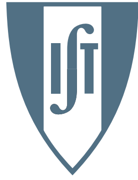


INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

VFC Large Scale: Consistency of Replicated Data in Large Scale Networks

André Filipe Pessoa Negrão

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática e de Computadores

Júri

Presidente:	Professor Doutor Nuno João Neves Mamede
Orientador:	Professor Doutor Paulo Jorge Pires Ferreira
Co-Orientador:	Professor Doutor Luís Manuel Antunes Veiga
Vogal:	Professor Doutor João António Madeiras Pereira

Novembro 2009

This work was partially funded by the FCT research grant PTDC/EIA/66589/2006 in the context of the Mercury project.

Agradecimentos

Em primeiro lugar gostaria de agradecer aos meus tios, Maria Eduarda e Sidónio, por todos os sacrifícios que fizeram por mim ao longo destes 20 anos que partilhámos. Obrigado por toda a compreensão, pela paciência inesgotável e pelo amor incondicional. Embora possa nem sempre parecer agradeço-vos por tudo e tenho imenso orgulho em ter sido (e ser!) vosso filho.

Quero também agradecer ao meu avô António por todo o apoio e pela constante e importante presença nestes anos de faculdade. Obrigado à minha irmã Patrícia por...bem, por ser a melhor irmã do mundo! Obrigado também ao resto dos meus familiares: aos Pessoa, aos Negrão e aos outros que não partilham comigo o apelido, mas que partilham a união e a amizade desta família espectacular.

Envio um agradecimento especial à Luciana por ter sido o meu grande suporte nos últimos dois anos. O teu apoio, a tua amizade, a tua paciência e o teu sorriso contagiante estão espalhados pelas páginas que compõem esta dissertação. Obrigado por tornares tudo muito mais fácil e divertido.

Aos meus colegas António Gorgulho, João Paiva, Diogo Paulo e Daniel Santos (e aos outros que também se cruzaram pelo meu caminho). Foi (e ainda é) um prazer partilhar esta aventura com vocês. Obrigado também àqueles que só apareceram no último ano: João Leitão, João Nuno e David Matos. Com vocês, o lado geek da força é muito mais divertido.

Um abraço também para o meu amigo Henrique Campos, o melhor afilhado que se pode desejar. Ao Gonçalo Rosmaninho pelas noites de degustação e discussão vitivinícola tão inspiradoras. Aos dois, obrigado por serem os melhores amigos que se pode ter. E ao profeta alfaiate Moisés por ser o profeta alfaiate Moisés. Ele sabe.

Obrigado também a todos aqueles a quem me esqueci de agradecer. Se me esqueci é porque devia ter-me lembrado!

Por fim, quero agradecer aos meus orientadores Paulo Ferreira e Luís Veiga por todo o apoio que me deram no decurso deste trabalho e pela disponibilidade total. Obrigado também por me terem iniciado no mundo da investigação científica e por me darem a oportunidade de trabalhar num projecto tão interessante.

Lisboa, 25 de Novembro de 2009
André Filipe Pessoa Negrão

Aos meus tios, Maria Eduarda e Sidónio.

Resumo

Ao longo da última década os jogos de computador multijogador jogados através da Internet sofreram um rápido crescimento de popularidade, ajudados pela expansão da largura de banda das ligações de rede e os avanços nas capacidades gráficas e de processamento dos computadores pessoais. A acompanhar este crescimento, também os jogadores se tornaram mais exigentes. Em particular, estes requerem alta performance, disponibilidade e escalabilidade de modo a maximizarem a sua experiência de jogo. Actualmente, os principais jogos comerciais suportam estes requisitos através da utilização de hardware de alto desempenho. Para conseguirem suportar o crescente número de utilizadores, estes jogos, tipicamente, confinam os utilizadores a uma única divisão do espaço virtual, impedindo-os de interagirem livremente com os restantes jogadores.

Neste trabalho propomos uma abordagem diferente em que, embora o espaço virtual também esteja dividido entre servidores, se permite que os jogadores mudem de divisão e interajam livremente sem se darem conta da topologia de suporte. Para melhorar o seu desempenho, o sistema aplica o modelo de gestão de interesse Vector-Field Consistency. Assim, consegue-se reduzir a largura de banda necessária para que cada participante represente o estado do jogo, sem, no entanto, prejudicar a consistência destes dados.

Para testar o nosso sistema desenhámos uma infraestrutura de simulação que permite simular diversos ambientes de jogo, incluindo diferentes tipos de clientes e modelos de gestão de interesse. Os resultados preliminares mostram que o sistema é capaz de suportar um elevado número de jogadores mais eficientemente do que outras arquitecturas alternativas.

Abstract

In the last decade Multiplayer Online Games experienced a fast increase in popularity, helped by the expansion of broadband Internet access and the advances in graphic cards and processing power. Accompanying this growth, the players of this games also became more demanding. In particular, game users require high performance, availability and scalability in order to maximize their gaming experience. Currently, most commercial games meet these requirements by confining users to isolated partitions provisioned by powerful server clusters.

In this work, we propose a different approach to virtual world partition in which players are allowed to freely interact and move around the game map. To improve performance, the system enforces the Vector-Field Consistency model. This way it is able to reduce the bandwidth requirements imposed on the participants of the game without injuring the consistency of game data.

To test our system we designed a simulation infrastructure that is able to simulate different system architectures and interest management models, as well as different types of game clients. The preliminary results show that our system is able to handle a large number of user in a more efficient manner than other alternative architectures.

Palavras Chave
Keywords

Palavras Chave

Jogos Multijogador pela Internet

Sistemas Distribuídos de Larga Escala

Consistência de Dados Replicados

Gestão de Interesse

Keywords

Multiplayer Online Games

Large Scale Distributed System

Replicated Data Consistency

Interest Management

Contents

1 Introduction **1**

1.1 Challenges . . . 2

1.2 Contributions . . . 4

1.3 Document Roadmap . . . 4

2 Related Work **7**

2.1 Massively Multiplayer Online Games . . . 7

2.2 System Architectures . . . 10

2.2.1 Centralized Client/Server . . . 10

2.2.2 Distributed Client/Server . . . 11

2.2.3 Peer-to-peer . . . 14

2.2.4 Discussion . . . 17

2.3 Replicated Data Consistency . . . 19

2.3.1 Pessimistic Replication . . . 20

2.3.2 Optimistic Replication . . . 21

2.4 Replication in MMOGs . . . 22

2.4.1 Interest Management . . . 23

2.5 Commercial & Academic Systems . . . 26

2.5.1 Commercial MMOGs . . . 26

2.5.2 Academic works . . . 28

2.6 Summary . . . 31

3 **Architecture** **33**

3.1 System Overview 33

3.2 The Vector-Field Consistency model 35

3.2.1 Consistency Zones & Consistency Vectors 35

3.2.2 Consistency Enforcement 37

3.3 System Architecture 38

3.3.1 Server Organization 39

3.3.2 Inter-server Communication 41

3.3.3 Client - Server Communication 44

3.4 Distributed Vector-Field Consistency 45

3.4.1 Update Processing 47

3.4.2 Updating Clients 47

3.5 Summary 48

4 **Implementation** **49**

4.1 Development Environment 49

4.2 Algorithms and Supporting Data Structures 49

4.2.1 Object and User Representation 49

4.2.2 Object Pool 50

4.2.3 Subscription Protocol 50

4.2.4 Distributed Vector-Field Consistency 51

4.2.5 Remote Interfaces 52

4.3 Game Programming Interface 52

4.4 Simulation Infrastructure 54

4.5 Summary 55

5 **Evaluation** **57**

5.1 Interest management evaluation 58

5.2 Architectural evaluation 62

5.2.1 Performance 63

5.2.2 Server To Client Bandwidth 64

5.2.3 Inter-server Communication 65

5.3 Summary 66

6 Conclusion **67**

6.1 Future Work 67

List of Figures

2.1 Example Centralized Client-Server network. Note that the server can be a single machine or a locally distributed system. . . . 11

2.2 Example Distributed Client-Server network. . . . 11

2.3 Replicated system with four servers and different types of objects (represented by the circles and square). Each server stores the exact same data. . . . 12

2.4 Partitioned representation of the same application state represented in Figure 2.3. . . . 13

2.5 Example P2P network with eight peers (P). . . . 15

2.6 Hybrid/Partial P2P network. . . . 16

2.7 Distributed System Architectures. . . . 19

2.8 Pessimistic replication: servers respond to client requests only after executing a synchronization protocol. . . . 21

2.9 Optimistic replication: synchronization is performed after responding to the request of the client. . . . 22

2.10 Avatars scattered across a virtual world with region based IM. . . . 24

2.11 Example of avatars and their respective auras. . . . 24

2.12 Consistency zones defined around a pivot. . . . 25

2.13 Sharding example: server one and two manage two independent copies of the same virtual world while server three is responsible for a different map. . . . 27

2.14 Geographic decomposition example: three servers managing different independent zones connected to each other via portals. . . . 28

3.1 VFCLS main concepts: Virtual world partition and Vector-Field Consistency. . . . 34

3.2 VFCLS architecture: Components and interactions. . . . 38

3.3 Example of a partitioned virtual world. . . . 39

3.4 Multiple region inter-partition interaction. . . . 41

3.5 Owned and subscribed objects. 46

4.1 VFCLS integration with game applications. 53

5.1 Client-side bandwidth requirements. 58

5.2 Client-side bandwidth requirements: results without the NoAOI configuration. 59

5.3 Client-side bandwidth requirements with 500 and 1000 players scattered across a 5000x5000 map. 59

5.4 Client-side bandwidth requirements: influence of player density. 60

5.5 AOI size and variations. 61

5.6 Client side bandwidth requirements: auras versus VFC. 62

5.7 Execution times of function round-trigger for the different architectures. 63

5.8 Execution times of function round-trigger for the different architectures: VFCLS and replicated architecture highlight. 64

5.9 Per-server bandwidth requirements of the different architectures. 65

List of Tables

2.1 Summary Table. 32

5.1 Description of the different parameters variations. 57

5.2 Different VFC configurations that yield the same bandwidth as Aura3. 62

5.3 Description of the evaluated architectures. 63

5.4 Server communication in replicated architectures. 65

5.5 Server communication in VFCLS. 66

Chapter 1

Introduction

In the last decade Multiplayer Online Games experienced a fast increase in popularity, helped by the expansion of broadband Internet access and the advances in graphic cards and processing power [40]. In few years, games evolved from one-time play, small environments to Massively Multiplayer Online Games (MMOG) - worldwide networks with thousands of interacting users and, ever more often, persistent game state [8, 37, 16].

Supporting this new form of game playing presents diverse and challenging issues. First of all, high performance levels are required in order to provide the highly interactive experience demanded by the players of the game. Second, these games must scale to an increasingly large number of active users and size of game environment. Third, it is fundamental to provide constant availability, so users can play whenever they want, for as long as they would like and with as few disruptions as possible.

To meet these requirements current commercial game companies deploy powerful and expensive centralized servers (or server clusters) provisioned with high bandwidth and computational power. To reduce the restrictions to scalability imposed by the limitations of the hardware and, thus, allow a larger number of users, a game's virtual world is either duplicated or statically partitioned into several mini worlds [31]. Scalability is achieved by assigning each duplicate/partition to a different, independent server.

Although widely used, this approach presents several obvious drawbacks. First, and foremost, with these strategies users are confined to a single server at each moment and are unable to see/interact with players in other servers. Partitioned schemes do allow users to move to other partitions, but force them to cross some form of artificial boundary (e.g., doors, portals, tunnels) specially designed for that purpose. Scalability is, therefore, achieved at the expense of user interactivity. Moreover, although multiple servers are employed, each server is essentially a centralized, completely (or almost completely) independent system.

Another capital disadvantage of static partitioning is that it requires programmers to consider the physical division of the map into zones when they design and develop the virtual world. They have to carefully plan that division and predict the amount of resources (e.g., CPU, memory, bandwidth) necessary to provision each partition. Prediction is a difficult and inexact task in which failure results in either unnecessary expenses due to idle resources or in low performance.

In this work we aim at designing, develop and measure a scalable, efficient and highly available network infrastructure to support Massively Multiplayer Online Games. The main goal is to design an

infrastructure that does not rely on the static partitioning schemes used by commercial games to achieve scalability. Instead, we intend to design an architecture in which the game is presented to its players as a single large virtual world, regardless of the underlying organization of the system. This way players can freely move around the game map and interact with each other, without further limitations than those imposed by the logics of the game.

Moreover, we intend to design an architecture based on the cooperation of multiple distributed nodes working together to provide the bandwidth and computational resources required to attain the desired performance and scalability, instead of doing so through the use of high performance expensive server clusters/grids. This increases the opportunities for smaller companies (that do not have the economic capacity to acquire high performance hardware) to build and launch their own multiplayer game.

A secondary goal of our work is to design a middleware that aids programmers in the task of designing multiplayer online games that use our designed infrastructure. Ideally, this middleware layer would take care of all the work regarding the distributed and network aspects of the game, allowing programmers to focus on the design of the logics and graphics of the game, interacting with the system only through a mechanism easy to use and understand.

1.1 Challenges

Designing this distributed architecture, however, presents some challenges that need to be addressed. The main decision consists in choosing on how to balance load among the distributed nodes, so that the system can adapt to the movement of players across the game world. Both processing and bandwidth must be efficiently balanced, otherwise it is impossible to match the performance of cluster-based architectures.

A major challenge regarding bandwidth is consistency management. To improve performance, game state data (e.g., maps, player's positions) is replicated on (i.e., copied to) the users' computers. Enforcing replica consistency requires intensive synchronization communication, which leads to strong cuts in performance due to the high latency of large scale networks. Furthermore, as the number of users and the size of the virtual world increases, the bandwidth required to manage replica consistency grows immensely.

This fact is particularly dramatic on the side of the users of the game, whose hardware is typically limited in both bandwidth and processing power, specially when compared to dedicated servers. If no precautions are taken the players' computers can easily become overloaded with large update messages that not only block incoming bandwidth, but also force the player side game application to process high amounts of (possibly useless) data. Hence, in order to effectively scale, a game's supporting infrastructure must apply a consistency model that minimizes the bandwidth and communication requirements imposed by consistency enforcement without degrading users' experience.

In the past years, several research/academic works have proposed both peer-to-peer(P2P) [19, 7, 25] and distributed client/server [5, 10, 4, 24, 14] alternatives to the commercial approach. In P2P game infrastructures the game management is handled by the players themselves who directly forward their update messages between each other. This approach offers computational power, bandwidth and adaptability without the need for dedicated servers.

However, although the overall processing capacity of a peer to peer network - composed of the combined computational power of each peer - grows as users join the game, the computational and bandwidth

resources of each user remain scarce when compared to dedicated servers. As the dimensions of the game increase, users - who, in P2P architectures, act both as players and servers - are required to manage (receive and process) increasingly large amounts of data. Because users' resources are limited, bottlenecks can arise and, therefore, peer to peer systems can not guarantee the demanded performance levels. Furthermore, because P2P approaches depend completely on game users, who may or may not be present, it does not guarantee the required availability.

Client/server (C/S) solutions, on the other hand, use multiple servers geographically distributed that cooperate in order to efficiently balance computational and network load. Unlike P2P, C/S architectures put the burden of game management on server machines dedicated exclusively to that task. Servers collaborate by replicating [24, 14] or partitioning [5, 10, 4] the game state among each other. In a replicated system, each server holds a complete copy of the virtual world, but manages only a subset of the total number of players. The division of players is performed with the goal of minimizing latency between players and their servers - players are assigned to the server that is geographically closer to him. Because in these systems each server holds a different copy of the same data set there must be a means to maintain a consistent view of the virtual world. This requires servers to synchronize with each other - according to some synchronization protocol - a task that may negatively impact performance in large scale highly interactive systems.

Partitioned systems, on the other hand, divide players by partitioning the virtual world into disjoint regions, each assigned to a different server. Load balancing is achieved because each server manages only those players located on their region. A key difference between partitioned and replicated schemes is the fact that in the former systems each server holds a different portion of the complete data set. Therefore, there is no need to perform server synchronization. The exception to this rule occurs when two (or more) interacting players are located on regions managed by different servers.

In these approaches, in spite of the logical division of players and regions among servers, from the perspective of the player there is a single virtual world. Both in replicated and partitioned systems servers constantly communicate and synchronize to guarantee that players located on different servers can freely interact. The division (either by replication or partition) is, therefore, completely unknown to the players, unlike what happens on commercial games.

