原因是：_____

( )**不同意**，原因是：_____

以非專屬、無償授權國立中央大學圖書館與國家圖書館，基於推動「資源共享、互惠合作」之理念，於回饋社會與學術研究之目的，得不限地域、時間與次數，以紙本、微縮、光碟及其它各種方法將上列論文收錄、重製、公開陳列、與發行，或再授權他人以各種方法重製與利用，並得將數位化之上列論文與論文電子檔以上載網路方式，提供讀者基於個人非營利性質之線上檢索、閱覽、下載或列印。

研究生簽名: 張 少 榛　　學號： 955202049

論文名稱: 基於范諾圖之同儕式網路虛擬環境狀態管理

指導教授姓名： 江 振 瑞

系所 ： 資 訊 工 程 所 □博士班 ■碩士班

日期：民國 97 年 7 月 8 日

備註：

1. 本授權書之授權範圍僅限電子檔，紙本論文部分依著作權法第 15 條第 3 款之規定，採推定原則即預設同意圖書館得公開上架閱覽，如您有申請專利或投稿等考量，不同意紙本上架陳列，須另行加填聲明書，詳細說明與紙本聲明書請至 http://blog.lib.ncu.edu.tw/plog/ 碩博士論文專區查閱下載。
2. 本授權書請填寫並**親筆**簽名後，裝訂於各紙本論文封面後之次頁（全文電子檔內之授權書簽名，可用電腦打字代替）。
3. 請加印一份單張之授權書，填寫並親筆簽名後，於辦理離校時交圖書館（以統一代轉寄給國家圖書館）。
4. 讀者基於個人非營利性質之線上檢索、閱覽、下載或列印上列論文，應依著作權法相關規定辦理。

論文名稱：基於范諾圖之同儕式網路虛擬環境狀態管理
校所組別：國立中央大學　資訊工程研究所　頁數：35
研究生：張少榛　指導教授：江振瑞　博士

中文摘要：

如同像大型多人線上遊戲 (Massively Multi-player Online Games, MMOGs) 這樣的網路虛擬環境 (Networked Virtual Environments, NVEs) 隨著時間逐漸變得越來越受歡迎。現在的系統大多採用以伺服器爲主的主從式架構 (Client-server architectures)，但此架構也因爲同一伺服器所能同時服務人數有限，進而限制了可能的同時最大使用者。另一方面，同儕式 (Peer-to-peer) 網路逐漸被證明它可被用來解決許多網路應用的可擴充性 (Scalability) 問題。透過分享與使用網路節點 (Peer) 的資源，同儕式網路將可提供資源解決伺服器資源不足的問題。

我們提出了以同儕式網路爲基礎之網路虛擬環境狀態管理系統，稱之爲范諾圖狀態管理 (Voronoi State Management, VSM)，主要用以解決以同儕網路爲基礎的虛擬世界中的物件管理問題。本系統使用范諾圖 (Voronoi Diagram) 來分割環境，並將狀態管理成本分散至化身物件處於鄰近的節點上。當系統負載正常時，所有的使用者可透過直接連線的方式交換狀態更新資訊，但在需要時（系統負載過重，或是使用者電腦資源不足以負擔該區的管理所需時），將會尋找並喚起能力較強之使用者電腦來擔任集中管理者（Aggregator，簡稱集管者）。集管者將會同時管理多個小區域來減輕其他使用者電腦之負載過重情形；同時也會動態地調整其管理區域的大小來平衡系統負載。透過模擬結果顯示，本系統可支援一個網路虛擬環境所需的一致性 (Consistency) 、可擴充性和負載平衡 (Load balancing) 等特性。

關鍵字：同儕式網路、網路虛擬環境、范諾圖、狀態管理

Voronoi Diagram Based State Management for Peer-to-Peer Virtual Environments

Student : Shao-Chen Chang Advisor : Prof. Jehn-Ruey Jiang

Department of Computer Science and Information Engineering,

National Central University

Jhongli City, Taoyuan, 320, Taiwan

Abstract– Networked Virtual Environments (NVEs), such as Massively Multi-player Online Games (MMOGs), have become more and more popular nowadays. Current systems use server-based architectures which possess bottlenecks for the number of concurrent online users on a single server. Peer-to-Peer (P2P) systems have been shown as a feasible solution to scalability in many network applications. Through the resource sharing of peers, P2P systems can be seen as an additional source of resources for improving the lack of server resources.

We propose a state management strategy for supporting P2P-based virtual environments called Voronoi State Management (VSM). By using Voronoi diagram to divide the environment, VSM can distribute the management loading of the system onto selected nodes. Every peer in VSM represents as one site on the Voronoi diagram, and manages the nearest Voronoi cell. When load increases due to a higher density of objects/peers, VSM promotes a capable node called aggregator to join the overloaded area and take over the loads. An aggregator also dynamically adjusts its covering area according to system load. Simulation results show that VSM can achieve the NVE property of consistency, scalability, and load balancing.

Keywords–Peer-to-Peer, NVE, Voronoi diagram, State management

Table of Contents

List of Figures

1 Introduction

A *Networked Virtual Environment* (NVE) is a virtual world generated by computers, in which users may be geographically distributed and control their virtual characters, called *avatars*, to interact with each other via message exchanges. NVE has become more and more popular nowadays, for example, a famous Massively Multi-player Online Game (MMOG), World of Warcraft [1], has more than one hundred thousand online users simultaneously. Second Life [2], which is a social-based NVE, also has over 10 million registered accounts, of which over 1 million are active (i.e., has been on-line in a recent two-month period). To host such massive number of users, the system must be scalable and affordable.

A NVE can be seen as composed of many objects, each of which describes an entity that has many attributes called *states*, such as the object's type, shape, color, etc.. There must also be some rules describing how the worlds should work for VEs, and are called *world logic*, or *game logic* for games. For example, a rule may state that "everything in the VE is attracted by Gravity, so it will fall if not propped by the ground or other objects." Users who join a NVE may act through a virtual character called *avatar*, who has an area called *Area of Interest* (AOI) within which the user can see and interact. Executing world logic in order to modify and distribute states is called *state management*, and is a fundamental requirement to operate an NVE.

Basic requirements for NVEs are *consistency*, *scalability*, *load balancing*, and *fault tolerance*. Further requirements may include *anti-cheating*, *security*, and *persistency* [3, 4]. Consistency is a basic requirement since a single virtual world should have a consistent view for each participants. Both scalability and load balancing share similar goal of increasing the number of online users for a virtual world. More precisely, scalability focuses on the upper limit of the number of concurrent entities (i.e., users), but load balancing focuses on balancing workload for every participating node. Fault tolerance is an important property since faults may happen due to various reasons, and may cause unpredictable damages to the system. Additionally, potential cheating should be avoided from either a good design or implementation, and security should be ensured. Persistency

is a special property for some massive VEs, it is to ensure that the users' states are stored for a long time period, as long as the VE is still in operation.

Current NVEs adopt server-based architectures due to their maturity and security, where all processing are done at server-side, and clients only send requests or receive updates. State management in server-cluster architecture is a consistency control problem, and a common definition is how to make states consistent between any number of interacting participants at any given time. The goal is to minimize the duration in which states are inconsistent and the side effects (e.g., latency or transmission overhead) caused by the consistency control mechanism. Many schemes have been proposed to achieve better consistency with specific goals [5, 6, 7], or to distribute loads between servers (i.e., inter-server *load balancing*) [8, 9]. When the number of servers is fixed, the total amount of usable resources is limited to the servers' resources, and may not satisfy the requirements to build Massively Multiuser Virtual Environments (MMVEs) which include millions of users. In addition, server-clusters are costly to develop, deploy, and maintain [10]. Some researches [3, 11, 12] thus try to solve this problem with Peer-to-Peer (P2P) architectures.

P2P-based architectures have been proven as a usable and feasible solution for many applications, such as media streaming [13, 14], Voice over IP (VoIP) [15], and especially file sharing [16]. Through sharing the resources of peers (including computation power, bandwidth, storage, and other resources), P2P networks provide the possibility of an unlimited resource pool that grows with the number of peers.

The first problem we face for P2P-based state management is a connectivity problem. For server-based architectures, all servers are known in advance during most of the system operation time, and thus can be located easily by a fixed mapping table, but such is not true for P2P-based architectures. For P2P VE research, this is actually a common *discovery problem*. Several solutions to the problem [3, 11, 12, 17, 18, 19, 20, 21, 22, 23] have been proposed, as well as some improvements [24]. As the discovery problem is more or less solved, further researches [4, 25, 26] then direct at the problem of state management, but no comprehensive solutions have been shown.