Regardless of the architecture, most of these systems rely on Interest Management [29] to minimize bandwidth requirements both at the client and server side. IM is a technique that reduces the number of state update messages - and, as a result, bandwidth - that travel through the network by sending to clients only those updates that are within the players *area of interest* (AOI). Areas of interest can be, for instance, the user's line of sight visibility [17, 9] - forward only the updates that concern objects that are within the player's vision radius - or radius of visibility [5, 10] - send only the updates to objects that are within an area surrounding the player. Although these strategies considerably reduce bandwidth, they are too inflexible and follow an all-or-nothing approach that fails to capture the user's real interest: every update to objects within the AOI is forwarded to the player and none of those considered irrelevant - the ones outside the player's AOI - is. With these strategies players experience an abrupt loss of visibility and end up seeing objects suddenly appearing/disappearing on/from their area of interest. Furthermore, players may even see objects that are not actually in the position they see them.

A different approach to the one taken by common IM schemes was proposed by the Vector-Field Consistency (VFC) [35] model. Instead of distinguishing only between relevant and irrelevant updates, VFC defines degrees of relevancy in the form of *consistency zones* - concentric areas defined around special objects (called pivots), to which is assigned a radius and a consistency degree defining the importance of

the updates to objects that are inside that zone. VFC allows the definition of multiple consistency zones, with different radius and consistency degrees that are weakened as the distance to the pivot increases. Due to this fact, VFC is able to eliminate the abrupt visibility loss characteristic of other IM strategies. Instead, it gracefully degrades player view, in a similar manner to human visibility.

Besides their contributions to the area of interest management, the authors of VFC developed their work in the context of a middleware infrastructure (Mobihoc) designed with to help programmers to develop wireless multiplayer games for mobile phones. Because it was specifically conceived to these small, resource-constrained environments, Mobihoc employs a centralized architecture, where a single server enforces VFC to every player of the game. As such, it is not suitable for large scale environments.

Despite these problems, we believe that VFC and Mobihoc have great potential and, if correctly revised, can improve both the performance of MMOGs and the way these games are programmed. In this work we propose *Vector-Field Consistency Large Scale* (VFCLS), a distributed client/server architecture that uses as its interest management scheme a version of VFC extend to achieve improved scalability. In our system, the virtual world is partitioned into different (but not independent) regions each handled by a distinct server. Players can move around the virtual world, transparently switching between regions handled by different servers. They are also able to seamlessly interact with players located in other regions, limited only by the logics of the game and the consistency model defined using VFC. The server organization enforces VFC through a subscription protocol that allows servers to apply VFC for players located in other, contiguous or not, regions, thus effectively allowing inter-partition interaction. We also provide a game programming interface that allows game programmers to design multiplayer online games on top of our designed architecture.

To evaluate our system we designed a simulation infrastructure that allows us to simulate different architectural settings, Interest Management models and game clients. The preliminary evaluation results show that our system performs well when compared to other architectures and gives us indications of possible future work to improve some aspects of VFCLS.

1.2 Contributions

In the work described by this dissertation we developed the following contributions:

- Distributed multiplayer online game support infrastructure
- Distributed Vector-Field Consistency model
- Game Programming Interface
- Flexible simulator infrastructure
- Evaluation of the system

1.3 Document Roadmap

This document is organized as follows. In Chapter 2 we describe in detail the context of our work and the topics addressed by it. We also present a survey on commercial and academic works that share our

topics of interest. Chapter 3 describes the architecture of our system and Chapter 4 presents details about its implementation. In Chapter 5 we describe the evaluation environment and present the results obtained, analyzing them at the same time. The text finishes with Chapter 6, where we summarize the work performed, introduce our ideas for future work and draw some conclusions.

Chapter 2

Related Work

This Chapter overviews the main concepts and background about the topic at hands. The following section introduces some basic notions about multiplayer online games and further clarifies the context and requirements associated with our work. After that we discuss the two main topics of this work, system architectures and consistency management, clarifying, along the way their, importance. We finish the Chapter with a review of concrete works that focus on the same topics as ours, addressing both commercial and academic solutions.

2.1 Massively Multiplayer Online Games

In a massively multiplayer online game (MMOG), a large number of players interact through an extensive virtual world, shared over a wide area network. This virtual world is typically represented as a complex 3D environment, that tries to simulate real life or fantasy/science fiction scenarios and situations. Games have their own storyline, typically involving disputes between races of fictional characters over territorial or racial dominance.

In online games, players control an entity (the *avatar*) that represents them in the game's virtual world. Avatars can move across the game map and interact with each other, either cooperating or competing, according to the instructions given by the human player through some input device (e.g., a keyboard or a mouse). Players can also find several objects (e.g., health items, food, weapons) and computer controlled characters (NPC)[1] - with which they can also interact.

Each avatar has its own state that comprises several properties like position, health, abilities and owned items. Interactions with other avatars or objects may change both its state and the others'; for example, a player that drinks from an health potion from a bottle will gain health, while the content of the bottle decreases. Moreover, although only these two objects are directly involved in this interaction, other players must be notified about its results.

One main distinction between online games and most other commercial distributed systems is the frequency of read and write operations. In most distributed systems and application reads are the dominant operation - data is stored at some node of the network and users search, request and download

[1] "Non-player Characters"

it to their local storage. This is the case of, for instance, file sharing systems and regular web sites, where objects are mostly immutable. In distributed file systems writes are more frequent but reads are still more common. Even in e-commerce and social networks - like e-bay[2] and facebook[3], respectively - where users are required to input data in order to enjoy the full potentials of those systems, reads are still considerably more frequent.

Multiplayer online games, on the other hand, are write intensive distributed application. This distinguishing characteristic stems from the fact that, to improve performance, game state information (e.g., the virtual world itself, other players' avatars, computer controlled objects, scores) is replicated at each player's computer - i.e., it is copied to each player's computer and kept synchronized according to some consistency strategy. This way, players use only locally available information to perform graphic rendering and game logics processing.

Having replicated data means that there must be some mechanism to maintain consistency. For example, if a player attacks another player, the copy - in the context of replication it is called *secondary replica* or simply *replica*- of the attacker's avatar that is placed at the second player's computer must be updated with that action. Updated data can be achieved in two ways, either the node that holds the replica issues read requests to the node that stores the real version of the data - called *primary replica* - or the primary replica updates - either directly (P2P) or through a server (C/S) - its replicas. In multiplayer games, it is the latter approach that is used, because it yields more efficient network usage. If the first approach was to be used, clients would have to issue periodic read requests to all the primary copies of their replicated data. This would result in high network traffic, with much of it corresponding to unnecessary read requests to unmodified data. With the chosen approach, however, only relevant messages, containing actual updates, are propagated. However, because player actions are performed frequently, so is update sending, which results in the mentioned write intensity that characterizes multiplayer games.

User Requirements

The most important goal of any type of game is that it be fun [40]. Being fun means providing an experience that captivates users and makes them want to keep on playing. Users usually are attracted by rich, dynamic game stories with many features and complex/realistic 3-dimensional graphics. These requirements are imposed on a game, be it multiplayer or single player. In multiplayer online environments, however, users impose additional requirements, related to the interactive and distributed nature of these games:

- Performance: Single player games are limited by the capacity of the user's hardware to process game logic and render graphics. When playing online, however, network related aspects can become the local bottleneck (as will be explained in the following subsection). Despite that, users expect an online game to be efficient, as close to the performance of a single player game as possible. In particular, players expect to "see" the actions performed by others reflected on their view of the game under soft real time constraint - they can tolerate some delay, but only if it does not affect them negatively.

- Scalability: The main purpose of a multiplayer game is to allow interaction between several users. Users want to play with as many opponents/friends as possible. Moreover, they want to have

[2]www.e-bay.com
[3]www.facebook.com

the same levels of performance, regardless of the number of simultaneous players. Scalability is the most important requirement for an MMOG, because interaction in a large scale is what distinguishes them from the other game genres.

- Availability: A player expects to have the game available at any time. In single player mode, the game is available as long as the user has a computer with the game software installed. When playing online, however, users can only play if the game's network infrastructure is available. Therefore, a game infrastructure has to provide 24/7 support. Furthermore, it has to be fault tolerant, so users can play with as few disruption as possible.

Technical Challenges

Network communication places three main challenges to MMOGs [36]:

Latency Latency is the time necessary for a message to travel from one node to another. It has an high influence on the time it takes for an action performed by one player to be perceived by the other players. In large scale networks, latency is particularly high, due to the distance between nodes and routing processing. If it is too high, the delays can have a severe impact on the interactive feeling of the game. This impact, however, depends heavily on the game type. Research studies have shown that fast paced games, like FPS, can tolerate latencies around 100ms, while RPGs can accept up to 500ms of latency [6].

Bandwidth In MMOGs, players are constantly moving around the virtual world and interacting with each other and the environment. To ensure the required real time interactivity, messages corresponding to a player's action must be sent to other players within strict time constraints. As the system grows, the number and frequency of messages that arrive at and departure from a node increases immensely. Bandwidth - the maximum amount of data that a communication line can send (outbound) or receive (inbound) per unit of time - places a limit on the rate at which messages are exchanged and, therefore, on the scalability of the game. To ensure that a game is capable of growing gracefully it is necessary to reduce the bandwidth requirements. For that purpose, games use a technique called Interest Management (more details in Section 2.4).

Computation The large number of users imposes high processing requirements on the participants of the game. Besides the game logic and graphics computation, nodes must process the updates to the replicated objects received from other nodes and the messages to send to the network. As the number of users increases, so does the number of messages to send and receive and, consequently, the processing requirements. Furthermore, while interest management reduces the bandwidth required, it can considerably augment CPU usage.

The way these characteristics impact the game is highly dependent on the system architecture and the state management strategies, as will be explained in the following sections.

2.2 System Architectures

A system architecture defines how the nodes of the system are connected to each other, how they communicate and their responsibilities. Different architectures have distinct performance, scalability and availability properties. This section describes the most relevant architectural settings and how they apply to MMOGs.

There are two main classes of system architectures. The most common and historically most important is the client/server (C/S). C/S systems are composed of two distinct types of nodes, *clients* and *server*. A server is an application, usually running on an independent computer, dedicated to providing some kind of service by responding to requests issued by client applications. A service can be supplied by a single server or a server cluster (*centralized C/S*) or by multiple distributed servers (*distributed C/S*). A client is the application that interacts with the server on behalf and under the control of a user (like a game client that sends updates to a game server based on the data input by the game user).

The other main architectural class is peer-to-peer (P2P). P2P systems distinguishes from C/S by the fact that a service is provided through the direct cooperation of the nodes (called *peers*) of the system. In a pure P2P network, every node is equivalent, each performing the same tasks. In many cases, however, some peers play special roles and, in other, there may even be dedicated servers, used to assist in some tasks. Regardless of the actual topology, the premise is that, for most tasks, nodes can directly cooperate/communicate without needing a central intermediate.

The following sections take a closer look at these two architectural groups and their variants in a broader context than multiplayer games. We finalize with a discussion of their relative merits and drawbacks regarding multiplayer games.

2.2.1 Centralized Client/Server

Centralized architectures can be composed of one server or a cluster of servers. In the former, the service is provided by a single server, that responds to all the requests. It is the simplest architecture and the easiest to implement. However, it is not scalable, since the number of clients that a server can efficiently handle is limited. After such limit is exceeded, performance decreases dramatically; it can even lead the server to crash, rendering the service completely unavailable. Thus, single server systems cannot provide strong guarantees regarding scalability.

A centralized server cluster is an agglomerate composed of several servers distributed over a high speed local private network. For this reason they are referred to as *Locally Distributed Systems* [11]. Clusters can be formed by numerous computers/workstations and, thus, are able to provide high processing power.

Clusters are connected to the outside world by a router/switch that is responsible for balancing workload among the servers of the local the network. Hence, they can be seen as a powerful single server that achieves scalability due to its (possibly expensive) dedicated computational and network hardware. Hereafter, the term *server* will interchangeably refer to a single server or to a cluster of locally distributed servers.

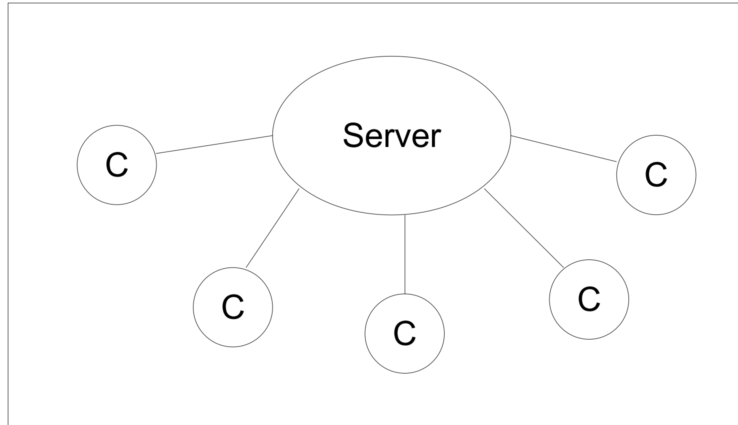


Figure 2.1: Example Centralized Client-Server network. Note that the server can be a single machine or a locally distributed system.

2.2.2 Distributed Client/Server

In a distributed client/server system, multiple interconnected and geographically distributed servers work together to divide computational and network load. Clients connect to servers based on load sharing rules and can switch servers according to those rules. Servers respond to clients' requests, possibly after communicating with each other. From the user's perspective, there is only one service, regardless of the number of servers used to provide it.

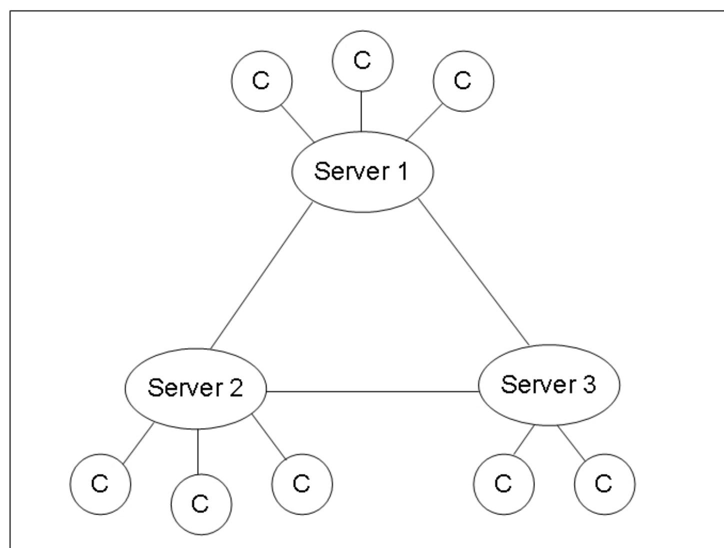


Figure 2.2: Example Distributed Client-Server network.

One major challenge that arises when designing a distributed C/S system is deciding on how to divide work between the servers. The generic goal of load balancing - i.e., the distribution of workload between the nodes of a distributed systems - is to improve the overall performance of the system. Performance, in this case, does not necessarily means efficiency; it can mean improved availability, fault tolerance or scalability, among other things.

Different load balancing strategies have different properties regarding these, and other, factors. Gener-

ically speaking, it is possible to identify to main superclasses of load balancing for client server architectures [15]:

- State replication: Each server has a copy of the same logical application state(e.g., files in a distributed file system (DFS), avatar and item objects in MMOGs, pages in a web site). Load is shared by distributing users between servers.
- State partition: The application state is partitioned into several distinct groups and each group is assigned to a different server.

State Replication

In a replicated system, there are several copies of the same logical data space. Clients' requests are serviced by any of the available servers. Deciding which server responds to a client can be based on diverse factors. The most straightforward form of sharing load between servers consists in sending requests randomly or in round robin to servers. Despite its simplicity, it is not an efficient solution, as it may leave the system unbalanced.

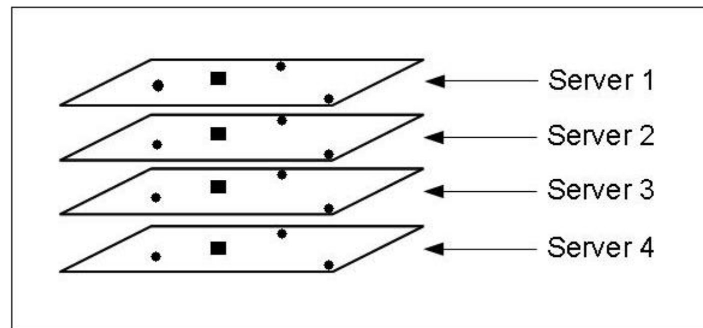


Figure 2.3: Replicated system with four servers and different types of objects (represented by the circles and square). Each server stores the exact same data.

More sophisticated approaches base client assignment on the state of the system in order to minimize the load on servers or the response time [12]:

- Server load: Clients are assigned according to server state metrics (e.g., computational load, available bandwidth) with the goal of minimizing the load in each server.
- Distance to users: Client assignment is based on proximity between clients and servers. Clients connect to the nearest available server in order to improve performance by minimizing message latency and, as a result, response times.
- Hybrid: This strategy considers both server state and proximity to try to balance both server load and latency in client assignment.

To avoid overloading on the servers, replicated systems can dynamically readjust the system. If a server gets overloaded during execution, clients are reassigned to other less busy servers. In general, the goal of rebalancing is to equally divide the load on servers. Alternative load sharing strategies can try to minimize both server load and latency (e.g., by redistributing clients to the nearest server that is not full) [30].

A major challenge in replicated systems is maintaining data consistency. Because several copies of the same application state are available on different locations, a change to a copy located at a server must be propagated to the other sites. Moreover, concurrent updates to different copies of the same logical data may result in inconsistencies (e.g., two users simultaneously change the same file on different servers of a DFS; two players grab the same health item in an MMOG).

There are two main techniques to achieve consistency in replicated systems: pessimistic and optimistic replication [18]. Pessimistic strategies prevent inconsistencies by forcing server synchronization as part of an update request. A response is sent back to the client only after synchronization is complete. As a result, response times increase and the performance of the system decreases. On the other hand, optimistic approaches reduce response times by performing synchronization only after answering to clients. Response time and performance are improved, but clients may end up seeing inconsistent data. (More details about this issue in Section 2.3).

State Partition

In these systems clients connect to the server that holds the information the user wishes to obtain. In a single request, they may have to contact several servers if the data they want is scattered across several distinct locations. For example, in an multiplayer online game the virtual world may be partitioned between various servers. While players explore the game world they may have to connect to several different servers.

In this approach, the primary issue revolves around the decision of how to partition the state of the application. A simple approach that evenly divides the application data between each server may lead to unbalanced load on certain servers if a portion of the state is more requested than others. When it is known, *a priori*, that some subsets of the data are more frequently requested than others it is possible to apply more efficient and elegant approaches. For instance, some servers can be statically assigned a larger set of less requested data, while others handle smaller subsets of the most requested information.

Alternatively, the most requested partitions may be provisioned, regardless of its size, by more powerful servers. *Geographic Decomposition*, a technique employed by some successful MMOGs [37], follows this later approach.

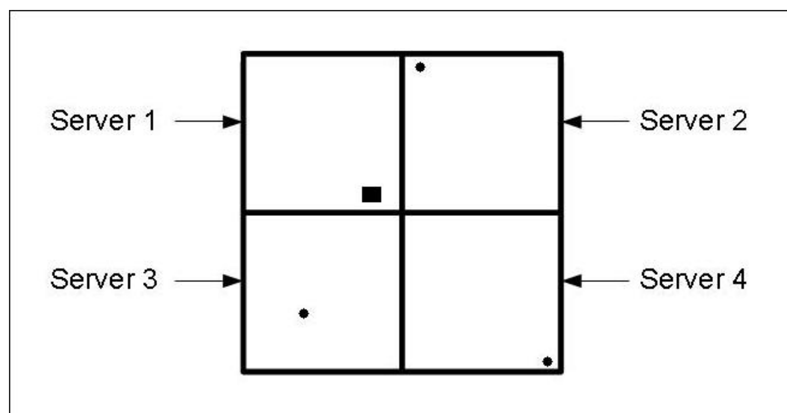


Figure 2.4: Partitioned representation of the same application state represented in Figure 2.3.

Unfortunately it is not always possible to predict which portions of the application state are more

accessed. Furthermore, in many cases user interest on data changes dynamically. If a static partition scheme is used under these conditions some servers may become overcrowded while others are idle most of the time. In these cases a more appropriate approach is to dynamically partition the state according to the load on servers. This way if a server gets overloaded, repartition can be employed to ease excessive load.

Since each server is responsible for a distinct set of the application state, in most applications, updates to a server do not influence the data managed by others. MMOGs are an exception to this rule. Consider, for example, two avatars, each located on the border of one of two different, adjacent regions of the virtual world. If one of the players shoots the other, that action is felt on the two regions and, therefore, on two different servers.

2.2.3 Peer-to-peer

Unlike client/server, peer-to-peer systems aim at providing services through the direct sharing of resources (e.g.,computation, bandwidth, content) between the nodes of the network [15] mostly or completely without the mediation of a central entity. In a P2P based multiplayer game like *Age of Empires*, for example, players exchange updates directly with no central server involved in the process. However, many P2P systems may have some degree of centralization. According to this aspect P2P systems can be classified as follows [3]:

- Purely Decentralized (Pure P2P): Every peer has the same functionalities. They can interchangeably act as clients and servers.
- Partially Centralized: Some peers (called *superpeers* or *supernodes*) play special roles. Promotion to superpeer is dynamic and any peer can become one.
- Hybrid Decentralized: Like in a partially centralized system, there are some special nodes. However, in these cases those nodes are not dynamically assigned peers, but dedicated and statically allocated servers.

Purely Decentralized

A purely decentralized system can be viewed as a flat hierarchy where every participant has the same responsibilities and can perform any task. Peers can act both as clients and servers of a given resource/service at the same time.

Depending on the existence or not of an underlying node organization these systems can be classified as *structured* and *unstructured*, respectively. In an unstructured P2P infrastructure there is no assurance about where resources (e.g., files in a file sharing system or avatars in an MMOG) and peers are. Each peer only knows a set of nodes, through which they try to find resources.

Flooding is a simple location mechanism in which peers send query messages to all their neighbors [3]. Neighbor peers, in turn, forward the query message to their peers, and so forth, until the resource is found or a given number of lookup iterations is completed. In spite of its simplicity flooding can lead to poor performance due to high number of messages that travel the network and have to be processed by each peer.

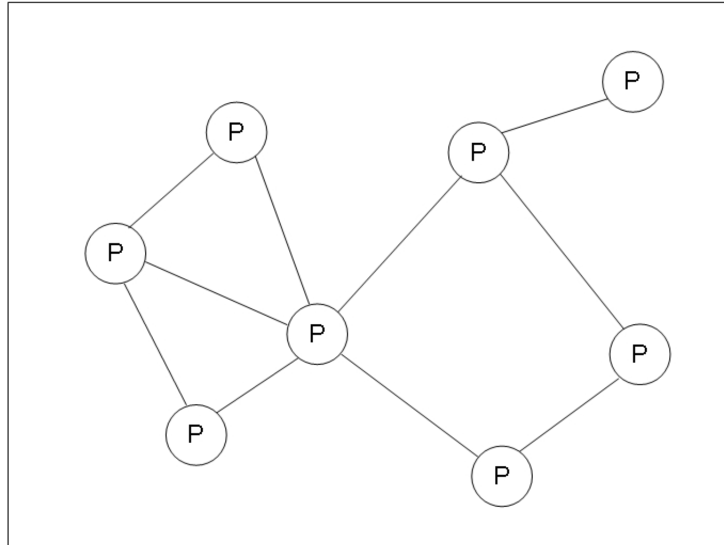


Figure 2.5: Example P2P network with eight peers (P).

In structured architectures resources are organized in such a way that peers deterministically know how to reach them. A common way of organizing the network is to use a distributed hash table (DHT) [38, 33]. Unstructured networks are highly self-organizing and scalable systems. However, because they have to maintain a certain structure, they may suffer from *churn* - the independent and constant arrival and departure of users [39]. Churn is problematic because when peers leave or enter the network, it has to adapt to that change by reconfiguring its structure. If the arrival/departure rate is high the constant reconfiguration prevents the network from maintaining its necessary structure, which incapacitates the system.

Partially Centralized

In partially centralized architectures, superpeers are employed to help other peers in some tasks. A typical role of a superpeer is to store information about the location of a small part of the data. This way peers can find data faster and more easily by directly consulting superpeers, instead of running a lookup protocol involving other nodes.

Although supernodes play special roles in a P2P network, they are still peers like the rest of the nodes and any peer can become a supernode. If a superpeer leaves the network or crashes the system automatically reorganizes and another one takes its place.

Hybrid Decentralized

Instead of having one of the peers playing a special role, hybrid decentralized systems use a dedicated server. Servers essentially play the same roles of a superpeer (e.g., resource location, routing). However, because they are managed by a central authority they can provide other services like authentication and authenticity. Furthermore, because they have dedicated bandwidth and CPU they can provide better performance than a superpeers.

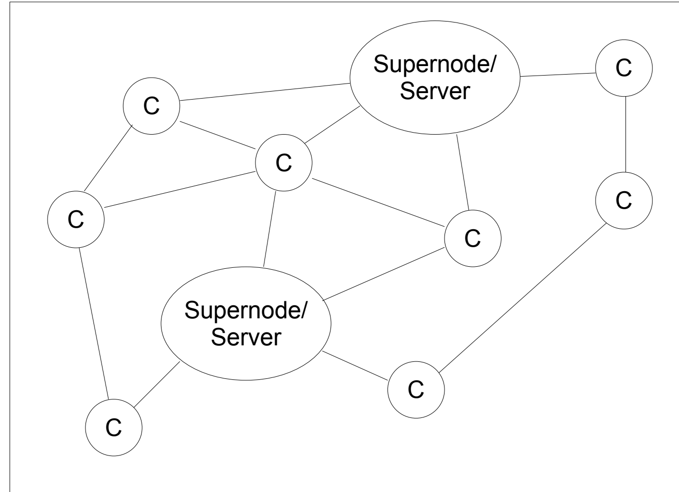


Figure 2.6: Hybrid/Partial P2P network.

P2P example: Content Addressable Network

An interesting example of a P2P infrastructure is the Content Addressable Network (CAN) [32]. CAN employs a purely decentralized scheme where peers cooperate to divide the responsibility for storing a set of objects. It is highly scalable, dynamic and self-organizing system.

In a CAN, each peer is responsible for a region of a logical virtual d-dimensional Cartesian space. Objects in the system are mapped onto a given position P in the virtual space according to a uniform hash function and are assigned to the peer responsible for the region where P lies. It should be noted that the virtual space is purely logical and is not related to any real physical coordinate system; objects in a CAN can be files in a distributed file system or in file sharing system and peers those systems' users. They could also be avatars and other objects of a multiplayer game, in which case the peers could be the players themselves or even the servers of the game.

In a CAN each peer knows only their direct neighbors (and possibly a small set of non neighbors). To find an object in a non neighboring region a node has to execute a lookup protocol that consists in sending a lookup message to the neighbor that is closer to the position of the desired object, who in turn forwards it to its neighbor that is closer to the object's position and so forth until the peer responsible for the object is found.

Self organization is achieved by redistributing object responsibility whenever a node leaves or a new one joins the network. When a new node wishes to join the CAN it chooses a random position P and issues a lookup to find the node responsible for that position. After finding that peer its zone is split in two with one half assigned to the new peer and the other to the old one.

If a node leaves it simply handles its region to one of its neighbors. CAN is also very resilient; when a node fails the others are able to detect the failure and recover from it.

A wide range of applications can be built on top of a CAN. The most striking example is a file sharing system. In a CAN based file sharing system the files to be shared would be mapped onto points and new the would be users assigned zones of the virtual space by splitting some pre-existing zone. To find a given file the system would only have to apply CAN's lookup protocol. Hence, every aspect of the system

is handled by the clients themselves. In a client-server system, on the other hand, files would be stored on dedicated servers and clients would simply send requests to a server which would then process the request and send a response back to the client.

2.2.4 Discussion

P2P Multiplayer Games

P2P support for multiplayer games has been an active research topic [19, 7, 25]. In these systems, clients exchange state updates directly with each other, instead of doing so through a server. In comparison to client/server, P2P networks are more able to scale up and down with the number of users. In a C/S architecture servers have limited bandwidth and capacity. In P2P, the overall system bandwidth and processing power comes and goes with the peers themselves.

P2P systems are also specially tolerant to failures. If a player leaves or crashes, only its state is lost; all other players remain active. If, however, the leaving node is a superpeer, the system must readapt, possibly at the expense of some overhead and interaction loss.

P2P raises some challenges not present or not common in client/server architectures, mainly data persistence and security. In C/S, servers are responsible for storing the complete state of the virtual world plus the players account information. Data is available as long as the server itself is. Furthermore, servers are trustworthy entities, thus, having server intermediating interaction is considered a guarantee of veracity.

As for completely decentralized P2P games, data persistence and availability is dependent on the availability of the user, which is not guaranteed. Moreover, because there is no central authoritative entity these systems are more vulnerable to security issues like cheating [21].

To solve or minimize the persistence and availability limitations of P2P game infrastructures, some works have proposed a hybrid P2P-Client/Server solution [41]. In these systems state management and update propagation is still handled by peers through direct connection, but servers are involved in the process to ensure security and persistent state.

With or without servers to help on specific tasks, in P2P systems, the burden of managing the virtual world and maintaining it consistent is put on peers that are executed on the players' machines. These machines are considerably limited in both bandwidth and computational power when compared to the dedicated servers used on C/S games. As the number of players of a game increases so does network traffic and the amount of data each peer needs to process both as a client (compute updates received, execute game logics, graphic rendering) and a server (message routing, consistency management). As a result, the performance of each peer degrades with the expansion of the game and with it, so does the overall performance of the network.

Another problem of supporting games with P2P networks is the aforementioned churn. In multiplayer games the overall number of users is large and their participation is highly dynamic - players are constantly entering and leaving the game. This characteristic makes it very difficult for a structured P2P network to be able to maintain the structure it needs to correctly handle the game.

These limitations are further aggravated in the face of player flocking/hot spots. If many players are involved in a group interaction, bandwidth and computation at the peers may become too high. As more

peers enter the group, more resources are required at each peer. Because, as we mentioned, peers have limited resources hot spots may incapacitate the system. Because of all these reasons, we believe that a multiserver client/server solution is still more suitable for large scale multiplayer games.

Partitioned vs Replicated Client/Server

In general, replication provides better availability than state partition, because it grants access to the complete data set even when some nodes are unavailable - if a server crashes or a network partition occurs, there are still other replicas containing the required application state. In an MMOG this means that the virtual world is available as long as there is at least one server.

A server crash on a partitioned system, on the other hand, leaves the data for which that server was responsible inaccessible. This problem can be reduced, however, by storing read-only backups of partitions on other nodes (this is, in fact, a simple form of replication). This way, if a server crashes, another one can take its place.

Regarding scalability and performance, the difference between replicated and partitioned systems is highly dependent on the characteristics of the application. State replication exploits the fact that most distributed applications are read intensive. In this case the need for server synchronization is usually small which results in lower response times and an increase in performance.

However, as we explained earlier, MMOGs are write intensive applications. State updates are issued by many clients and at a high rate. As result coordination is frequently necessary. If a pessimistic approach is chosen, response times increase and performance lowers. If optimistic schemes are used users may see stale data. In the event of conflicting updates (e.g., two players grab the same health item), users may even see some action being rolled back due to conflict resolution, which has a great negative impact on playability.

Besides this tradeoff between consistency and performance, this strategy also suffers from low scalability. Because servers keep a complete copy of the virtual world, each has to maintain every object that inhabits it. Since in MMOGs both the size of the environment and the number of users is large, processing and storage at each server may become a bottleneck. Moreover, as the size of the system grows, the number of servers involved in state synchronization increases, resulting in high inter-server communication. As a result, the necessary bandwidth and computational resources may be prohibitively high.

In addition to these drawbacks, the premise that this scheme provides lower latency is somewhat misleading. Although the latency between a client and its designated server is, in fact, minimized, *action latency* - the time it takes for an action to be perceived by other players - is likely to be larger, as it may need an extra hop between servers.

Considering this analysis, we believe that state partition seems more appropriate than state replication regarding scalability. Because each server only has a region of the entire virtual world most updates only need to be propagated to clients whose avatars are located in that zone. Even when inter-server communication is necessary (e.g., when two players on the border of two regions interact) it is cheaper because it involves a smaller number of servers.