In this thesis, we propose a new state management strategy for P2P VE called *Voronoi State Management* (VSM). Basically, VSM divides the VE into cells according to a Voronoi

diagram [27], and uses the positions of the avatars as the associated sites for each cells. Each cell has a manager, called *arbitrator*, which is run on the client whose avatar resides in the cell. Arbitrator of a cell is the owner of all VE objects in the cell, and has the rights and responsibility to modify and distribute the objects' states. When workload increases over a pre-defined threshold, VSM tries to decrease workload by clustering arbitrators and have them taken over by a selected, but capable superpeer called *aggregator*. However, as aggregators still may overload when the object density or event frequency are high, an overloaded aggregator thus may try to decrease the load by reducing its *sphere of control*. As this may raise the workload on peers again, another cluster may be established by similar mechanism. We also evaluate the performance of VSM through simulations of a small hunt-and-gather game.

The rest of this thesis is organized as follows. Requirements for P2P NVE, background, and related work are in Chapter 2. In Chapter 3, a problem formulation for state management is given. Detailed designs and evaluations of VSM are shown in Chapter 4 and 5, respectively. Finally, conclusion of this thesis is given in Chapter 6.

2 Related Work

In this chapter, we will describe the background and related work for VE state management. In the first section, requirements for P2P NVE and background for state management in the past is shown, and the second part consists of two parts: consistency control and server-based architectures. In Section 2, the discovery problem and recent work on P2P state management are introduced.

2.1 Background

2.1.1 Requirements of P2P NVE

As described in [3, 4, 26, 28], there are 7 requirements for P2P NVEs described separately below. The first 4 requirements are fundamental requirements for every P2P-based state management system.

Consistency Consistency is a basic requirement for NVEs, which deals with how to handle and converge differences between the users' views. Since perfect consistency (i.e., all participants have exactly the same view at a given moment in time) cannot be achieved due to the existence of network latency, the main concern for consistency thus becomes how to reduce or bound inconsistency to levels that are acceptable. Furthermore, how much inconsistency is acceptable depends on the application types and requirements. It is also an important factor affecting the users' perception for the environment. If there exists differences significant enough between users' views, users may fail to understand each other or the world and the sociability or the usability of the system thus may reduce.

Scalability An environment may have millions of users, as described in Chapter 1, so how to accommodate a huge number of users must be considered. A common trait of scalable systems is that resources consumed by each node (both clients and servers) will not increase with the number of users. In another word, it is easy to imagine that every single peer consumes some computation and communication resources of the system to interact with the environment, and given a fixed resource pool (e.g., a fixed-size server

cluster), after a certain number of users have joined the system, resources will run out. The issue that considers the total number of users is called *system scalability*, and another type of scalability is *AOI scalability*, which focuses on increasing the number of users within the range of an AOI radius.

Load balancing Different from scalability that considers system-wide workload distribution, *load balancing* concerns how to avoid overloading for any participating hosts (i.e., both clients and servers). Since load may distribute to all participants randomly (depending on the system strategy), overloading may happen anytime. A system that is load balanced can deal with the situation by transferring or decreasing the work load before more critical problems (e.g., node failures) happen. This property becomes more and more important as more load sharing and distribution schemes are adopted, both for server-cluster or peer-to-peer architectures.

Fault tolerance Failure of participating nodes may incur critical system problems, like data loss or system crash, fault tolerance thus is an important property. Different problem considerations exist for peer-to-peer or server-based architectures. For server-based architectures, most of the problems it faces are hardware or software failures. However, as peer-to-peer networks consist of peers that may have significantly different capacities and high dynamics (i.e., peer may join or leave the network at any moment), so the problem of node failures (i.e., leaving suddenly) must be considered more seriously.

The following are advanced requirements which may not be needed by all applications, but can still be critical issues for some specific environments.

Cheat avoidance The problem of cheating comes from the unpredictable trustworthiness of participating nodes. For server-based architectures, the server(s) is always trustable and under controlled, any decisions made are thus authoritative and considered as the final results. For a peer-to-peer network, since each nodes may modify any program run on itself, they should therefore be treated as potential cheating nodes. Since we cannot predict if a peer cheats or not, whether states are advanced normally also becomes unpredictable.

Security Security and privacy control are basic requirements for all types of applications before they can be practically adopted in the real world. Remaining security issues for hosting virtual environments excluding cheating, are mainly users privacy control and protection of users' wealth in the environment. This problem is considered as the last step (together with cheating avoidance) before a system becomes applicable.

Persistency Persistency is a critical property for specific types of applications, like MMOGs, since they must be on-line 24 hours a day. The goal is to ensure that the states in a world can be preserved while changing forward. Players may log in and out of the server at any time, while retaining their status (e.g., levels, experience points, amount of in-game currency). Persistency is usually provided by transactional databases that keep continuous records of all important game state updates.

2.1.2 Network Model and Consistency Model

State transitions in VEs can be seen as operations of finite state machines, where states transit by world logic and inputs (i.e., *events*) generated by user actions or VE semantics. A *world logic* is a set of rules on how the VE operates, similar to laws of Physics for the real world. Once states have been modified, state *updates* are then distributed. State management therefore can be understood as a *request* - *process* - *update* - *display* sequence.

A state management system for NVE can be understood from its *consistency model* (i.e., how state updates and distributes across nodes) and *networking model* (i.e., how nodes connect and communicate). The consistency model and networking model usually depend on each other, and if not, consistency may not be guaranteed. In networking, two main architectures are *point-to-point* (also known as peer-to-peer, to distinguish between them, the word here indicates fully connected peer-to-peer network) and *client-server* (including traditional client-server or server-cluster networks). In point-to-point, peers connect to all other peers for exchanging messages. It usually has the lowest transmission latency between peers, but heavy bandwidth requirement due to its $O(n^2)$ communication cost. In client-server, all nodes connect to the server(s), and can only exchange messages via the server(s). Differ from point-to-point, client-server has a cheaper communication

cost of $O(n)$ [29], but server(s) may become the single point of failure.

Two common consistency models in use today are the *update-based* and the *event-based* models. Update-based consistency, as the name implies, is a consistency model based on distributing updates, or more precisely, executing the 'event then update' sequence. It is often used on client-server architectures, where a central authority (i.e., the owner) is assigned for each object, and has the rights to modify and distribute the objects. Owners also serialize and execute the events. Events thus can be seen as requests to update one or more objects, and once some states have been modified, updates are then distributed.

This is the most adopted consistency model for client-server applications which include most of the current MMOGs. In the case of using update-based model with client-server, the server maintains a full set of authoritative states, and consistency is achieved by making the clients' states as closely matched to the servers' as possible. Advantages of this consistency model is that the owner of objects can advance logic clock without considering networking latency or waiting for other late coming messages, so the VE's operations will not be affected by the failures of clients. On the other hand, the failure of an owner will affect, at least, its managed objects, or even crash the whole system in the worst case.

Event-based consistency model stands from the basis that if two or more nodes have the same initial states and execute the same events, they will have the same final states [30]. Starting from this basis, all nodes running event-based consistency model need to exchange all relevant events occurred at every time step, and then process all events respectively. As long as each node has the same states and run events in the same sequence (may just need to be roughly the same, depending on the consistency requirements), states on all nodes should be more or less consistent. In the process, it is important that the sequence of event execution must be synchronized, so various *conservative* (e.g., lock-step) and *optimistic* (e.g., Time Wrap [5], TSS [6], OSS [7], ILA-RED [31]) synchronization schemes have been proposed. This kind of consistency model has the advantage that when there are a large number of objects following the same logic, we can use just a few event messages to update them all [30] without having to distribute a large number of state updates. But it also inherits the same problem from a point-to-point topology, where the communication cost

may become more and more expensive with the number of participating nodes increasing. Strict synchronization among the nodes may also mean that logical time could progress at the speed of the slowest node.

2.1.3 Server-cluster State Management Schemes

To scale up virtual worlds, a common strategy is duplicated worlds, which replicates the VE to two or more duplicated worlds, each of which has exactly the same content, but all are independent, and cannot have any kind of interactions across worlds. Although the number of concurrent online users can be significantly increased, sociality and realism are also sacrificed.

To scale up a single virtual world, three common schemes exist in current server-cluster designs.

Replication-based Replication-based model [6] follows the basic principles of event-based consistency, in that all servers replicate all states in the world, and clients can choose the best server to connect. Event-based consistency is used between servers, who exchange among themselves the received events from clients on every step. Because the set of duplicated servers can be seen as a single server, update-based strategy is used between the clients and the servers. Although networking cost is distributed using the strategy, processing cost still remains unchanged, and may become unaffordable as it grows at $O(n^2)$.

Object-based Object-based model [8] considers not only distributing networking cost, but also processing cost. In this strategy, objects are distributed to servers based on selected optimizing function, and update-based consistency is used between servers. Objects' replicas may be created on servers whose AOI cover the object, where a server's AOI is determined by the objects that it manages. There must be an object querying service to discover new objects, and also to query where an object is located. This strategy is highly adjustable for objects distribution, but there are extra costs to maintain the locations and replicas of objects, and inter-server communication may still be heavy if many objects interact across servers.

Zone-based Zone-based strategies [32] are widely used by current MMOGs. Differ from object-based, zone-based schemes distribute objects onto servers by the positions of objects in the virtual world. Due to the fact that most virtual world interactions are localized, the strategy minimizes the communication needed between servers, and is simpler to design.

2.2 P2P-based Architecture

Several schemes [3, 11, 12, 18] have been proposed to solve the problem of neighbor discovery, and some other work have considered state management [11, 18]. We categorize these schemes into DHT-based or graph-based strategies.

2.2.1 Discovery Problem

DHT-based *SimMud* [11] and *Colyseus* [18] use *Distributed Hash Table* (DHT) overlay as its basis of the network structure. Furthermore, they also consider state management, we will have detailed descriptions later.

Graph-based Solipsis [12, 17] proposes an algorithm for P2P VE to ensure the connectivity of the network. In Solipsis, all nodes connect directly to their AOI neighbors, which must form a convex hull to cover the node itself (Figure 1 (a)). If not (Figure 1 (b)), the node should recover the convex hull by finding additional neighboring nodes. Neighbor discovery is done through mutual notifications by nodes who form the convex hull.

This thesis' origin is based on VON [3, 33], which also proposes a fully distributed P2P overlay network. All nodes in VON connect to their neighbors directly, and exchange messages (only movements and neighbor notifications) through direct connections. A later forwarding scheme has also been proposed as an improvement [24, 34]. Each node in the system uses its own and all neighbors' positions to locally construct a Voronoi diagram. The original definition of Voronoi diagram is that: given a set of points as *sites* on a plane, the plane is divided into multiple cells (i.e., the polygon formed by Voronoi diagram), each of which contains exactly one site. The nearest site to any point in the cell is the site that defines the cell [35]. In VON's design, as shown in Figure 2, *enclosing neighbors* are the

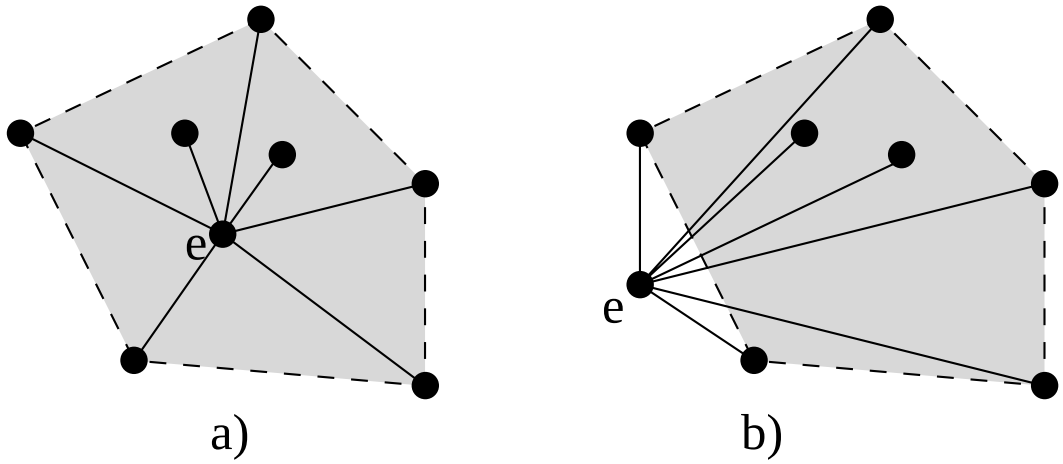

Figure 1: An example of Solipsis: (a) The node e is covered within a convex hull formed by its neighbors; (b) The node e is not covered by the convex hull.

neighbors who share at least one common edge with the node, and *boundary neighbors* are the neighbors whose Voronoi cells overlap the node's AOI. In the system, nodes always minimally keep connections with their enclosing neighbors, and inform boundary neighbors to notify them about new neighbors. By a few simple rules, VON constructs a P2P overlay network with only local information. Additionally, VON also prevents peer overloading by shrinking its AOI to decrease bandwidth consumption without affecting the operation of overlay construction.

2.2.2 Peer-to-Peer State Management

Few work exists that target on state management for P2P VEs. Chen and Lee [36] first suggest to use Voronoi diagram to partition VE, but no further scheme was proposed. *SimMud* [11] proposes a region-based strategy for P2P VE based on *Pastry* [37] and *Scribe* [38]. The VE formed by SimMud consists of a few independent regions, each of which have one coordinator who is the node with the nearest hashed ID value to the region's ID on the DHT formed by Pastry. Message exchange in the region is performed through *application layer multicast* (ALM) by Scribe, where the coordinator is a tree root. As shown in Figure 3, messages in a region are transmitted through multicast or direct connections, and inter-region communication depends on DHT. SimMud supports basic

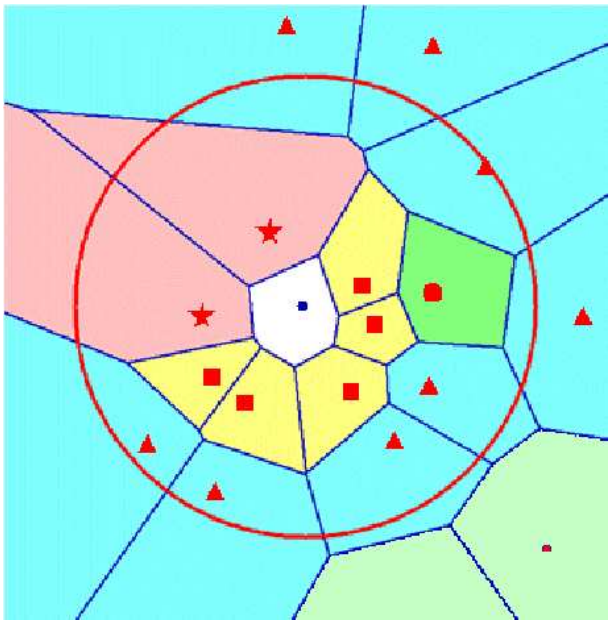

Figure 2: An example of VON
An example of the node's enclosing neighbors (■) and boundary neighbors (▲), and some nodes that are both enclosing and boundary neighbors(★).

state management by coordinators (i.e., superpeers), but fixed and independent regions lack flexibility and applicability. Inter-region interactions are also not considered.

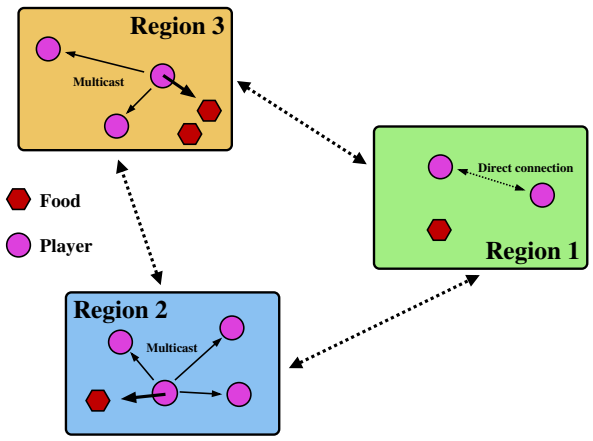

Figure 3: System design of SimMud

Colyseus [18] proposes also a DHT-based scheme, but goes one more step to support simple consistency control, and evaluates their scheme on a First Person Shooter (FPS) game, Quake II. As mentioned earlier, by using the DHT *Mercury* [39] that supports range queries, VE neighbors can be discovered by querying the AOI range of a node on the DHT. The consistency control in Colyseus has only replica control, where each object

in the system has exactly *one* primary copy and zero to more secondary replica copies (or replica for short). When an event is generated, a request to update the object is first sent and serialized at the primary, then distributed to all nodes who has a replica. The system architecture is shown in Figure 4. Although Colyseus contains designs for both consistency and scalability, but the $log(n)$ query latency on DHT for object discovery and the overhead of maintaining DHT may become performance bottlenecks when there are a large number of users in the system.