Another expected advantage of the partitioned approach is its lower conflict probability. Because in replicated systems each server manages objects that can be scattered across the whole virtual world and

interact with objects located on any other server. This way, any update to an object of a given server can result in conflicts whose resolution requires synchronization involving every server of the network. On a partitioned system, on the other hand, each server has sole authority over a different region of the virtual world. This way, most updates affect only objects located on the same server that received them and, thus, conflicts are detected and resolved locally by it. Even if an update that affects objects located on different regions results in a conflict, its resolution have a minor impact as it only involves a small amount of server - the ones responsible for those regions.

Partitioned systems have another important advantage regarding scalability. In replicated systems, adding a new user affects every server of the network. With a partitioned virtual world a server manages only a subset of the users and adding a new user affects only one or a few servers.

In spite of its advantages, virtual world partitioning raises some intricate challenges not present on replicated systems. The first one is player handoff, i.e, the movement (and consequent transfer) of a player between regions. When a player moves to another region he is, in fact, switching to another server. It may happen that the receiving server does not have any representation of the player's state. In that case, a state transfer between server is required which may lead to delays.

Another complicated scenario occurs when a region of the game becomes more popular than others. When this situation (known as *player flocking*) happens, the server responsible for that zone (*hot spot*) of the virtual world can get overloaded. To overcome this problem either more resources are added to provision that zone or a dynamic repartition of the state is applied [13, 26].

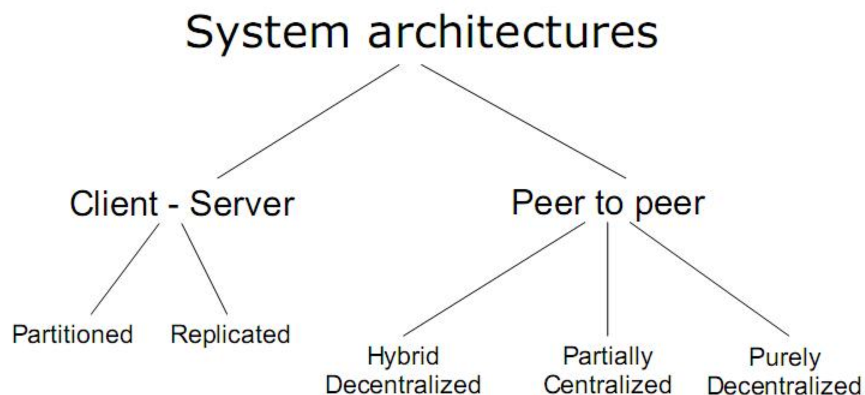


Figure 2.7: Distributed System Architectures.

2.3 Replicated Data Consistency

Data replication consists in placing and maintaining multiple copies (*replicas*) of a set of logical objects on different locations called *sites*. A logical object can be a file, a database record or a Java object and each has a set of physical instances called *replicas*. Each replica is stored at a single, distinct *site* and held by a *Replica Manager*(RM) to which clients can submit read and write operations (other operations can easily be derived from these) [15]. Replication is an important technique that allows applications to improve performance and availability.

Performance Because there are several locations from which clients can access data, they may choose the nearest one, thus improving response times due to the reduced latencies achieved. By replicating data at the user's computer (caching) clients can avoid accessing the network. Furthermore, with multiple distributed access points to the data, different clients can simultaneously access different copies of the same logical object. This increases the degree of concurrency of the system, enabling it to respond to an higher overall number of users.

Availability The increase in availability results from the possibility of accessing data even when some of the replicas are not available. If a node fails, clients can still access data by sending requests to one of the remaining available replicas.

Despite the potential advantages, replication brings along the problematic issue of replica consistency. Because several copies are available for concurrent access, clients may simultaneously try to update different replicas of the same logical data item. Therefore, there must be a mechanism for synchronizing replicas; otherwise, inconsistencies will arise. Without synchronization a client updating a data item at a replica could see his update being rolled back if he later accessed a different replica.

Replica synchronization is achieved by employing a replication protocol. A replication protocol defines what and under which circumstances operations are allowed and how replicas communicate in order to synchronize. These protocols can be classified according to the way updates are propagated to the remaining replicas and to where (at which replica) a logical object can be updated [18]. Regarding the latter, they can either be single or multi master. In a single master system, each object has a master node. Every update to an object can only be issued to its master, while the remaining replicas are used for read only operations. Therefore, there is no real update concurrency and every replica receive updates in the same order: the order defined by the master replica.

Multi master (or *group*) systems, on the other hand, allow real concurrency by admitting updates to be issued to any replica. This means that conflicts may arise and the system must have a mechanism to detect and resolve them, possibly by rolling back some operation performed at a given replica.

As for the way updates are propagated, a replication protocol can be classified as pessimistic or optimistic. We discuss these two alternative in the following sections.

2.3.1 Pessimistic Replication

Pessimistic protocols try to provide the user the illusion of having a single copy of the data, by ensuring that operations always leave the system in a consistent state [34]. To enforce strong consistency, when a client submits an operation, the replicas execute a synchronous coordination and agreement protocol before sending a response back to the client [42].

Maintaining replicas strictly consistent can be achieved in many ways. Primary Copy [2] defines one replica as the primary copy. Updates to a logic object are handled by the the primary and are propagated to the remaining secondary replicas as part of a single transaction, while reads require acquiring locks. In voting systems [27] operations are issued to all replicas and are allowed only if a quorum number of replicas accept it.

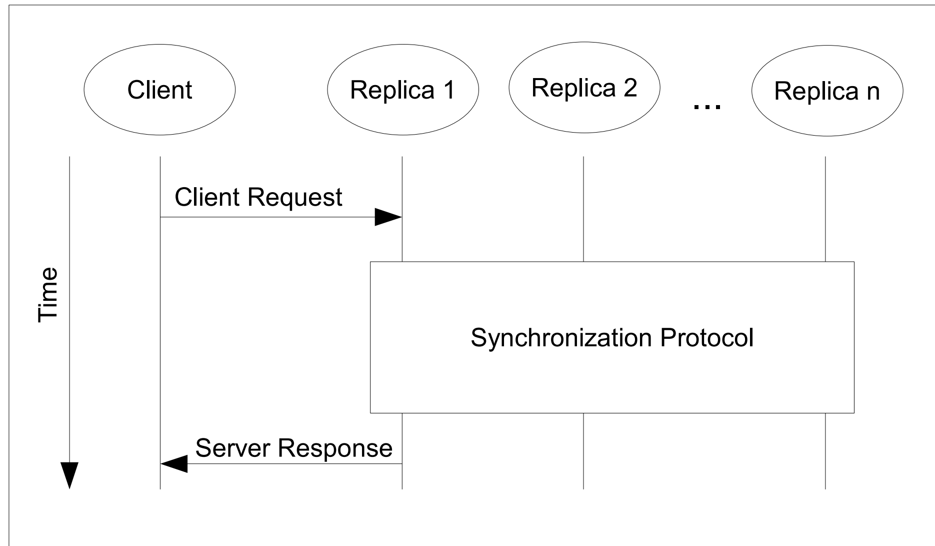


Figure 2.8: Pessimistic replication: servers respond to client requests only after executing a synchronization protocol.

Although pessimistic protocols provide high consistency, they require extensive synchronization, forcing clients to wait for coordination between replicas. In a large scale network, where latency is high, response time increases considerably, resulting in strong cuts in performance. Scalability is also an issue, since as the system grows so does the number of nodes involved in the complex synchronization process.

2.3.2 Optimistic Replication

In optimistic replication, contrary to its pessimistic counterpart, server synchronization occurs only after responding to the client [34]. When an operation is submitted to a replica it is applied to the local copy and a response is sent to the client based solely on the current local information the replica manager holds. Only then the update is propagated, asynchronously, to the remaining replicas, but not necessarily immediately. Therefore, the content of replicas can diverge for some time. Furthermore, because concurrent updates are allowed, conflicts may arise that must be detected and resolved.

The key advantage of optimistic replication is that it does not require *a priori* synchronization. This represents a substantial gain in availability, since it allows operations to be handled even in the presence of unavailable nodes. It also increases operation concurrency and reduces response time, thus, improving performance. These advantages, however, come at the expense of replica consistency.

Bounded Divergence

The simplest form of optimistic replication ensures eventual consistency (EC). Informally, EC means that if clients stop submitting operations, all replicas will, eventually, reach a consistent state. This means that there is no guarantee about when the system converges. It also means that the current state is not exact and may be rolled back (due to conflicts) to an indefinite moment in the past.

Bounded divergence protocols, on the other hand, let replicas' content to diverge but limit the maximum amount of deviation possible. Instead of allowing content to diverge indefinitely, they define under

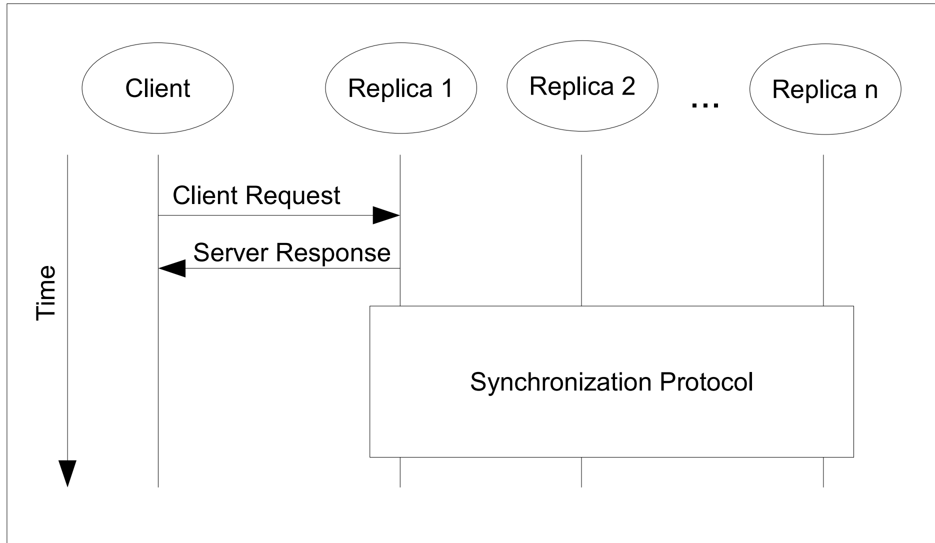


Figure 2.9: Optimistic replication: synchronization is performed after responding to the request of the client.

which conditions replicas are required to converge and how to enforce that convergence. This way, the amount of inconsistencies is limited and there is some certainty about the real state of the system.

There are various different techniques for limiting inconsistencies. The simplest form is to bound the amount of time a replica can be inconsistent (*Real time guarantees*) [1]. Another approach to control divergence limits the number of updates that can be applied to a replica without synchronization [22].

TACT is a more sophisticated protocol that bounds inconsistencies in three dimension, *Numerical Error*, *Order Error* and *Staleness* [43]. Staleness corresponds to the Real time guarantees technique previously mentioned. Order Error limits the number of updates that can be applied to a local error before synchronization. Numerical Error bounds the number of updates that can be applied to all replicas without being propagated to a given replica.

Interest Management (IM) is another form of bounded divergence [29]. Broadly speaking, IM consists in propagating an update only if it is considered relevant to the receiver. With interest management, applications can greatly reduce bandwidth by avoiding network access unless if strictly necessary. This results in a potential increase in performance and scalability. However, because message filtering is required to verify its relevance, the improvement is limited due to the additional computation. Nevertheless, applications that use IM are willing to pay the price for the increase in computational requirements, as CPU speed grows faster than bandwidth.

2.4 Replication in MMOGs

In the previous section we introduced and detailed some important concepts about replication and consistency management. In this section we will discuss these two topics in the context of multiplayer online games.

In MMOGs, maintaining the player's view consistent is crucial. If a player performs an action, other

players expect to receive the corresponding update under strict time constraints. Enforcing this real time like consistency is extremely hard, even more in large scale environment with thousands of users continuously interacting.

In multiplayer online games, the virtual world and its object are replicated, at least, on the computer of the user. It can also be replicated on several servers if a distributed client/server with state replication is used. This way, when an object performs an action the corresponding update must be propagated to the remaining replicas in order to achieve consistency.

As seen before, consistency can be achieved through pessimistic or optimistic strategies. For multiplayer games, pessimistic replication is not an option. If it was to be used, a player intending to move would have to wait for replicas to coordinate. In MMOGs there are numerous replicas spread across a large scale unreliable, high latency network. Obviously, this would prevent games from achieving the real time performance requirements.

Fortunately, MMOGs user and developers are willing to trade some consistency for performance and, therefore, opt for optimistic replication. Eventual consistency, however, is also not an option, because games require updates to be propagated on a timely manner. Allowing the value of a replica to remain boundlessly stale would break down game logics, resulting in user dissatisfaction. For instance, a player could not see the avatar of another player moving towards him if the corresponding replica was allowed to be out of date unlimitedly. For these reasons, MMOGs provide time constrained weak consistency, by updating replicas periodically. Update periodicity is typically of few dozens of milliseconds.

2.4.1 Interest Management

When analyzing replication in MMOG another important issue is the bandwidth required to propagate a virtual world to clients. A naive implementation that periodically forwards each update to every client would require a prohibitive amount of bandwidth, both for senders and receivers. To reduce these high bandwidth requirements without damaging the players' view of the virtual world, games use relaxed consistency models that try to identify and send to a player only those updates that he is interested in. In multiplayer games, the specification and enforcement of these consistency models is called Interest Management.

IM is motivated by the observation that players are not equally interested in all objects (e.g., other players, items). Usually, they are more concerned about objects located nearer them and as the proximity to objects decreases, so does the player's interest in them. This observation is typically materialized by two main different strategies: region and aura based Interest Management [28]:

Region

In region based IM the game world is partitioned in static, contiguous *consistency regions*. Objects in the same region receive updates from each other, but not from objects in outer regions (in practice, updates from objects in adjacent regions may also be visible). Region based is the most straightforward approach. Because regions are static, its easier to compute membership. However, its performance is highly dependent on the size of regions. If regions are too big, players end up receiving too many irrelevant messages; if they are too small its the other way around - relevant interactions may be lost. Defining the

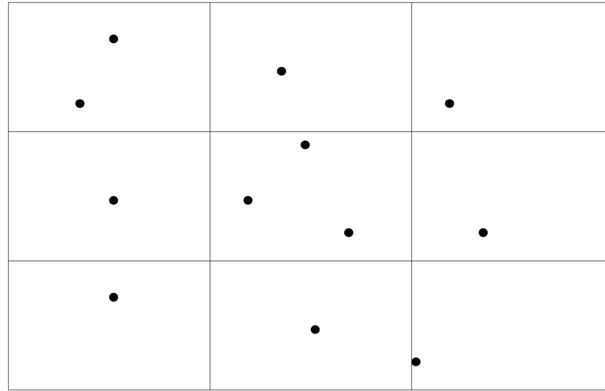


Figure 2.10: Avatars scattered across a virtual world with region based IM.

right size is not trivial and an optimal choice is typically not possible. Furthermore, it does not accurately capture players interest.

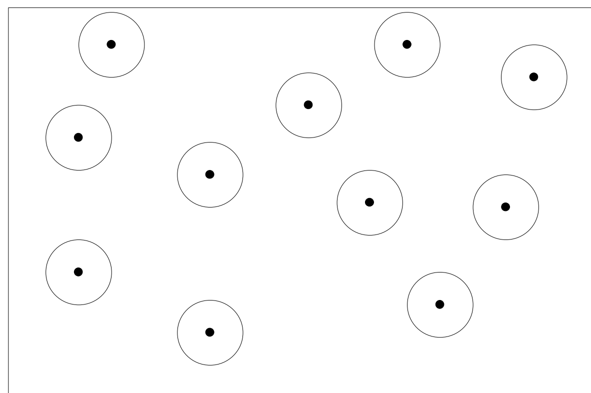


Figure 2.11: Example of avatars and their respective auras.

Aura

An aura is a consistency zone (usually concentric) defined around (and centered on) a player's avatar. When the auras of two avatars intersect they can see each other. When an object (other than an avatar) is within the aura of his avatar, the player can see it. These characteristics are used to achieve a considerable cut on the amount of data each player receives from the server and, thus, on the maximum inbound bandwidth required at each client.

Auras are also an important help to the scalability of the game, because as the overall number of players in the game grows, the amount of objects inside each players aura is expected to grow at slower pace. In comparison to region based AOI, auras allow more accurate message filtering and are more closely related to a players perception ability in the context of the game. On the other hand they require more complex computation to check which objects are on a given player aura .