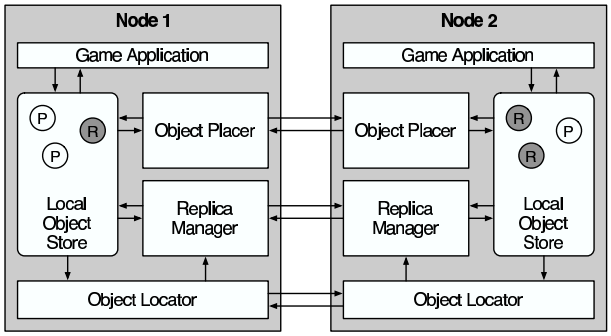

Figure 4: System architecture of Colyseus

Hydra [25] proposes a peer-to-peer architecture that considers fault tolerance issues. Differ from the previous schemes that consider VE plane divisions, Hydra focuses on server architecture and the message exchanging between servers (note the 'servers' may also indicate superpeers for P2P VE). Hydra assumes a world formed by individual regions as SimMud, and use *slices*, each of which manages a region's game states, to run on servers. As shown in Figure 5, there may be multiple slices for one region, one of them is called the primary slice, which is the owner of the region, and the others are backup slices. Clients send events to both the primary slice and backup slices. Once the primary slice executes the events, update commitments are sent to backup slices, where the objects are updated with new states, and the cached events related to the update are cleaned up. When the primary slice fails, one of the backup slices will take over the original primary slice, and execute the cached events which have not been committed by the primary slice, and continue to advance the region's logical time. Hydra proposes a workable fault tolerance architecture, but servers and slices must be tracked by a *Global Tracker* (i.e., a central server) which takes queries for newly joined peers, it may thus be a bottleneck or a single

point of failure.

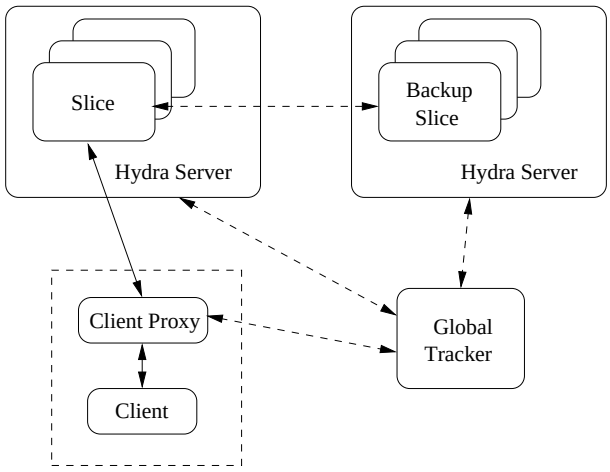

Figure 5: System architecture of Hydra

In [26], the authors try to extend Solipsis to support state management, and propose several topics to consider. Their proposed design includes that each entity (i.e., object) in the virtual world should have an entity descriptor to link objects with their 3D models, P2P overlay construction, decentralized physics computation, object management, and 3D model sharing (also known as P2P 3D Streaming [14]). They have also implemented a web-based navigator to demonstrate their scheme. The authors of Peers@Play project [4, 23] propose the considerations and software architecture to construct a P2P VE. They propose a few requirements for P2P VE, suggestions for VE consistency models, and software architectures. For a P2P VE consistency infrastructure, they suggest to consider flexibility (to adapt the diversity of games), adaptability (to adapt the dynamics of P2P network), and extensibility (consistency model can be extended by game developers).

Buyukkaya and Abdallah [28] propose an interest management algorithm for Voronoi diagram-based state management that could determine the region of broadcasting updates. The strategy uses only local information, and is based on Voronoi diagrams and the assumptions of fixed AOI radius. Through calculating the convex hull of the managed objects and extending the region for AOI, as shown in Figure 6, the potential positions (light-gray region) where other objects and the managed objects may see each other can be decided. By determining the proper subscribing region for updates, the problem of disseminating updates can be easily solved.

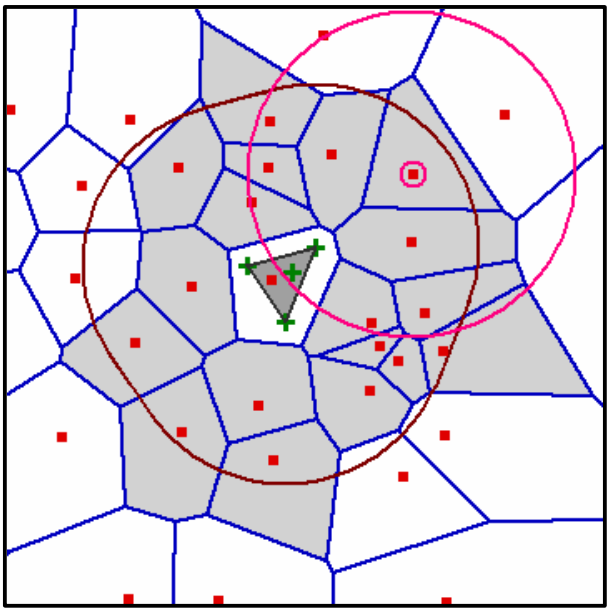

Figure 6: Visibility region of managing objects in Buyukkaya's scheme

3 Problem Formulation

As mentioned, a NVE is a computer-generated world, and to construct a virtual environment, two types of data are needed: fixed and variable data (referred as static and dynamic objects in [28]). *Fixed data*, such as 3D content (3D models/textures), topographies, etc., are rarely changed and huge in data size (in general, at least between 300 to 500 MB, in fact, more than 5 GB is needed for World of Warcraft). Since fixed data is changed infrequently and usually will be pre-installed together with the main program, in most current applications, we do not focus on how it is distributed or modified. Besides, there has been some studies that focus on the topic of progressive transmission of such fixed data (also known as *3D streaming*) [14].

Variable data are the states that may be changed rapidly in the environment, for example, positions of avatars, clothes worn, etc. In VSM, we adopt the model that the environment is described by objects and attributes. That is, the environment consists of many objects, and each object has many attributes to describe its states. For example, a table in the VE may be represented by an object which has a number of attributes describing its features, like its position (where it is placed), type (a dinner table or a coffee table), size (how high and wide it is), color, etc. The set of objects, attributes, or any combinations of them are called the *states* of the environment.

States should be modified according to working rules of the VE, called *world logic* (or *game logic* for game-based environments). World logic is a set of rules describing how the environment works, its rules for progression, or physical limitations, similar to the laws of Physics for the physical world. All state transitions should follow the rules. A basic requirement to build up a networked virtual environment, where the world logic is followed to modify and distribute states, thus requires a *state management* system. To summarize the problem, a formal definition of state management is given in the next section.

Problem Formulation of State Management We use following assumptions about a VE system:

1. A virtual world is a 2D plane with a fixed width and height, and filled with many objects, each of which has its position on the plane.

2. Objects are in the form of *(name, attribute-list, x-coord, y-coord)*. Name is an identifier of the object, x and y-coord defne its position on the plane (shown only for objects lying on the VE plane).

3. Attributes are in the form of *(name, type, value)*. *Name* is also an identifier of the attribute, *type* may be a simple data type such as **int**, **float**, **string** or **object**, and *value* is value of the attribute.

4. *States* (i.e., objects and attributes) are created, updated, and destroyed by processing *events* based on the world logic of the environment. Once objects are updated, modified states need to be distributed to all interested parties (i.e., avatars).

5. Each peer is represented by an *avatar object*, which is a regular object controlled by the player. All players have a fixed and system-specific AOI-radius, within which objects and avatars may have interactions with each other by generating some events (i.e., all events only influence objects within the AOI of the player that generates the event).

4 Proposed Scheme

In this chapter, we will describe the detailed design of VSM. The main purpose of VSM is to support state management on peer-to-peer networks, and the goal we targeting is to manage the distributed states of the virtual world and to balance the loads between participating peers. We first present an overview of the key ideas of VSM in Section 1. Afterwards, we describe the detailed design of VSM. For the last section, some additional issues for improving the performance and extensibility of VSM are given.

4.1 Basic Idea

The basic design of VSM can be seen from two aspects: architectural and operational. From the architectural perspective, we can describe the roles nodes play in VSM, what responsibilities they have, and how they collaborate to manage the world. Based on the architectural design, currently VSM has a distribution-based design for its operations, but we also introduce other possible operational designs in Section 4.3.

VSM defines for participating nodes several roles categorized by their responsibilities, and several roles may be executed on a single node. We first introduce two roles: *peers* and *arbitrators*, which are similar to the clients and the servers in client-server architectures. Users join the environment as peers, who control their avatars by sending action requests as events and receive updates for the environment. As peers receive updates about the environment, they can render the virtual world based on the updated states and the pre-installed *fixed objects* mentioned in Chapter 3.

Arbitrators act as servers to arbitrate the modifications of states by following the world logic and distribute them to those who are interested. All arbitrators share the management of the environment through the divisions via Voronoi diagrams (called Voronoi division for short, see Figure 7). We use similar rules of Voronoi diagram. In VSM, the environment is divided by the arbitrators' positions, and the cell formed is called the *managed area* of the arbitrator. Objects within a given managed area are owned by the arbitrator, who is known as the *managing arbitrator* of the objects. An example of Voronoi division is given in Figure 8, in which circles represents arbitrators and squares

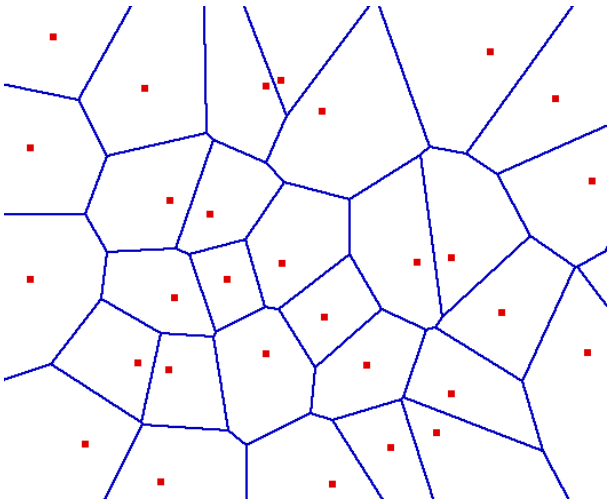

Figure 7: An example of Voronoi diagram

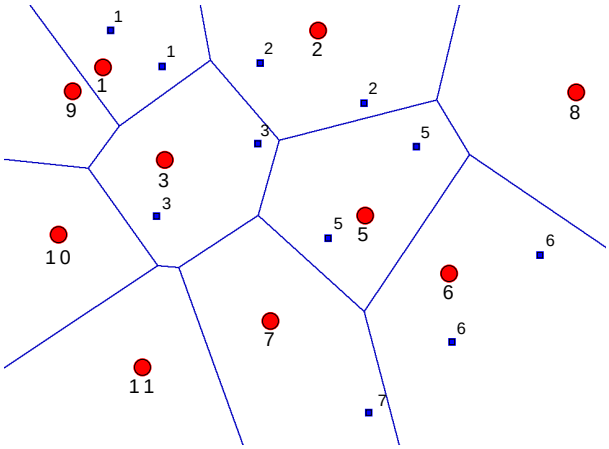

Figure 8: An example of Voronoi division
In the figure, circles (∘) represent arbitrators and squares (□) represent objects, numbers marked on the object indicate the object owner's node id.

represent objects in the VE, and object owners' node IDs are marked besides the objects. In other words, objects are managed by the nearest arbitrator to their positions, and peers are managed by whoever manages their avatar objects. In summary, all arbitrators form a Voronoi diagram, where both objects and peers (i.e., avatar objects) are managed by the arbitrator designated for each Voronoi cells.

From the operational view, VSM follows the idea of a fully distributed network, where each peers can be responsible for the area in which its avatar stay, so that a central resource allocator is not needed. Therefore, we let every node join the system as both peer and arbitrator first, and make the arbitrator move with the avatar object of the peer

(i.e., each peer manages the cell it stays).

4.2 Detailed Design

4.2.1 Consistency Control

VSM adopts the update-based consistency model, that is, each object has an authoritative owner (i.e., arbitrator), where events, which can be seen as requests to modify the object, are serialized and executed. All events sent by a peer should be sent to its managing arbitrator first, then forwarded to owner(s) of the object(s) that may be affected by the event. Avatar objects also have their managing arbitrators, and each peer connects to its managing arbitrator for sending events and receiving updates. Object ownership transfer is needed when the division of authority changes due to movements of the arbitrators or the objects themselves. As small inconsistent views of cell boundaries (caused by inconsistent views of arbitrator positions) or object positions (caused by missing or in-transit updates), ownership transfers thus need explicit message exchange.

4.2.2 Load Balancing

Computation and bandwidth are the main bottleneck resources for arbitrators, and they are affected by the number of managed objects and the activity levels of peers. More specifically, workload (including computation or bandwidth consumption) is determined by the number of objects, the rate of event generation, the complexity of world logic, and the number of observers to whom updates must be delivered. VSM divides the environment into cells and allocates arbitrators to manage them. While object distribution is decided by the application designers and users, unbalanced load (e.g., overload) may happen on any arbitrator. VSM assumes that there exists some load detection mechanism, which can detect clients' loading level (e.g., normal, overloaded, or underloaded). Once the arbitrator's workload increases over some pre-specified threshold, load balancing process is invoked. Contrary to the traditional thoughts on load balancing, VSM tries to cluster overloaded arbitrators onto one capable peer or server. On the first step of the load balancing process, VSM chooses a node from a list of capable peers or, if available, a

powerful server (methods to choose the node will be introduced in Section 4). A new system role, *aggregator*, is then started on a selected node, and joins at the same spot as the overloaded peer's avatar position. An aggregator will cluster and take over arbitrators within a sphere with specific radius (called its *sphere of control*), as shown in Figure 9. However, as aggregators still may be overloaded with a high object density, so it is also allowed to shrink its sphere of control to decrease the number of managed objects (alternatively, it could stretch the sphere of control when underloaded). This may cause other load balancing processes on nearby peers, repeatedly until the load of the whole system is balanced.

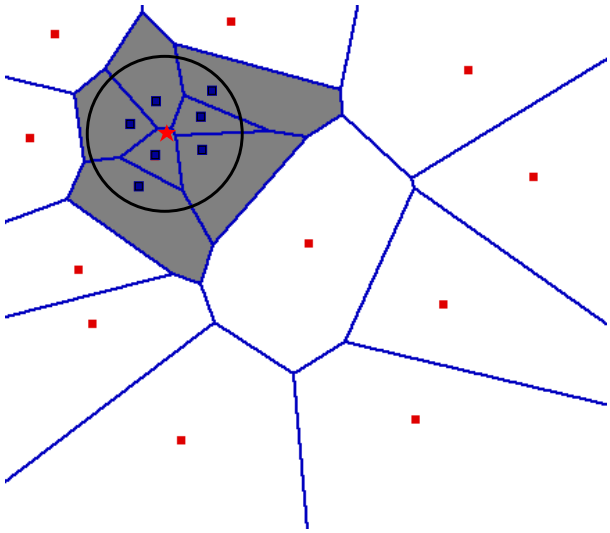

Figure 9: Aggregators take over overloaded arbitrators
Aggregators (★) take over arbitrators (■, blue ones will be taken over) within its sphere of control

4.3 Additional Issues

In this section, we address some problems that are not described in details in the previous sections.

Selection of Capable Peers Capable nodes are used to deal with load unbalance by acting as aggregators, and some possible methods for choosing aggregators are:

- through an external server

- through random-walk spiders
- through gossiping

The simplest and most efficient way is to record nodes' capacity information on a central server, and get the proper nodes from the server when needed. Capable peers will be recorded, together with their capacities, when entering the system. This method is practical and efficient, but relies on extra server resources and may not be too scalable.

Random-walk spiders are similar to searching schemes on unstructured peer-to-peer networks, that is, information of capable peers is collected through a information collection spider who walks randomly through the network. More specifically, the spider starts from the node that wants to collect the information, and randomly chooses one outgoing link to move until suitable nodes are found or a threshold of hop count is reached. Once suitable information is found, a direct connection to the spider-starting node is established to return the information. This collection may be invoked any time the system operates, and the results can be cached for later reference. Collecting information in this way is fully distributed, with only little extra transmission overhead.

The last method gets capable nodes from gossiping, that is, capacity information is periodically exchanged between arbitrators' neighbors (i.e., capable node lists come from neighbors in the past). The method may work well since it is independent of the distribution of capable peers or the distribution of avatar objects. Consider the costs required for the method, it is an efficient way to collect information.

Extensions for VSM VSM integrates peer-to-peer and client-server into a single scheme and is a general-purpose state management scheme for P2P VEs, that can support a variety of different types of virtual worlds by adding some additional support. Session-based VEs can be supported by VSM natively, like first person shooter (FPS) games, in which players interact inside a game room up to a few hours. After a game session/room is closed, nothing remains, and all records/scores in the session will be discard.

Getting one more step, VSM can also be used as a peer-assisted system (e.g., a low-cost MMOG hosted by running VSM with light-weight servers). In the case of MMOG, an external persistent storage service is also needed, but distributing states and running game

logic are already supported by VSM. More specifically, servers may host some pre-inserted aggregators with the full game states, then distribute managing rights of underloaded areas to peers to save server resources and scale up the system.

Operation Designs VSM's architectural design may be used for different purposes by having different operational designs. Here, we just introduce a simple example of using VSM with a powerful sever-cluster. Traditional server-cluster designs divide the virtual world into a few fixed size areas, and replication is used at the boundary of areas. When object distribution becomes clustered, one or more servers in the cluster may become overloaded. Because VSM can dynamically adjust object allocation by moving the positions of arbitrators, we can dynamically re-allocate objects by placing arbitrators more optimally based on the density of objects.

5 Evaluation

To evaluate our design of building a state management system for virtual worlds on peer-to-peer networks, we use simulations for the evaluation of our scheme.

5.1 Simulation Environment

We design a simple *hunt-and-gather* game, the details of its rules are listed below.

1. There are *avatars*, *food*, and *attractors* distributed on the 2D VE plane.
2. Avatars has one basic attribute, *health point*, and can move around everywhere, attack somebody, or eat food. Avatars' moving speed is a constant world parameter.
3. Someone who is being attacked loses its *health point*, but eating some food increases it back.
4. Total amount of food is fixed in the system, but is randomly distributed in the environment. Once some food is eaten, it will be regenerated at a new random position.