Analysis

All these interest management strategies follow an optimistic replication approach. Objects within a player's area of interest are, optimistically, up to date, while the ones outside that area are simply ignored. Strong consistency on the inside is achieved by periodically (few milliseconds [7]) updating replicas, instead of simply forwarding updates as received. Thus, this scheme can be seen as optimistic replication with real time bounds.

Despite providing bandwidth reduction, current IM mechanisms are too rigid - updates in an area of interest (region or aura) are visible, all others are not. This inflexibility can lead to some undesired game situations. Consider, for instance, a player moving in an open scenario. Because consistency outside the player's area of interest (AOI) is not maintained, if an object enters it, the player may see it appearing out of nowhere. A worst situation occurs when an object leaves the AOI. In that case the object will still be considered as being on the AOI because its updates are not received unless it re-enters the area.

To solve these problems, games end up defining AOIs larger that actually needed [24], resulting in a cut in performance due to increased bandwidth and computation requirements.

Vector-Field consistency

Vector-Field consistency (VFC) [35] is a consistency model that adds greater flexibility to games eliminating, at the same time, the previous problems. VFC is an aura based scheme that introduces two main novelties to interest management. First, it introduces the notion of multiple auras, each called a *consistency zone*. A consistency zone is an aura to which is assigned a consistency degree. The further an aura reaches, the weaker its consistency degree is. Figure 2.12 shows an example of an avatar surrounded by its consistency zones.

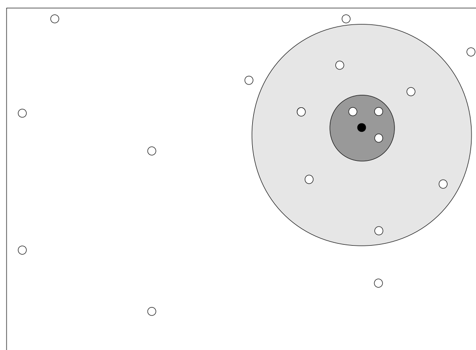


Figure 2.12: Consistency zones defined around a pivot.

The second innovation of VFC is the definition of consistency degrees. While in the other existent IM schemes, consistency degrees are based simply on a time metric (periodic updates), VFC introduces a multi criterion consistency model similar to TACT. Besides time delay, it also considers the number of lost updates (*sequence*) and an application specific function that captures the difference between replicas (*value*).

VFC permits a finer grained, more effective, albeit more complex, interest management. Furthermore it relies on the notion of *pivots* - special objects to which consistency zones are assigned. Pivots allow

even greater flexibility, as they can simulate more than one IM strategies and extend them with the multi zone and multi metrics of VFC. For instance, if the pivot is the player's avatar (as shown in Figure 2.12) VFC simulates a simple aura based scheme. By defining more pivots, other than the player's avatar, different schemes can be reproduced.

2.5 Commercial & Academic Systems

2.5.1 Commercial MMOGs

The standard approach in commercial games is to use several geographically dispersed servers to handle game state and intermediate user interaction. Having multiple servers means that there must be a way to divide the work among them. Currently, two main techniques are used, *sharding* and *geographic decomposition* (GD) [40]. In both cases, users must explicitly choose in which server they wish to play, or to start playing.

Because commercial games are proprietary there is little or no information about game design. In particular, information regarding interest management is virtually null and so, we will not address that topic in the discussion of commercial games that follows.

Sharding

Generically, sharding consists in assigning independent virtual worlds to different servers. Typically, there are several distinct small virtual worlds (*maps*) and multiple copies of them can be handled by different servers [8, 20].

When sharding is employed, users connect to one of several available servers, usually based on the distance to the server, to minimize message latency. Because each server hosts a different copy of the virtual world users can only interact with players located in the same shard. Hence, shards are basically several distinct centralized games that happen to have the same name.

This simple sharding strategy is employed by *World of Warcraft* (WOW). WOW employs a shard-based client/server architecture in which multiple servers (or server clusters) are deployed, each hosting an independent copy (*realm*) of the same virtual world. Realms are spread across the world and players manually connect to one of the available. Users are encouraged to connect to the closest server to achieve better connection times. Other successful MMOGs like and Ultima Online [16] also follow this strategy.

An alternative sharding strategy is used by First Person Shooter (FPS) games like Quake [20]. In these games, different servers can host copies of several different virtual worlds (*maps*). Like in simple sharding, each map copy is isolated from the remaining copies.

An example of this situation is illustrated in Figure 2.13, where the game is managed by three servers. Servers one and two hold two independent copies of the same game map; players on server one interact only with other players of that server, and the same goes for players on servers two and three. Hence, each server essentially hosts an independent game that can share (although it does not have to) the same map with other servers.

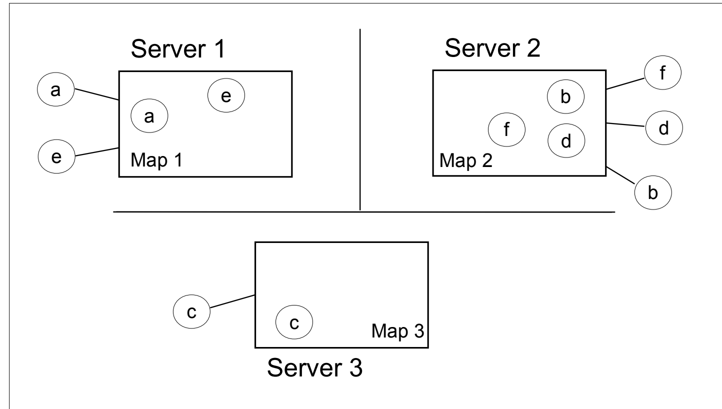


Figure 2.13: Sharding example: server one and two manage two independent copies of the same virtual world while server three is responsible for a different map.

Geographic decomposition

In this approach the virtual world is partitioned into several distinct and independent regions (e.g., island, countries, continents), each handled by a different server. Users connect to the server that is responsible for the region they wish to play in. Like in sharding, they can only interact with other players connected to the same server (and, hence, are in the same region). However, they can move to another partition, by crossing some artificial boundary especially designed for that purpose.

Geographic decomposition is applied in popular games like Everquest [37]. In Everquest the virtual world is divided into more than 4000 connected zones. Moving to another zones ("zoning") can take between several seconds and few minutes. Everquest is managed by a server farm (cluster) of more 1500 servers with each zone handled by a cluster of up to 30 servers, depending on prediction results [23].

Figure 2.14 an example of Geographic Decomposition in which we can see three servers each managing a different, independent zone of the virtual world. Players can move from zone one to zone two (and vice-versa) and from zone one to zone three (and vice-versa) through some form of portal (represented by the dotted lines linking those zones), but when located in one zone can only interact with players of the same zone.

Discussion

Both sharding and geographic decomposition are straightforward and easy to implement. Because servers hold independent instances of a game, they are basically centralized applications. In fact, with sharding, games are programmed as centralized applications that can be independently deployed on several machines. Even in geographic decomposition each region is almost completely independent. Communication between servers exists, but is rare and simple and happens only when a player crosses some of the few and strategically placed region boundaries.

The main problem with these strategies is that they achieve scalability at the expense of player interactivity. In both approaches, clients are isolated on a single server. Although they allow the overall number of users to grow with little or no impact on performance, they place a static limit on the number of users that can interact simultaneously.

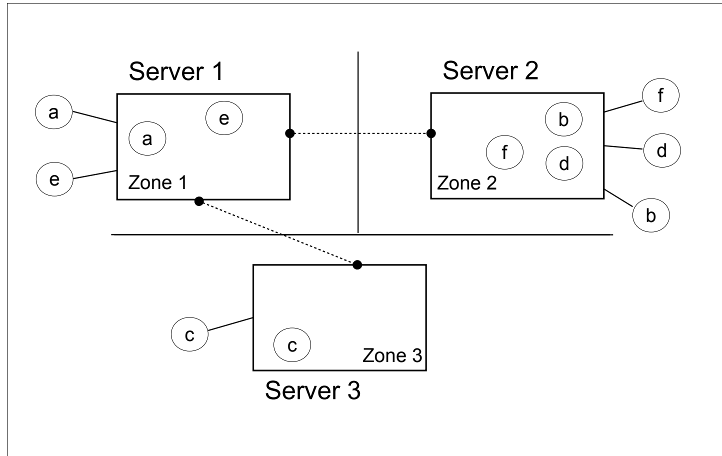


Figure 2.14: Geographic decomposition example: three servers managing different independent zones connected to each other via portals.

Geographic decomposition does allow players to move to other regions so they can interact with the players located on it. However, after moving to another region they will still be isolated on it. Furthermore, crossing a barrier is not transparent to the user. He has to wait until its state is transferred to the new server and is usually presented with a "loading menu". In sharding, this problem is even worse. Users can play in other maps, but they have to manually choose a server to play. Moreover, in both cases, if the server is overcrowded, user will be denied entrance.

Another major problem with GD is the burden on the programmers of the game. GD forces programmers to design the game world with partitioning in mind, instead of doing so solely based on the game semantics. Furthermore, it requires programmers to predict the resources (e.g.,number of servers, CPUs, memory) necessary to provision each region. Failing to predict resource usage results either in idle resources and wasted money or in poor performance.

2.5.2 Academic works

P2P Games

SimMud is a partially centralized P2P game [25]. It employs region based IM by splitting the virtual world into fixed-size regions. Peers inside a region are arranged in a multicast group and receive updates only from objects within the same region. Each region has a coordinator superpeer that intermediates access to shared objects (e.g., health items, potions) to avoid conflicts.

Although this early attempt was seminal to the development of P2P solutions it presents some drawbacks. The most obvious is that as the system grows the coordinator becomes a bottleneck, which is worsened by the fact that the coordinator is not a machine with dedicated bandwidth and computational power. Furthermore, as was explained earlier, region based IM does not truly capture players interest.

Colyseus is another example of a P2P game infrastructure [7]. Unlike SimMud, however, Colyseus is a fully decentralized structured P2P system in which no peer plays a special role. Each peer acts both as a server and a client of the game, performing the same tasks as the other.

Because in fully decentralized systems there is no central repository of the data, the peers of the Colyseus systems have to work on their own to find the objects they are or may be interested in. To do so the system uses a publish/subscribe strategy in which peers periodically publish their position to the network.

Also periodically, peers send subscribe messages to the network containing the description of their area of interest. To avoid flooding publications and subscriptions are routed to a *rendezvous* node based on the positions contained in those messages. When a publications matches a subscription, the rendezvous node informs the subscribing node who then replicates the subscribed object at his local data pool. From then on the subscribing node and the node that holds the subscribed object communicate directly. To minimize the latency of peer discovery - which can lead to lost interactions and missed objects - peers prefetch data that they predict will be, eventually, necessary.

The operation of Colyseus is greatly dependent on rendezvous points. However, which and how nodes are chosen to be rendezvous points and how publication and subscription messages are routed to those nodes is not specified. Furthermore, rendezvous nodes have a similar, albeit simples, task as coordinators, performing extra computation regarding the publish-subscribe system and, as such, are sources for possible bottlenecks. However, because rendezvous point definition and organization is not clear, it is not possible to fully understand the impact of this on the network.

Although the system is able to maintain each peer's subscription related outgoing bandwidth, results show that the overall traffic of the network is even larger than the overall traffic of client server networks. This means that each peer has to route and process increasingly larger amounts of data, which gradually degrades the performance of the network and, thus, its scalability.

Besides rendezvous points, another aspect of the system that is clearly defined is area of interest management. In Colyseus, subscription messages carry a description of the peer's area of interest which is based on prediction. However, what is an are of interest and how it is defined and managed is not explained.

Replicated game servers

Some academic works have proposed using replication as a means to share load between servers of a distributed game application. In this scheme (commonly referred to as "mirrored server architecture") each server holds a copy of the complete game world, but is only responsible for a subset of the players.

The main goal of this strategy is to reduce the response time of player updates, thus, increasing interactivity. To achieve this goal, clients are assigned to the server that is geographically closer to the client. Despite this real world allocation, players avatars can be located anywhere in the virtual map, regardless of their servers.

As explained earlier in this document, the main issue regarding replicated systems is server synchronization. In literature it is possible to find several proposed solutions to this challenge, with either pessimistic or optimistic solutions.

As previously discussed, pessimistic synchronization is not suitable for fast paced large scale games. To make it further clear we show, as an example, lockstep synchronization [17], arguably the simplest approach to conservative synchronization. In lockstep, each server waits until every participant has

received the exact same information and only then executes its computation (like checking if a player is allowed to move to a given position) and moves on to the next round, repeating these steps in a loop.

While it completely prevents inconsistencies, lockstep is obviously not fit for highly interactive games; if applied in a typical action game it would force each player to wait for all the others to submit their updates and only then (after his server computes its clients' updates) would he receive a response from the server.

Highly interactive multiplayer games' users, however, demand lower response times from servers, which typically is achieved through optimistic synchronization - servers apply updates when received and synchronize later in the background, detecting and resolving inconsistencies. Trailing state synchronization (TSS) [14] is an example of the optimistic take on server synchronization.

In TSS each server has a *leading state* storing the current game state and one or more *trailing states* each with an associated time delay relative to the time of the leading state. How these are used to detect and resolve conflicts is easier to understand by means of an example. Consider a server with a leading state and a trailing state with 100ms of delay. At time 225ms the server receives from another server i an update originally submitted to i at time 200ms (with the 25ms difference being caused by the optimistic synchronization delay). It applies the update immediately to its leading state, but only applies it to the trailing state at time 300ms (the time at which the update was originally submitted - 200ms - plus the 100ms of delay). When it does so it detects that the update does not exist at time 200ms on the leading state and, thus, a conflict was detected and the leading state is rolled back to the last know inconsistency free state (stored somewhere on the trailing state) and every action is re-executed.

Although it reduces the time taken by a server to respond to a client, TSS resolves conflicts by rolling back the state of the game. While this may not be problematic in games with fewer objects and players, as games grow, so does the conflict rate and, as a result, the frequency of rollbacks. Furthermore, TSS considers every out of order event as a conflict which may not always be the true.

Seamless zoned virtual worlds

Several research works have proposed distributed C/S architectures where the virtual world is partitioned into different zones/regions, each assigned to a different server. Unlike the similar geographic decomposition approach, this strategy aims at providing a virtual world where the player is unaware of the underlying state partition (hence, the name "seamless zoned virtual worlds"). Instead, players can transparently move across several regions.

Assiotis et al [4] proposed one such architecture. Their work addresses border interaction - interaction between players located near the border of different partitions -, player handoff and state repartition to overcome hot spots. They also use aura based interest management, although not much is said about this feature.

To maintain consistency when border interaction occurs, their solution uses a lock based synchronization mechanism. When a server receives an update for an action that may affect more than one region it acquires locks for those regions and, possibly, individual locks for the objects located on each of those regions. Every other action performed on any of the locked region or by any of the locked objects is forced to wait on a queue until the locks are released.

Player handoff is also handled using locks. When an avatar moves towards the border its server locks

the avatar and the destination region and proceeds to transfer the user to the server responsible for that region; while a player is being transferred, the locks are acquired and, thus, updates on that region also have to wait on a queue.

To reduce the load on a server whose region became a hot spot, when a server gets too busy its regions is split in two. One of the resulting halves is assigned to the server that originated the split while the other is transferred to another existing server or to a new one. Repartitioning also uses locks to prevent access to data that may be inconsistent due to the split process.

Although this work addresses the most important issues related to non static partitioning schemes, its lock strategy is neither efficient nor scalable and has the obvious disadvantage of preventing interaction while a region/object is locked. Even if locks are acquired for a small amount of time there will still be delays that the user can perceive. Furthermore, while partitioning reduces server load it increases border interaction which, in this case, is particularly negative because it increases lock related delays.

Cai et al [10] proposed a different approach to dynamic partitioned schemes that does not require locks. In their work the virtual world is statically partitioned in equally sized regions, each assigned to a different server. To allow border interaction, each server has an update and a subscription region. A server's update region is the one he is assigned to and the objects on that region are managed exclusively by that server. Its subscription region is the area surrounding its border, and thus, comprises a small part of its neighbors partition. When an object is on a subscribed region, its updates are propagated to the server/servers that subscribed to it.

This work also addresses player handoff. When a server receives a position update from a client, it checks the position of the player's avatar. If he has moved out of the servers region, the server starts a state transfer with the server responsible for the region where the avatar moved to. After that, the client is notified about the switch, so he starts communicating with the new server.