Avatars in the environment are controlled by simple AI semantics, and abide to the following rules.

1. An avatar's actions are determined by its *health point* and two stat points, *fatigue* and *anger*.
2. Avatars increase the levels of anger and fatigue with movements. If a pre-defined threshold for anger or fatigue is reached, the avatar gets into an angry or fatigued state.
3. An *angry* avatar will randomly attack another avatar, which decreases its anger stat.
4. A *hurt* avatar (health point is below a certain threshold) will seek some food to eat.

5. A *fatigued* avatar will find the nearest *attractor* to rest, which can be seen as a resting area centered around the attractor. Avatars in attractor's coverage range will have its fatigue stat decrease quickly.

6. Otherwise, avatars move around everywhere by *random waypoint*, that is, it chooses a target point, moves to the point with a constant speed, and chooses another random point to move.

7. Priorities among all actions are eating, resting, attacking, and moving.

We implement VSM using VAST [40] (an implementation of VON) as the underlying overlay network, and the *hunt-and-gather* game on top of VSM. We assume a constant end-to-end latency as one simulation step. For simplicity, no message losses, node failures, or bandwidth limitation is used during the simulation. The simulation is performed on a 2000x2000 map, and between 100 to 500 nodes (avatars) are put inside, with AOI radius of 50 and speed of 1. Update rate is 10 steps per second (i.e., transmission latency is 100ms). Each simulation is executed for 1000 steps (i.e., 100 simulation seconds).

5.2 Simulation Metrics

Several metrics described below are used to evaluate the system:

- Bandwidth: Bandwidth limitation is a main bottleneck for servers in client-server architectures, and is also important for peer-to-peer architectures as it is more limited for peers. From several previous research [18], system availability and quality is directly related to the amount of remaining network resources. Our main concern for the metric is how the bandwidth of aggregators and arbitrators is used, because they are the main service providers in the system.

- Consistency: Consistency is a basic requirement for state management, but perfect consistency cannot be achieved due to network latency or the consistency model used. Therefore, our main concern here is how inconsistent the world is. We separate the consistency of the system into two indicators: *discovery consistency* and *update consistency*.

Discovery consistency(DC) This indicator measures how well objects are discovered for all peers, given set of all peers ρ, it is defined as

$$DC = \frac{\sum_p D_p}{|\rho|}, p \in \rho$$

where D_p is the number of correctly discovered objects for peer p, and $|\rho|$ is total number of peers. That is, if one sees all objects it should see and none of objects it should not see, discovery consistency should then be 100%. In another example, if there are 15 objects in the world, 10 objects that should be discovered for a specific peer, and 8 objects are correctly discovered, 1 object is extra discovered, the discovery consistency is then is $((10-2)+(5-1))/15 = 80\%$.

Update consistency(UC) This indicator measures how well updates are delivered. DC measures the correctness of object discovery, but it is not enough to just correctly discover objects. Once an object is known, the next problem is how fast future updates can be learned after modifications are made. Since transmission latency cannot be avoided, our main concern becomes how much time is needed to deliver the proper updates. Thus, UC is defined as, for a peer p and objects in its knowledge O_p (we call the place storing all objects as the *object store*), update consistency at a future time step t, (UC_t) is defined as

$$UC_t = \sum_p \frac{NR_{O_p,t}}{N_{O_p}}$$

where $NR_{O_p,t}$ is the number of objects in store O_p whose states are newer or equal to the owners' historical states t steps earlier, N_{O_p} is number of objects in store O_p. The idea is to see how well the current states in the object store converge with the authoritative versions of the object states at some earlier time. Note that objects concerned here are only the correctly discovered ones, which is different from discovery consistency. For example, there has 5 objects I discovered correctly, versions of which are {22, 17, 11, 15, 23}, and at time t, versions of them are {18, 17, 12, 16, 23}. So UC_t for me is 3/5 (cause third and fourth object is older than

time t's version).

5.3 Simulation Results

First, we will show the consistency and scalability of VSM. Figure 10 shows the overall consistency of the system achieved by VSM. As described, we separate consistency into two different aspects: *discovery consistency* (DC) and *update consistency* (UC), UC-0 to 2 means how consistent the object stores are with the owners' states after some specified simulation steps. *Pos* or *State* indicate the consistency for position or normal object states, respectively. The figure shows that VSM can achieve discovery consistency near 100%, but a little inconsistency still may happen near the boundary of AOI. Since this problem may be solved by extending peers' AOI to add a buffer zone, we consider it a future improvement that can be trivially addressed. Update inconsistency is due to network transmission latency between the owner and replicas' states, the latency ideally varies from 1 (changed objects are managed by the same managing arbitrator as the receiving peers) to 2 (changed objects are owned by a neighboring arbitrator of the receiving peers' managing arbitrator) steps of end-to-end transmission latency.

Figure 11 shows overall bandwidth consumption of VSM separated by roles: arbitrators, aggregators, and gateway server. The numbers of bandwidth consumption indicate average values from all roles, all steps in the simulation, except for the gateway server which shows the total transmission size. Bandwidth consumed by peers and gateway server are ignorable. Arbitrators in VSM are usually run on normal peers, and the figure shows that their bandwidth usage is successfully controlled. Aggregators are chosen from capable peers or powerful servers, so it could accommodate heavier loads by taking over overloaded areas. Figure 12 shows the bandwidth comparison between VSM and traditional client-server model. In the figure, "Server send/recv" is the bandwidth consumption for the server in client-server, "-NA" indicates measurements with no aggregation. The result clearly shows that VSM, through distributing the management loads onto peers, is more affordable than client-server model. Average transmission in our client-server model uses 1.9 MB/s, but VSM uses only 12.7 KB/s (for arbitrators) and 50.0 KB/s (for aggregators) when the system has 500 nodes.

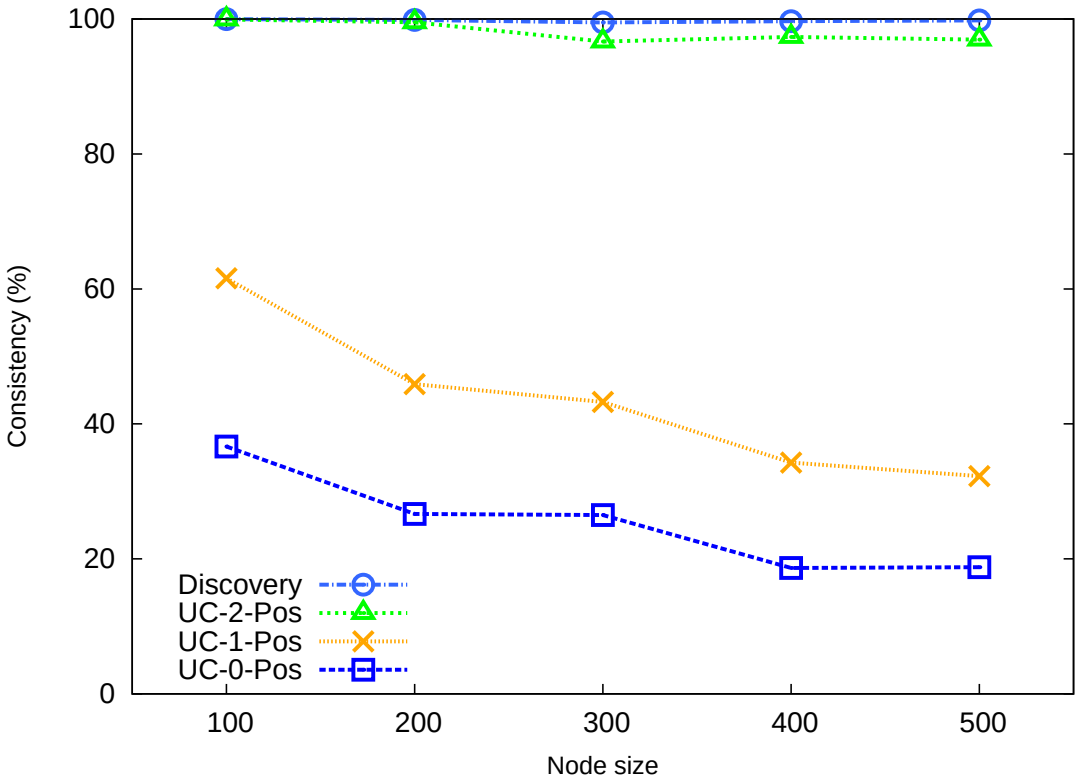

Figure 10: System discovery and update consistency

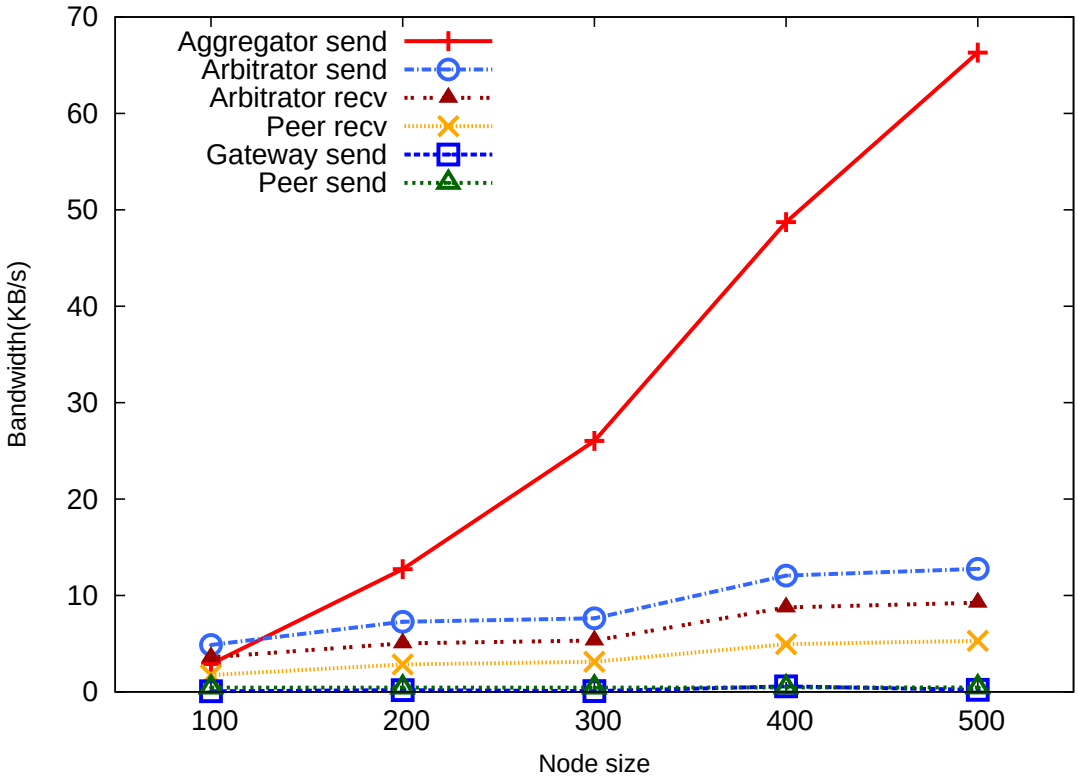

Figure 11: Overall bandwidth consumption

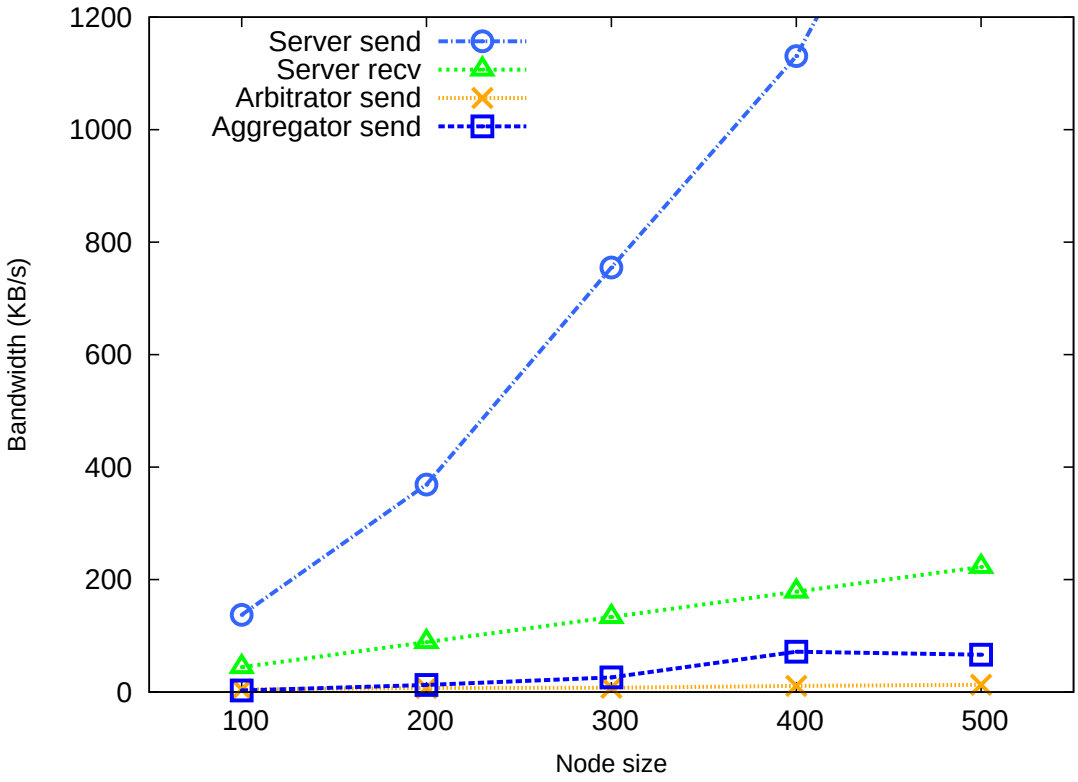

Figure 12: Bandwidth consumption comparison with client-server

Figure 13 shows the comparison of bandwidth consumption when aggregation is enabled or not. While arbitrators come from normal peers, heavy loads may incur unpredictable results. We can see that aggregation imposes a limit on bandwidth consumption for the arbitrators.

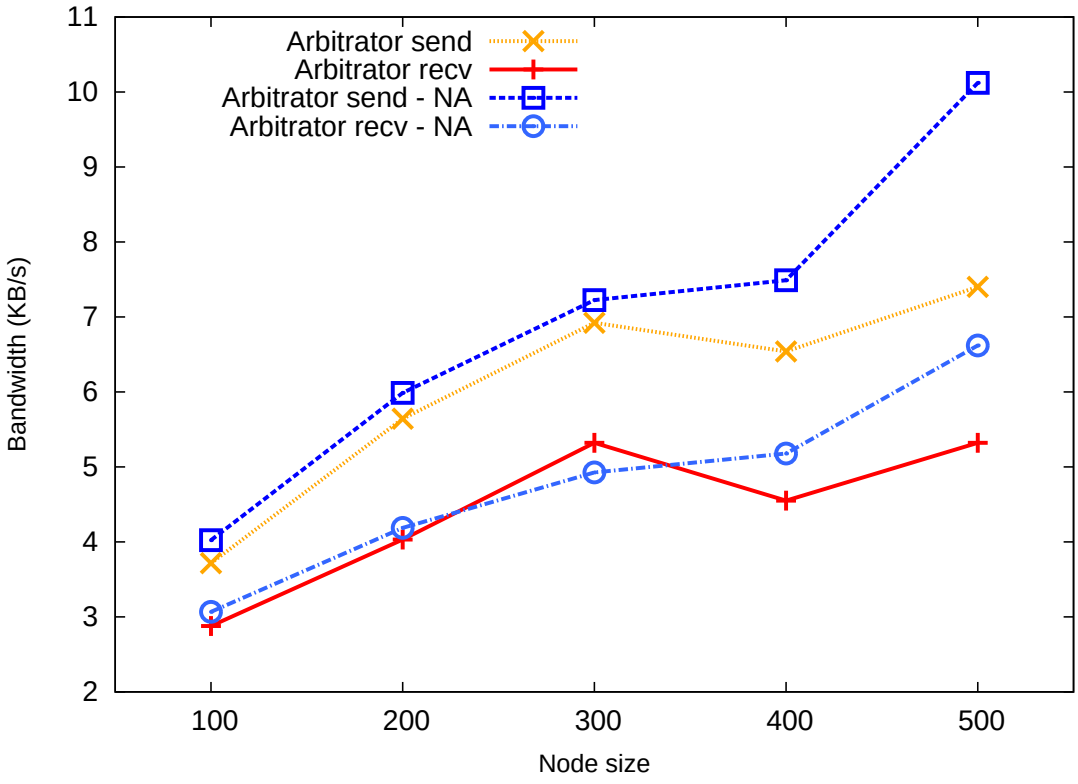

Figure 13: Aggregation bandwidth consumption comparison

6 Conclusion

In this thesis, we propose Voronoi State Management (VSM), a Peer-to-Peer-based state management scheme. By using a Voronoi diagram to divide the virtual world into regions, peers share the responsibility to manage the virtual world, and the load of managing states is spread out to all nodes who act as arbitrators of Voronoi regions. When the positions of avatars are clustered, load balancing is achieved by aggregating arbitrators to be taken over by aggregators, which are selected superpeers with better capabilities. VSM integrates Peer-to-Peer and Client-Server architecture into a single scheme, and can continually transform to one of them during the system operation. Simulation results show that VSM can keep the properties of consistency, scalability, and load balancing.