Although this solution does not require locks its approach to player transfer is rather limiting. Issuing state transfers only when the position of a player's avatar switches to a different region leads to delays that are directly proportional to the size of the data that needs to be transferred and make partitioning not completely transparent for the user.

Furthermore, their approach to partitioning is static - regions are defined beforehand and remain unchanged throughout the execution of the game. This makes the system not only vulnerable to hot spots, but also incapable of supporting the progressive growth of the system and the addition of new content.

2.6 Summary

In this Chapter we discussed the most important research topics related to our work. We started by providing some background on multiplayer online games, mentioning its goals, the requirements imposed by game players and the unique challenges these types of games present due to their networked and interactive nature. Next, we generically discussed the topic of system architectures in distributed systems, focusing on peer to peer and client/server architectures. This subject is of major importance in multiplayer games, as the organization of nodes in a distributed system greatly (and in different ways) affects its

scalability, performance and availability properties. At the end of that discussion we analyzed the merits and demerits of the application of each of the discussed architectures on the context of multiplayer games.

After the discussion about the architectural aspects of games we provided a generic overview on the topic of data replication and consistency management after which we inserted a section discussing the important area of interest management. We finalized the Chapter with a presentation and discussion of concrete examples of multiplayer game infrastructures, covering both commercial and academic systems.

Table 2.1 summarizes the systems discussed in this section, regarding the main topics covered by this work: Interest Management and System Architectures.

System	**Architecture**	**Load Balancing/ Node Organization**	**Interest Management**	**Notes**
Mobihoc	Centralized Client/Server	Single Server	VFC	Wireless Mobile Games
Cai et al.	Distributed Client/Server	Partitioned	Aura	
Assiotis et al.	Distributed Client/Server	Partitioned	Aura	
RING	Distributed Client/Server	Replicated	Line-of-Sight Visibility	
TSS	Distributed Client/Server	Replicated	Undefined	Synchronization Only
SimMud	P2P	Partially Centralized	Regions	
Colyseus	P2P	Purely Decentralized	Any	
World of Warcraft	Locally Distributed Server Cluster	Sharding	Unknwon	Commercial Game
EverQuest	Locally Distributed Server Cluster	Geographic Decomposition	Unknwon	Commercial Game

Table 2.1: Summary Table.

Chapter 3

Architecture

In this Chapter we describe in detail the architecture of our system. We start with an overview on the overall system architecture and its main characteristics. Then we describe the Vector-Field Consistency model in detail, highlighting our modifications to the original design. We then proceed to explain the details of our architecture, describing its most important components and how they interact to perform the main features of our system. Finally, we present our distributed version of the Vector-Field Consistency model.

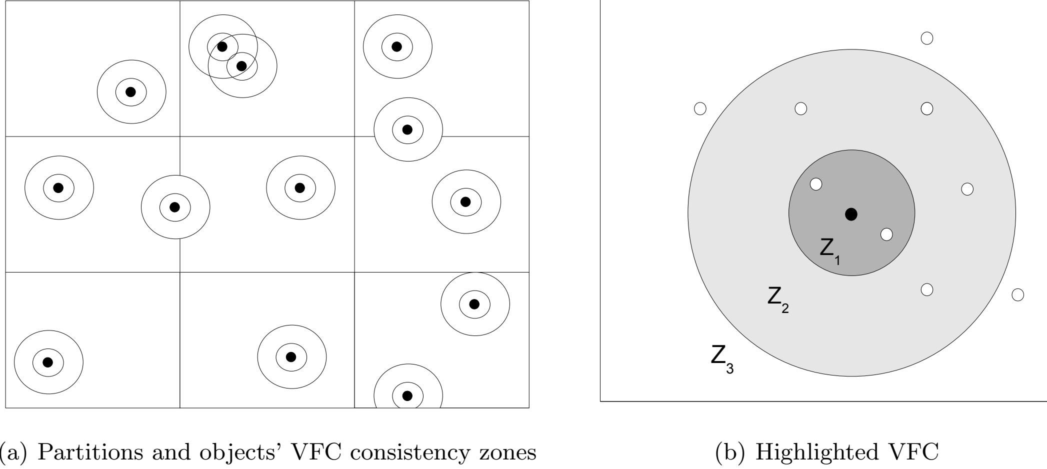
3.1 System Overview

In our solution we propose a distributed Client/Server architecture with state partition that provides a seamless view of the virtual world to its users, allowing players to interact with one another and move freely across the game map. We use Vector-Field Consistency (VFC) [35] as our interest management strategy to reduce the bandwidth requirements imposed on both the users and the servers of the game. The version of VFC applied by our system is an extension to the original model, designed to improve its scalability and suitability for large scale environments. For this reason, we named our system as "Vector-Field Consistency Large Scale" (VFCLS).

Our approach to achieve scalability consists in partitioning the virtual world into dynamically sized rectangular regions/partitions[1] and assign each one to a different server of a distributed server network. Clients (players) are assigned to one of those servers (the client's "designated server") based on the position of their avatars in the game's virtual world. Although the game map is divided in disjoint regions handled by different servers, the user is oblivious to that fact. From his perspective there is a single large shared virtual world he can freely wander and explore and where he is able to interact with several other players and different game objects. Our system allows players to move to different partitions and interact with players located on regions managed by other servers regardless of the underlying virtual world partition/organization, unlike the static partition approach employed by commercial games. His only limitations are the ones imposed by the logics of the game and the consistency model defined by the programmers (which is, itself, related to the logics of the game).

Figure 3.1(a) presents an example of a game scenario illustrating both the partitions of a game's

[1] Hereafter we refer to regions and partitions interchangeably



(a) Partitions and objects' VFC consistency zones

(b) Highlighted VFC

Figure 3.1: VFCLS main concepts: Virtual world partition and Vector-Field Consistency.

virtual world and a small number of avatar surrounded by their VFC consistency zones. For simplicity the image represents regions/partitions with the same size; however, in a real game context different regions may have dimensions/areas different from the ones represented in the image and from each other. The number and radius of the avatars' consistency zones can also differ from the ones represented in the image.

Servers in our infrastructure are organized in a peer to peer network similar to a Content Addressable Network (CAN) [32] (see Section 2.2.3 for more information). Like peers in a CAN, each server is connected to the servers responsible for their direct neighbor partitions. Unlike CAN, however, servers may also need to know and communicate with servers that are not their direct neighbors. This is necessary due to the fact that a player's area of interest (AOI) (defined by a game programmer using VFC) may be arbitrarily large and, thus, the player can potentially interact with objects located at non-neighboring partitions; to enforce VFC, the player's server needs information about those objects and, as a result, needs to communicate with the servers responsible for the regions where they are positioned.

To minimize the impact of the server synchronization required to perform inter-partition interaction we designed a server subscription protocol that guarantees that each server only knows about the non-neighbor servers it actually needs to be aware of, i.e., those whose partition may be crossed by one of its players' AOI. This way, each server knows only a subset of the complete server set - it requires having only a *partial view* of the network, which favors scalability. Furthermore, the protocol also ensures that an update from a player is received only by the servers whose objects may be affected by the update.

This approach to server organization allows the system to adapt to its own growth and servers to be added to the network with a limited impact on the overall organization. Only servers near the region where new servers are added are affected by it. This way, our system is able to gracefully scale as the system grows - it allows servers to be added, with minimal impact, to the network with the purpose of helping overloaded servers or simply to improve the overall performance of the system. Furthermore, due to the partitioning approach, a server only has to be aware of and manage a subset of the total number of objects and users of a game, as opposed to what happens in pure replicated systems where each server must be aware of all the objects of the game and apply interest management considering all of them.

Another advantage of partitioned systems is that each server has sole authority for its region and, therefore, can check and prevent conflicts for the objects located on it. Inter-server conflicts only arise as

a result of the interaction between objects located on two different regions, which is expected to be less frequent than interaction between players located on the same region.

In our previous analysis of P2P systems (see Section 2.2.4) we stated that they have some properties that limit their scalability when applied to online games. Now, it is worth mentioning why those problems do not arise in the context of our system. We identified two main constraints to scalability in P2P online games: peers' limited resources and churn.

Regarding resources, in our system each server is an entity exclusively dedicated to perform server tasks. This is unlike peers on a traditional P2P game who have to perform both server tasks, client tasks and whose computer may be running other non game related programs. Obviously, even dedicated machines have resource limitations; however, the fact that they are dedicated and special purpose machines means not only that those resources are likely to be more powerful than the average user's computer, but also (and more importantly) that its resource are better used. Furthermore, if a server gets overloaded, he can shed some of its load to a lightly loaded server or even be replaced by a new, more powerful one.

As for churn, because the P2P server network in our system is composed of dedicated machines it is a highly stable system. Modifications to the infrastructure occur only when a server crashes or is added/removed/replaced, all of which are much less frequent and more stable operations than players entering and leaving the game. Moreover, the number of servers is much smaller than the number of players, which further reduces the probability of a change in the server network.

3.2 The Vector-Field Consistency model

Vector-Field Consistency is a consistency (and Interest Management) model initially designed to manage consistency of replicated data in wireless multiplayer games running on resource-constrained machines. Its goal, as it happens with the other IM models, is to reduce bandwidth requirements (both at the client and the server side). However, unlike the other IM schemes, VFC allows the specification of multiple *consistency zones* defined around *pivot* objects (e.g, the player's avatar), each with different consistency requirements that decrease as the distance to the *pivot* increases. In the following sections we describe the Vector-Field Consistency model in detail.

3.2.1 Consistency Zones & Consistency Vectors

VFC represents the virtual world as an N-dimensional space populated with game objects (e.g., players' avatars, food objects, computer controlled entities (NPCs)). Each client node (e.g., a player's game client) of VFC has a local *view* of this virtual world that can have bounded inconsistencies with relation to their primary replica, which is located at the node (or nodes) that is (are) in charge of enforcing VFC. In each view, the consistency of objects depends on their distance to a *pivot* object. A pivot can be, for example, a player's avatar or a radar, while objects would be other players' avatars.

Pivots are associated with *consistency zones* - concentric, ring shaped areas defined around the *pivot* - that define its AOI. Each *consistency zone* has a *consistency degree* that specifies the consistency requirements of the objects located within that zone, regarding the *pivot*. As the distance to the *pivot* increases the consistency degrees become weaker. This way, a *pivot* generates a discrete "consistency field", analogous to the continuous gravitational and electric fields. As a result of this model, an object

located closer to the *pivot* is required to be more often refreshed. Although in the definition of the model consistency zones are ring shaped areas, they are implemented as concentric squares to improve the performance of the computationally expensive operation of determining if an object is within a radial surface.

Figure 3.1(b) illustrates an example of a *pivot* and its corresponding consistency zones z_1, z_2 and z_3. In VFC *consistency zones* Z_i are defined as follows: if i = 1, then Z_1 is the inner circle of the surrounding the *pivot* (in Figure 3.1(b) it corresponds to zone Z_1); if 1 < i < n, then Z_i is the area enclosed between Z_i and Z_{i-1} (zone Z_2 in Figure 3.1(b)); if i = n, then Z_i is the zone beyond the last circumference that surrounds the *pivot* (in the case of the Figure 3.1(b) it corresponds to Z_3).

Consistency degrees are by 3-dimensional *consistency vectors* (κ) that bound the maximum divergence between the objects inside a consistency zone in a given view regarding the value of their primary replica. Each dimension corresponds to a scalar value defining the maximum divergence of objects regarding the following metrics:

- *Time* (θ): Specifies the maximum time (in seconds) an object can stay without being refreshed with its primary replica's latest value. With this metric VFC guarantees that an object within a consistency zone specified by vector κ is, at most, κ_θ seconds in an inconsistent state. Informally, it captures this metric "freshness/staleness" of an object;
- *Sequence* (σ): Specifies the maximum number of updates to the primary replica that are allowed to not be applied to the object (missing updates). With this metric VFC guarantees that an object within a consistency zone specified by vector κ is, at most, κ_σ updates behind the primary replica;
- *Value* (ν): Specifies the maximum divergence between the contents of the local copy of an object and its primary replica. *Value* is an application-dependent metric that defines the maximum percentage difference of an object to its replica (divergence impact). *Value* is calculated by a function specially defined by the application's programmers. VFC guarantees that an object within a consistency zone specified by vector κ is, at most, κ_ν percent divergent from its primary replica.

VFC guarantees that an object is updated whenever at least one of the previous criteria is about to be violated. Consider a consistency vector $\kappa = [0.25, 6, 20]$. An object within the consistency zone corresponding to κ is guaranteed to be, at most, 0.25 seconds outdated or 6 updates behind the primary replica or with contents diverging 20% from the object's latest value.

VFC Generalization

So far we discussed VFC considering that each node has a single view and a single *pivot*. However, VFC is a *multi-pivot* and *multi-view* model - it allows both multiple pivots per view and multiple views per node. Considering a "capture-the-flag" First Person Shooter game, with the multi-pivot feature it is possible to have both the flag and the player's avatar as pivots. This way, the player would receive updates regarding his nearby opponents and the surroundings of the flag. In a multi-pivot setup, an object's consistency zone is assigned with relation to its closest *pivot*.

The multi-view feature allows different sets of objects to be characterized with different consistency requirements regarding the same *pivot*. Considering a team game with a player's avatar as his *pivot*,

objects representing the player's teammates can have different requirements than objects representing the player's opponents.

Despite this generalization, in our system we use VFC as a single-pivot and single-view model. Hence, when discussing the VFC consistency model, hereafter we refer to the terms *pivot*, *view* and *client* interchangeably.

3.2.2 Consistency Enforcement

The VFC model is enforced by a two part algorithm, with each part performed at independent times. The first part consist in keeping track of the updates to the objects of the system and maintaining the information required to enforce clients' consistency. It is executed by function *update-received* every time a client issues an update to an object. The second part of the algorithm consists in periodically updating clients according to their VFC settings. This requires deciding which updates are to be propagated to each client, a task executed by function *round-triggered.*

To enforce the VFC model the following data structures are required:

- Dirty Objects Table (D): Table that stores, for each client, information about which objects have been modified since the last time they were updated to that client;
- Consistency Vector Table (K): Given an object o and a client c, K returns the consistency vector corresponding to the consistency zone of c's AOI where o is located, regarding c;
- Current Consistency Vector State (S): S keeps per client information about the state of every object of the system regarding each client's consistency requirements (specified by VFC). It stores the last time each object was sent to the client (as a part of VFC enforcement), the number of missing updates to it (i.e, the number of updates since the last time the object was refreshed at the client) and the *Value* of the object at the time of the last time it was sent to the client.

update-received The update-received function is executed whenever the server receives an update to an object o. The operation is simple: when the update is received the number of missing updates to o is increased by one for every client of the system (i.e, for each client c, $S[c,o] = S[c,o] + 1$); then *update-received* checks (considering object o), for every client c, if the resulting number of lost updates or the total time since the last update or the difference between the object's value since the last update exceeds the maximum divergence allowed for any of those criteria in that view; if so, the object is considered dirty and $D[c, o]$ is updated accordingly.

round-triggered VFC updates the local view of its clients by (periodically) sending them *round* messages piggybacked with the due updates. To do so it first invokes function *round-triggered*, which returns the updates to be sent to each client, based on the data gathered and processed in *update-received.* The algorithm that identifies which objects are to be sent is straightforward: for each *pivot* identify those objects that are within the *pivot*'s AOI; then, for each object identified, check in which *consistency zone* of the *pivot*'s AOI that object is located; finally, verify (using the information gathered and stored during the execution of update-received) if the object violates the *consistency degree* associated with the *consistency zone* it is located in; if so, the object has to be sent. After updating client c with information

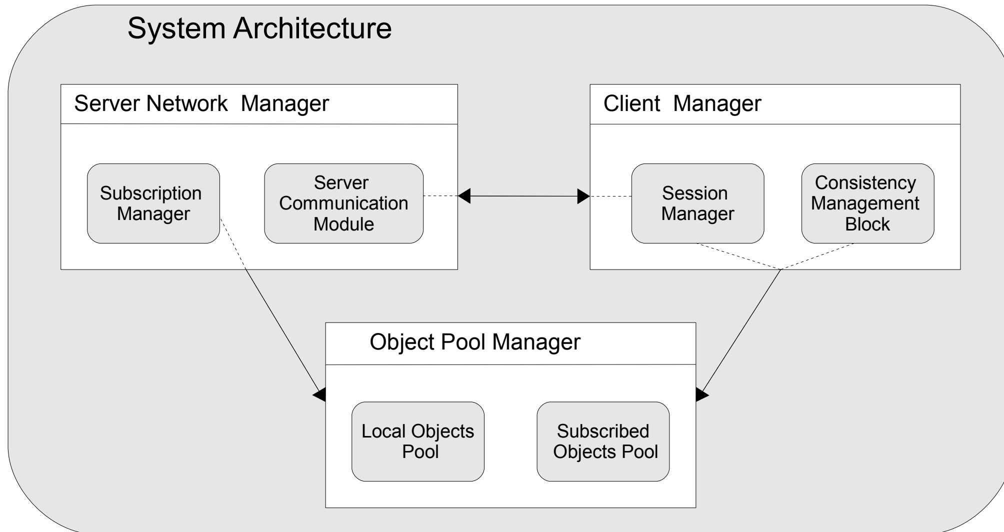


Figure 3.2: VFCLS architecture: Components and interactions.

about object o, the server updates S[c, o] accordingly, i.e, sets the last time o was sent to c as the current time, sets the number of missing updates to o back to zero and stores o's *value*.

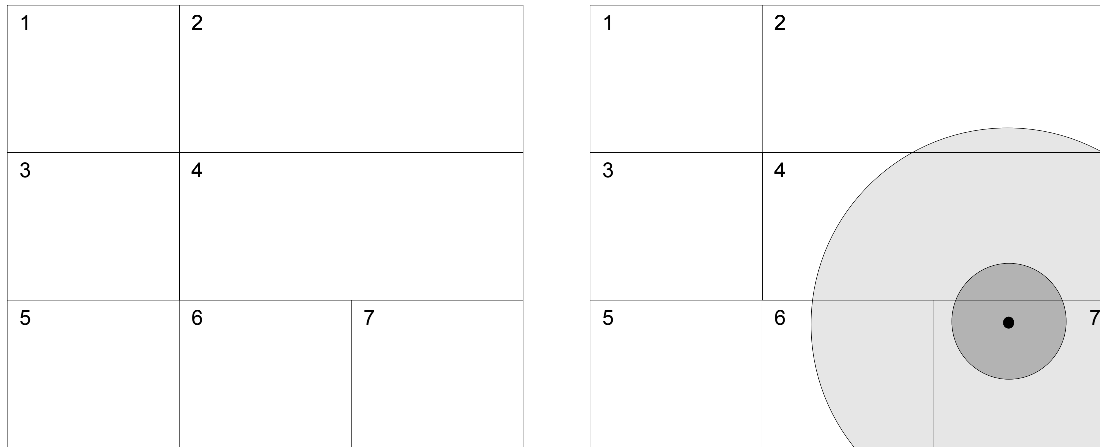
3.3 System Architecture

Figure 3.2 presents shows the core components of the VFCLS architecture and their interactions. VFCLS comprises three main building blocks:

Server Network Manager Manages the server's view of the server network (partial views) and handles server to server communication - object subscription and state synchronization. It is composed of two components, the "Subscription Manager" and the "Server Communication Module". The Subscription Manager is in charge of the execution of the subscription protocol. The Server Communication Module is the server's interface with the other servers of the network. It is responsible for coordinating and executing every communication between a server and the other servers of the network.

Client Manager The Client Manager administers client data and enforces VFC through two components, the "Session Manager" and the "Consistency Management Block". The latter is responsible for creating and maintaining the data structures necessary for the enforcement of the VFC model, as well as for verifying and retrieving the updates that have to be send to the clients. The "Session Manager" administers client connections, stores and manages client data and directly communicates with clients - receives their updates and sends state updates according to the information retrieved from the "Consistency Management Block".

Object Pool Manager Manages the server's game objects and encapsulates the stored data, performing every operation on it on behalf of the other components. The data repository is divided in two pools - one for the server's local objects and the other for the subscribed objects - due to our VFC enforcement



(a)

Figure 3.3: Example of a partitioned virtual world.

protocol, which was extended in order to support inter-partition interaction. Both the Server Network Manager and the Client Manager (through their internal component) use the information managed by the Object Pool Manager in order to perform their respective tasks.

3.3.1 Server Organization

As we mentioned earlier in this Chapter, servers are organized in a content addressable P2P network. Each server has a partial view of the network - it is only aware of a subset of the complete server set - constructed and maintained by the "Subscription Manager". This partial view always contains the server's direct neighbors and may also contain a number of other servers it needs to be connected to in order to support inter-partition interaction and player transfer.

State Partition

Figure 3.3 shows an example of a possible virtual world partition in our system[2]. In the image we can see that the virtual world is divided into straight rectangular partitions and that different partitions may have different dimensions. We chose this shape for our partitions because it is computationally inexpensive to maintain and operate and simpler to reason about.

Despite the logic partitioning of the virtual world, our system gives the user the illusion of being in a single large virtual world. We allow players (through their avatars) to move around the virtual world seamlessly and to interact with players located on other partitions. For example, considering Figure 3.3(b), the player represented is allowed to move from partition 7 to partitions 4 and 6 (and from there to any other) as if there was no division. Furthermore, the player could be interacting with any player located on partitions 7 (his own partition), 4, 6 and 2. In Section 3.3.2 we discuss how our system supports these features.

[2]For simplicity, in our discussion we consider only 2-dimensional virtual worlds; the results and conclusions drawn considering these dimensions can be easily mapped to 3-dimensional virtual worlds.

Server entrance

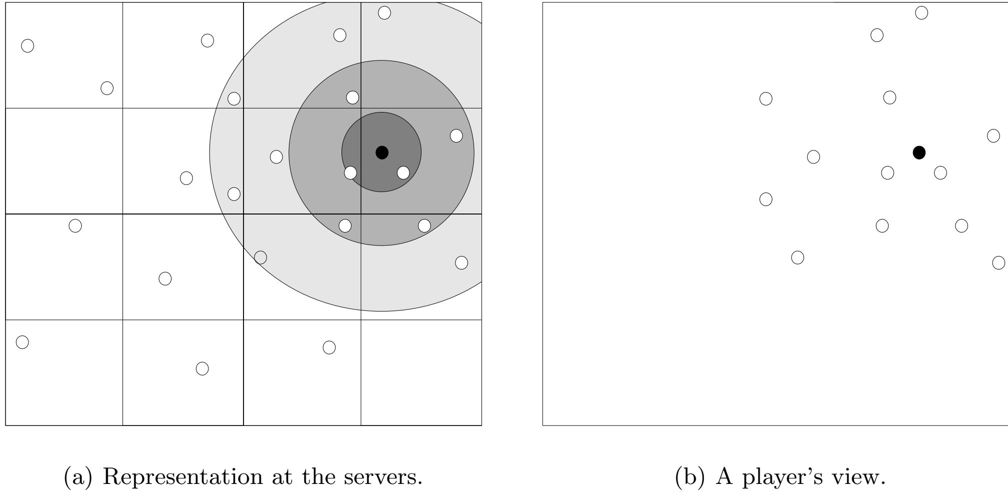
The organization of servers in the network is assembled while servers join the system. In our system, contrary to what happens in other P2P infrastructures (like CAN), a server entrance is expected to be relatively uncommon; typically a server joins the network on setup — i.e, before the game is open to users — or with the goal of reducing load on an overloaded server. Therefore, the system provides two different join strategies, one for which of the previous cases:

Setup-time server join Setup time occurs before the game is officially launched. In this phase the game consists only of the virtual world, immutable world objects and, in some cases, NPCs (Non-player characters). At this point in time game companies are likely to want to have great flexibility to set the initial game environment. For that purpose, we provide a manual server join that allows game programmers to manually define the dimensions of each partition of the virtual world as well as define each server's set of neighbors and known servers, so that each server directly informs them of his entrance as opposed to having to find them. This feature allows game companies to set the game environment based on the load predictions they usually make. It should be noted, however, that setup-time server join does not need to be manual. If game developers do not want to spend time manually setting up the game environment they can simply use the automatic run-time server join protocol, which will be described in the next paragraphs.

Run-time server join Unlike its manual complement, run-time server join is, as its name indicates, meant to occur while the game is already in execution. The most common reason to add a new server to a game is to reduce the load on a pre-existing machine. If this is the case, the node that joins the network has to know the address of the server that is in need of help and directly sends it the join request. The two servers then split the partition between each other and inform their resulting neighbors about the new configuration. Optionally the joining server may carry information about the desired dimensions of the partitions that should result from the split.

Because the game is in execution when a server joins the network it is necessary to take some precautions in order to minimize the occurrence of losses and prevent other problems during the joining process. For this purpose, when a server S_i joins the network it remains in synchronization with the server S_j with which it has split its partition until the system is again stable (i.e, until every server that should have S_i in its partial view has it and every player whose avatar is located on S_i's partition has been transferred to it). To do so, it first copies from S_j the objects located on S_i's new partition. Then it informs the servers on its newly created partial view that it is now in charge of the new partition. As a result, the other servers update their partial view with S_i and are now ready to include it in server synchronization (and vice-versa). Only then the server informs its clients that it is their new designated server by piggybacking that information in the state update messages of the next VFC enforcement execution. After receiving the transfer acknowledge from every client, S_i notifies S_j that synchronization is no longer required and effectively becomes the partition's new server.

A new server can also be added to the network to replace an existing one. In this case, instead of a partition split the servers perform a partition migration, from the old server to the entering one. After the migration process the new server has a complete copy of the replaced server's state, including virtual world representation, game objects and client and server data.



(a) Representation at the servers. (b) A player's view.

Figure 3.4: Multiple region inter-partition interaction.

3.3.2 Inter-server Communication

As we said before, although our system partitions the virtual world into regions it provides a seamless view to the players. This means that players (through their avatars) are able to interact with each other without any limitations other than the ones imposed by the virtual world design and the VFC specification. It also means that avatars can move around the virtual world, switching between partitions (and, as a result, servers) without the user being aware of it. As we have stressed before in this document, this approach contrasts with the limiting strategies typically used by commercial games.

Figure 3.4 shows an example of a simple virtual world (a) as represented by servers and (b) as perceived by a player (solid circle). Because the consistency zones of a *pivot* object may cross partitions handled by a different server, servers may need to share information with each other in order to enforce VFC. To do so while still minimizing server synchronization requirements we designed a subscription protocol in which servers communicate only when strictly necessary, to ensure the correct enforcement of VFC. In this section we describe the subscription protocol and explain how VFCLS supports player transfer through a protocol that is dependent on the subscription mechanism.

Subscription Protocol

Inter-partition interaction occurs when two interacting players are located on different partitions/regions of the virtual world. Given that each partition is managed by a different server, supporting inter-partition interaction requires extra server synchronization mechanisms. In Figure 3.4 we can see a player (represented by the solid black circle) interacting with several other players located on his and other partitions. We consider that two players interact when one is within the other's AOI, whether they are actually (at the application level) interacting or not.

Other works that propose seamless partitioned virtual worlds have addressed this topic, but in a very simplistic manner [4, 10, 5]. First, in those systems inter-partition interaction only occurs between neighbor servers and near the border of the partitions (hence the name "border interaction"). It is not clear, however, if the authors assume that players' AOIs are defined in such a way that only this kind

of interaction is allowed or if the design of the system imposes that only players in contiguous partitions can interact.

Our system, on the other hand, makes no such assumptions nor limits inter-partition interaction to neighbor servers. Instead, because VFC allows game programmers to freely define consistency zones, in our system, a player's AOI can cross multiple partitions, including non-neighbor ones (as is represented in Figure 3.4). Thus, we provide full support for inter-partition interaction that includes regions beyond a server's "neighborhood".

Inter-partition interaction is achieved through a subscription protocol arranged in cooperation by the Subscription Manager and the Server Communication Module. The protocol is divided in three parts, performed at independent times:

Server Subscription When we explained the entry protocol of a server we, purposely, neglected to explain a step of major importance for the operation of the subscription protocol. When a server joins the network it immediately identifies and connects to those servers whose partition its objects' AOI may cross. To do so, it inspects the objects located on its newly defined partition to find the one with the largest AOI radius R. It then publishes to the network the dimensions of the area *partition outset* defined by adding R to each side of the partition. As a result of this publication, every server whose region is crossed by the *partition outset* informs the publishing server of its existence and is added to its *partial view*.

Object Subscription The main task of the Subscription Manager consists in subscribing its own player objects (*pivot* objects) to servers that may contain information required by its players. It does so by continuously and periodically executing an object subscription protocol that runs as follows:

1. The Subscription Manager starts by checking if any of its players' AOI crosses the partitions of the servers on its *partial view* (defined in the previous step, i.e., when the server joins). When it checks that an object's AOI crosses the partition of another server it does not perform the subscription immediately; instead it adds the mapping "server $\leftrightarrow$ player object" to a *subscription queue* to be processed in the next step.

   To allow programmers to define different types of objects with different consistency requirements and AOI radius it is necessary to check, for every player, if the radius of the AOI of its avatar is larger than the largest AOI radius known by the server. If it finds one such object O it has to update its *partial view* with the server that are crossed by the new *partition outset*. To do so the server again publishes the *outset*, this time piggybacked with a subscription request for O. As a result, the servers whose partition is crossed by the *outset* subscribe to object O and are added to the server's *partial view*.

2. Then, the Subscription Manager uses the *subscription queue* of step 1 to publish the list of objects to the servers determined in that step. Publication is performed by directly sending, to each server, the list of objects whose AOI crosses its partition and, thus, require information about objects only known by it.

3. After the subscription process is finished, the mappings "object $\leftrightarrow$ subscribed server" are stored in a *subscription table* at the Subscription Manager. Later the Server Communication Module will

require information from this table to perform the second part of the server subscription protocol - server synchronization.

Server Synchronization The second part of the protocol consists in maintaining objects synchronized between the servers according to the subscription results. Synchronization is necessary because, as explained in section 3.4, servers apply VFC both to their *owned* (i.e., the objects located on the server's partition) and *subscribed* players (i.e., those located on other partitions, but that have subscribed to the server). Hence, servers need to be informed of the positions of the subscribed player's avatar.

Synchronization is performed, optimistically, every time a player submits an update. When an update is received (and after the server responds to it) the Server Communication Module consults the Subscription Manager and retrieves from it the entry of the subscription table corresponding to the object to update. Then, it forwards the received update to the servers on that list.

As a result of these steps (i) the servers become aware of those players located in other partitions that may need information about their objects and (ii) each server knows to which other server updates to a given object have to be forwarded. Thus, each server has all the information it needs in order to perform VFC enforcement and, as a result, allow inter-partition interaction. Another consequence of the protocol is that each server has its object pool divided in *owned objects* and *subscribed objects*.

Player Transfer Protocol

Providing a seamless view of the virtual world to players requires our system to allow a player's avatar to transparently move between partitions. When an avatar moves to another partition its designated server becomes the one responsible for that region. Thus, the new server needs to receive the player's data from his previous designated server. If the object transfer is triggered only when the player's avatar switches to the new partition the delay caused by transferring its data to the new server may be noticed by the user. One possible solution to prevent this from happening consists in having servers transferring a player's data before that information is actually needed. With this simple technique - called *data prefetch* - when an object moves to another server's partition, the latter already has the object's data and, thus, the transfer consists in a simple notification message.

In VFCLS we take advantage of the fact that the subscription system (described in the previous section) needs information from objects located on remote servers to achieve similar results as data prefetch by executing player transfer when a server receives an object publication message. When a server S_i sends a subscription message directly to another server S_j it piggybacks the player's data on that message. When S_j receives the message it gets all the information it needs about the player from the piggybacked data (hence, there is no real data prefetch - the server receives all the information without requesting it). This way, the actual player transfer involves small amounts of data and is unnoticeable to the user.

The actual transfer of a player from his current designated server S_i to his new designated server S_d occurs only when the player's avatar moves to that server's partition. The transfer is triggered by S_i after it receives an update from the player that positions its avatar on a neighboring region. After finding out (by analyzing the entry of the subscription table of the Subscription Manager corresponding to the object) which server S_d is responsible for that region, S_i issues the transfer request to it. Because S_d has previously transferred the player's data it needs no additional information. As such, the transfer

request is, in fact, a simple one-way, asynchronous transfer notification. As soon as the transfer request is completed, S_d notifies the transferred player's client that it is its new designated server. From then on, the client communicates exclusively with server S_d.