Although VSM can support basic operation of VEs, there still are a few more advanced topics that need to be considered, e.g., AOI scalability, fault tolerance, and some optimizations such as topology-aware routing/division. With an increasing number of nodes in a fixed area, transmission loads could increase exponentially. Once aggregators are overloaded, the system may crash or cause unrecoverable faults. Since exchanged information always increases with the node size, the way grouping and aggregating object/event/update is done now may give up some 'must know' information. VSM currently chooses aggregators in a random way, and may not select the most proper node for a given situation. A better solution may be to choose aggregators based on client capability and the physical topology, which may reduce message latency and achieve higher communication cost. To make VSM more applicable to real world scenarios, these problems need to be solved in the future.

References

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

[2] Second life. http://www.secondlife.com/.

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

[4] Gregor Schiele, Richard Suselbeck, Arno Wacker, Tonio Triebel, and Christian Becker. Consistency management for peer-to-peer-based massively multiuser virtual environments. In *MMVE '08: Proceedings of 1st International Workshop on Massively Multiuser Virtual Environments*, pages 14–18, March 2008.

[5] Richard M. Fujimoto. Parallel discrete event simulation. *Commun. ACM*, 33(10):30–53, 1990.

[6] Eric Cronin, Anthony R. Kurc, Burton Filstrup, and Sugih Jamin. An efficient synchronization mechanism for mirrored game architectures. *Multimedia Tools Appl.*, 23(1):7–30, 2004.

[7] Stefano Ferretti and Marco Roccetti. Fast delivery of game events with an optimistic synchronization mechanism in massive multiplayer online games. In *ACE '05: Proceedings of the 2005 ACM SIGCHI International Conference on Advances in computer entertainment technology*, pages 405–412, 2005.

[8] P. Morillo, S. Rueda, J.M. Orduna, and J. Duato. A latency-aware partitioning method for distributed virtual environment systems. *IEEE Transactions on Parallel and Distributed Systems*, 18(9):1215–1226, September 2007.

[9] D.T. Ahmed, S. Shirmohammadi, J.C. de Oliveira, and J. Bonney. Supporting large-scale networked virtual environments. *Virtual Environments, Human-Computer Interfaces and Measurement Systems, 2007. VECIMS 2007. IEEE Symposium on*, pages 150–154, 25-27 June 2007.

[10] David Kushner. Engineering everquest. *IEEE Spectrum*, 2005.

[11] Bjorn Knutsson et al. Peer-to-peer support for massively multiplayer games. In *INFOCOM 2004: Proceeding of 23th AnnualJoint Conference of the IEEE Computer and Communications Societies*, pages 96–107, March 2004.

[12] Joaquin Keller and Gwendal Simon. Solipsis: A massively multi-participant virtual world. In *PDPTA '03: Proceedings of International Conference on Parallel and Distribued Techniques and Applications*, volume 1, pages 262–268, 2003.

[13] D.A. Tran, K.A. Hua, and T. Do. Zigzag: an efficient peer-to-peer scheme for media streaming. In *INFOCOM 2003:Proceedings of Twenty-Second Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 2, pages 1283–1292 vol.2, April 2003.

[14] Shun-Yun Hu, Ting-Hao Huang, Shao-Chen Chang, Wei-Lun Sung, Jehn-Ruey Jiang, and Bing-Yu Chen. FLoD: A Framework for Peer-to-Peer 3D Streaming. In *to appear in INFOCOM 2008: The 27th Conference on Computer Communications*, April 2008.

[15] Skype. http://www.skype.com/.

[16] Bittorrent. http://www.bittorrent.com/, 2001-2005.

[17] J. Keller and G. Simon. Toward a peer-to-peer shared virtual reality. In *ICDCSW '02: Proceedings of 22th International Conference on Distributed Computing Systems Workshops*, pages 695–700, 2002.

[18] Ashwin Bharambe, Jeffrey Pang, and Srinivasan Seshan. Colyseus: a distributed architecture for online multiplayer games. In *NSDI'06: Proceedings of the 3rd conference on 3rd Symposium on Networked Systems Design & Implementation*, pages 12–12, 2006.

[19] Y. Kawahara, T. Aoyama, and H. Morikawa. A Peer-to-Peer Message Exchange Scheme for Large-Scale Networked Virtual Environments. *Telecommunication Systems*, 25(3):353–370, 2004.

[20] A.P. Yu and S.T. Vuong. MOPAR: a mobile peer-to-peer overlay architecture for interest management of massively multiplayer online games. In *NOSSDAV '05: Pro-*

ceedings of the international workshop on Network and operating systems support for digital audio and video, pages 99–104, 2005.

[21] Sunghwan Ihm Tcaesvk Gim Jinwon Lee, Hyonik Lee and Junehwa Song. APOLO: Ad-hoc Peer-to-Peer Overlay Network for Massively Multi-player Online Games. Technical report, Korea Advanced Institute of Science and Technology (KAIST), 2006.

[22] P. Morillo, W. Moncho, J.M. Orduna, and J. Duato. Providing Full Awareness to Distributed Virtual Environments Based on Peer-To-Peer Architectures. *Springer LNCS*, 4035:336–347, June 2006.

[23] Gregor Schiele, Richard Suselbeck, Arno Wacker, Jorg Hahner, Christian Becker, and Torben Weis. Requirements of peer-to-peer-based massively multiplayer online gaming. *CCGRID '07: Seventh IEEE International Symposium on Cluster Computing and the Grid*, pages 773–782, May 2007.

[24] Jui-Fa Chen, Wei-Chuan Lin, Tsu-Han Chen, and Shun-Yun Hu. A forwarding model for voronoi-based overlay network. *2007 International Conference on Parallel and Distributed Systems*, 2:1–7, Dec. 2007.

[25] Luther Chan, James Yong, Jiaqiang Bai, Ben Leong, and Raymond Tan. Hydra: a massively-multiplayer peer-to-peer architecture for the game developer. In *NetGames '07: Proceedings of the 6th ACM SIGCOMM workshop on Network and system support for games*, pages 37–42, 2007.

[26] D. Frey, J. Royan, R. Piegay, A.-M. Kermarrec, E. Anceaume, and F. Le Fessant. Solipsis: A decentralized architecture for virtual environments. In *MMVE '08: 1st International Workshop on Massively Multiuser Virtual Environments*, pages 29–33, March 2008.

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

[28] E. Buyukkaya and M. Abdallah. Data management in voronoi-based p2p gaming. In *Consumer Communications and Networking Conference, 2008. CCNC 2008. 5th IEEE*, pages 1050–1053, 2008.

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

[30] Paul Bettner and Mark Terrano. 1500 archers on a 28.8: Network programming in Age of Empires and beyond. Proc. GDC, 2001.

[31] Claudio E. Palazzi, Stefano Ferretti, Stefano Cacciaguerra, and Marco Roccetti. Interactivity-loss avoidance in event delivery synchronization for mirrored game architectures. *IEEE Transactions on Multimedia*, 8(4):874–879, August 2006.

[32] Marios Assiotis and Velin Tzanov. A distributed architecture for mmorpg. In *NetGames '06: Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*, page 4, 2006.

[33] Jehn-Ruey Jiang, Jiun-Shiang Chiou, and Shun-Yun Hu. Enhancing neighborship consistency for peer-to-peer distributed virtual environments. In *ICDCSW '07: Proceedings of 27th International Conference on Distributed Computing Systems Workshops*, pages 71–71, June 2007.

[34] Jehn-Ruey Jiang, Yu-Li Huang, and Shun-Yun Hu. Scalable AOI-Cast for Peer-to-Peer Networked Virtual Environments. In *CDS '08: Proceedings of 28th International Conference on Distributed Computing Systems Workshops (ICDCSW) Cooperative Distributed Systems*, Jun. 2008.

[35] Wikipedia: Voronoi diagram. http://en.wikipedia.org/wiki/Voronoi_diagram.

[36] Chi-Chang Chen and Cheng-Jong Lee. A dynamic load balancing model for the multi-server online game systems. In *HPC Asia, Poster*, 2004.

[37] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, 11:329–350, 2001.

[38] M. Castro, M.B. Jones, A.-M. Kermarrec, A. Rowstron, M. Theimer, H. Wang, and A. Wolman. An evaluation of scalable application-level multicast built using peer-to-peer overlays. In *INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications Societies. IEEE*, volume 2, pages 1510–1520 vol.2, 2003.

[39] Ashwin R. Bharambe, Mukesh Agrawal, and Srinivasan Seshan. Mercury: supporting scalable multi-attribute range queries. In *SIGCOMM '04: Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 353–366, 2004.

[40] Vast project. http://vast.sourceforge.net/.