After the transfer terminates the two servers involved, S_i and S_d, perform a subscription inversion - S_i, the previous designated server, becomes a subscriber of the object, while S_d, a former subscriber, becomes its new designated server. Because both servers have all the required information about the player, the inversion is automatically and independently executed by each server, without any message exchange between the two.

3.3.3 Client - Server Communication

In VFCLS clients are connected to a single server at each moment. When they join the network they are assigned to a given server and remain connected to that server throughout the execution of the game, until their avatar moves to a position located on a partition managed by a different server, in which case the player is transferred to the new server. The management and execution of the interactions between a server and its clients is performed by the server's Session Manager.

Four types of messages can be exchanged between a player and its designated server. Two of them, player updates and player joins, are issued by the client; the remaining two, state updates and player transfers, are triggered by the server. Player updates and state updates are the most frequently exchanged messages. Player updates are issued by a client whenever the player inputs an action command - like a move or a shoot. State updates are sent, periodically, by each server to their clients as a part of the VFC enforcement protocol. Player join requests are sent from a player that wishes to enter the game. Server transfers are sent from a designated server to a game client to inform it that its avatar has moved to a partition handled by a different server.

In the following section we describe the two player issued messages. Player transfers have already been covered in Section 3.3.2 and state updates are addressed later in Section 3.4.

Player Join

When a player wishes to join the network he first is required to find which server will be his designated server. One simple way to do this would be to have a know public server storing information about all the servers of the network and their corresponding regions. The player would send the request to that server and retrieve automatically the address of his designated server. For our system, however, we opted for a decentralized approach that consists in routing the join protocol through the server network until the desired server is found. The joining player sends the request to any of the servers of the network which runs a distributed lookup protocol, orchestrated by the Server Network Manager, and returns the desired server's address. This way, players can join the game using any known server, e.g., one where a fellow player is already connected to. Furthermore, with this approach, players have multiple entrypoints through which they can enter the game, as long as the addresses of the servers are public (e.g., available in Internet forums).

The server lookup protocol is similar to the lookup protocol of CAN. First, the client sends a lookup request to the server containing the player's initial position. On receiving this message the server checks if the position sent lies within the limits of its own region. If so the protocol terminates and the server

returns its own address directly to the client. Otherwise, the protocol continues as follows. The Server Network Manager starts by checking which of the servers on its partial view is closer to the player's desired position. Then, the Server Communication Module forwards the lookup request to that server.

These previous steps are repeated by each of the servers that receive the forwarded lookup until the right server is found. After receiving the address of its designated server the player sends it a join message with its data and as a response receives the necessary data from the server. After that he can start playing the game, send his updates and receive state updates from the server.

The reader may notice that the lookup and the join requests could be merged into a single message, by applying the lookup protocol and sending along with it all the necessary information for the server to register the joining user. However, we decided to separate this process in two distinct operations to reduce the bandwidth expenses on the participants of the lookup protocol. This way, the messages exchanged between those servers are very small, containing only the players position. Only the server that really needs to receive the players data actually receives it.

Player Updates

In VFCLS, a server accepts player updates only to objects it owns. Updates to a subscribed object, on the other hand, are executed as a result of synchronization with that object's designated server. Player updates are sent from the client application to the game server according to the logics of the application. When the server receives, through the Session Manager, an update message from a player it invokes the Consistency Management Block. The CMB, in turn, executes a function (*update-received* necessary for the enforcement of the VFC model. The server may also have to forward it - via the Server Communication Module - to other servers according to the subscription information retrieved from the Subscription Manager, as described in Section 3.3.2. After receiving the response from the server the client is again ready to receive inputs from the player and to, again, send them to the server.

To reduce response times and, thus, improve performance, updates are forwarded optimistically - the server first responds to the player and only some time later it forwards the update to other servers. As a result of optimistic synchronization conflicts may occur. Currently our system does not have an explicit mechanism to detect and prevent conflicts. However, it guarantees that conflicts are, eventually, resolved (as a result of VFC enforcement) by overwriting the conflicting object with a correct version. In the future, however, we intend to address this issue and incorporate some conflict management mechanism in VFCLS.

Instead of sending the updates to a single server it would be possible to make the client send the updates directly to all the servers interested in them. This way the delay introduced by the extra step of optimistically forwarding updates could be reduced and, with it, the probability of conflicts. However we decided not to do so because it would yield much higher response times and make the client much more CPU and network demanding, both of which are the complete opposite of our goals.

3.4 Distributed Vector-Field Consistency

In our previous discussion we described the subscription protocol required to enforce the VFC model. We now explain how the information gathered by (and available due to) the subscription protocol is used

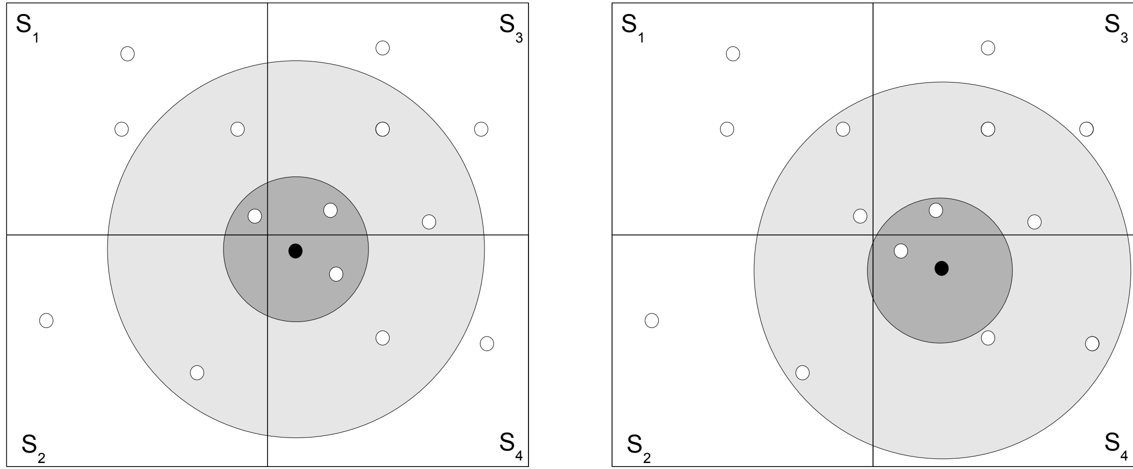


Figure 3.5: Owned and subscribed objects.

to actually enforce the consistency model.

It is now appropriate to explain one main difference between the original VFC model and our VFCLS extension. The original design of VFC forces programmers to define one zone Z_n that includes the whole virtual world in the consistency management (unbounded VFC). We believe, however, that having players receive updates from every object in the system is not suitable for large scale, wide and highly populated virtual environment, as it may overload both the players and the servers of the system. As such, in VFCLS we consider only *consistency zones* with finite (albeit arbitrary) radius, thus, removing the outer zone Z_n.

We mentioned before that, as a result of the subscription protocol, each server has a set of *owned objects* O and a set of subscribed objects S. Considering Figure 3.5 the player represented by the solid (black) circle is owned by server S_4 and is subscribed to servers S_1, S_2 and S_3 because his area of interest crosses their partitions. It is easy to see that, had we considered any other object owned by server S_4, it would also be subscribed to the other three servers (assuming that its AOI is equal to the one represented), as is represented by Figure 3.5.

The Distributed VFC algorithm ensures that a player receives updates from every game object that is within its AOI, even those located at different partitions. This is achieved by having each server updating both its owned and subscribed players with the information about the objects located on its partition. This means that each server enforces VFC only for the slice of a player's AOI that crosses its partitions. As a result, the responsibility of enforcing VFC for a given player is divided between the servers whose partitions are intersected by the player's AOI.

Considering again Figure 3.5, consistency for the player p represented by the solid (black) circle would be enforced complementarily by the four servers: S_1 would be responsible for updating p with the information about the four objects located in its partitions, S_2 would update it considering only its two owned objects, and so forth. Hence, servers divide load between each other by enforcing consistency for disjoint slices of players' areas of interest, effectively distributing the responsibility of consistency enforcement for each of the players.

Our distributed version of VFC enforces consistency through the same two functions provided by

the original VFC, update-received and round-triggered. The general method is similar: throughout the execution of the game, servers maintain the information necessary to enforce VFC to their players by keeping track of their updates. Periodically, the server issues a round in which it updates its players, according to their VFC specifications.

3.4.1 Update Processing

To update clients according to their VFC specification, servers have to monitor every modification to the objects on their pools. For that purpose, every time a server receives an update (as a result of a direct player update or synchronization with another server) it transfers control to the "Consistency Management Block". Depending on the origin of the update, the "Consistency Management Block" executes one of the following actions:

- If the update is received, directly, from a player, then it concerns an *owned object*. Hence, because in VCFLS each server enforces VFC considering only its owned objects, the server updates its information about each player's (considering both owned and subscribed) current state of consistency to reflect the update received. This is done, like originally in VFC, by function *update-received* and the operation is the same (see Section 3.2 for more information).
- If the update is received from a server as a result of server synchronization, then it concerns a *subscribed object* corresponding to a player p owned by a different server. As such, the information received is only necessary to update the object's position, so that, when enforcing VFC to player p, our system can correctly identify which objects are within his AOI.

3.4.2 Updating Clients

The process of updating clients is almost equal to the way it is done in the original design of VFC. Servers update clients by periodically sending them round messages piggybacked with each client's updates. The only difference in this case is that clients can be both *owned* or *subscribed* and the servers, when checking consistency for those clients, only consider their *owned* objects. The revised algorithm is as follows:

1. First, the server identifies, for each owned player p_o, which objects *owned* by the server (i.e, those that are located in the server's partition) are within p_o's AOI. Then, for each object o previously identified, it checks in which *consistency zone* of p_o's AOI object o is located. Finally, it verifies if o is in violation of the *consistency degree* associated with that *consistency zone*. If so, that object is queued and, after verifying the remaining objects, the server sends it to the player p_o.
2. Then, the server performs the exact same steps, but now enforcing consistency to its subscribed players. Hence, the server identifies, for each subscribed player p_s, which objects *owned* by the server are within p_s's AOI. Then, for each object o previously identified, it checks in which *consistency zone* of p_s's AOI object o is located. Finally, it verifies if o is in violation of the *consistency degree* associated with that *consistency zone*. If so, that object is queued and, after verifying the remaining objects, the server sends it to the player.
3. After verifying consistency for every player (owned and subscribed), the server sends them the round message piggybacked with the objects identified in the previous steps.

As we can see, steps 1 and 2 differ only in the fact that 1 verifies consistency for the server's owned players, while step 2 does the same but for the server's subscribed objects. The reason for this separation lies in the fact that the information about the consistency state of owned and subscribed objects is stored in different data structures, just like the information about the position of those objects is (one is stored in the *local objects pool* and the other in the *subscribed objects pool*).

As a result of the combined work between the subscription protocol (performed by the "Subscription Manager") and the consistency enforcement algorithm (executed by the "Session Manager" in conjunction with the "Consistency Management Block") we achieve a distributed VFC algorithm in which the consistency of a single player is enforced not by a single server but by the complementary work of a group of servers. Having the load and the responsibility for enforcing consistency divided between the nodes of the network improves the flexibility of the system and fosters scalability.

3.5 Summary

In this section we described in detail the architecture of our system. We started with a general overview on the system and its most relevant aspects. Then, we described the original design of the Vector-Field Consistency model that is used, with extension, by our VFCLS to apply interest management. We then described our object subscription system, necessary to allow interaction between player's located in different partitions and transfer of players between partitions. We finished the Chapter by describing our modified distributed Vector-Field Consistency algorithm.

Chapter 4

Implementation

This Chapter present some details about the implementation of our system. We start by providing information about our development environment. We describe the main data structures of our system, both the completely new and the ones adapted from the previous implementation of VFC. We also present details about the algorithms and protocols used by VFCLS, as well as describing the simulation infrastructure designed to evaluate our VFC prototype.

4.1 Development Environment

Our VFCLS prototype was developed in the Java programming language. More specifically we developed the system using Sun's J2SE 6.0 development kit (JDK) and runtime environment. VFCLS was developed using only the standard Java libraries provided by JDK.

We used Java's Remote Method Invocation (RMI) architecture to support communication between the nodes of the system. As a result, every message exchanged between the nodes of the system are implemented as methods of a remote interface. Later in this Chapter we present the remote interfaces designed for our prototype.

4.2 Algorithms and Supporting Data Structures

In this section we overview the main algorithms and protocols of VFCLS, as well as its mains supporting data structures.

4.2.1 Object and User Representation

VFCLS incorporates two fundamental data structures based on the original VFC implementation: DataUnits and UserAgents. In our work they were subject to extension, necessary to support the distributed nature of VFCLS. We highlight those modifications where appropriate.

As far as VFCLS is concerned, the virtual world is a bounded area populated with DataUnits. A DataUnit (DU) is an object that represents a shared game entity (like an avatar or a food object). Each DU carries a unique integer session identifier *duId* and the DU's position in the virtual world. To create a personalized game object, with other variables, a programmer using VFCLS is required to write a DU subclass containing the extra data fields. This feature makes VFCLS completely independent of the application that runs on top of it. Not only can it support different games, but also has a great potential to support different types of applications, in the future.

Users are represented in the system by class UserAgent (UA). Like DUs, UserAgent objects also have a unique integer session identifier (*uaId*), along with an also unique nickname and the user's remote interface. The server also stores a list containing the mapping between UserAgents and its corresponding DataUnits. As we mentioned earlier, to simplify our implementation we considered that each user is associated with a single object/DataUnit corresponding to the player's avatar. In the future we intend to give full support to the multiple objects per user feature. Nonetheless, this can be partially achieved by having composite DUs that serve as containers for other DUs.

4.2.2 Object Pool

The object pool is implemented by class DataPool. Although this class was inherited from the original implementation of VFC it was subject to an almost complete re-write. In the inherited implementation the DataPool was internally implemented as a fixed size array of DataUnit. Adding objects to the pool could only be performed at *Setup*: players would join the network, registering their objects on the single server, and wait for the server to notify them that the game had begun. After receiving every player registration, the server would create the DataUnit array with its fixed size corresponding to the number of registered objects. DataUnits were placed on the array index corresponding to their id - not only the array was fixed sized, but it was also immutable. This way, accessing any object of the array could be achieved in constant time. However, it required players to join the game at the same time and the server to know every player in the game. While the latter characteristic is desirable in a single server architecture, it is not suitable for our partitioned virtual world approach in which a server may not know every object of the game and new objects may be added or removed due to player joins, transfers and subscriptions. To support these and other operations we implemented the object pool internally as an hashtable encapsulated by the class DataPool.

4.2.3 Subscription Protocol

We now present the implementation details of our Subscription Protocol.

Server Representation

In VFCLS servers are represented by class VFCLSServer. VFCLSServer objects have a unique string identifier corresponding to its network address (ip address and port) and the server's RMI interface. Each VFCLSServer instance also has a Region associated with it, representing the partition of the corresponding server. Class Region holds the partition's boundary values and center, besides providing the region cross checking methods used by the subscription system. VFCLSServer class is used by each server to represent the servers on its partial view. The partial view, in turn, is represented by a simple list of VFCLSServers.

Subscription Protocol

The subscription protocol is performed, independently of the rest of the servers' operation, by a dedicated thread pool of the Subscription Manager. The mechanism is straightforward: threads in the pool iterate over the list of the users owned by the server and check, for each of them, if their avatars' AOIs cross any of the server's known neighbor partitions. At each time, different threads are performing region cross checking for different users.

We implemented the publication of *partition outset* messages by *flooding* the message to the network, i.e, propagating the message through every known server. Publication messages are identified by a pair (serverId, messageId) to guarantee that no server receives the same publication message twice. A server that receives a publication stops to propagate it when (1) it has already received it or (2) its partition is not crossed by the published *partition outset*. We opted for flooding because it is a simple and easy to implement and, more importantly, because these publication messages are expected to be very rare.

The subscription protocol uses two auxiliary data structures: a temporary queue holding the mapping "server $\leftrightarrow$ subscribing data unit" to send to the other servers after region cross checking terminates; a permanent *subscritpion table* (implemented as an hashtable) storing the mapping "data unit $\leftrightarrow$ subscribed server", to be used later in server synchronization.

As we explained in the previous Chapter, server synchronization is optimistically performed after a player update is received by a server. Optimistic synchronization is achieved by using a queued thread pool. The operation is simple: when a server's Session Manager receives an update it adds it to an *updateQueue* managed by the aforementioned thread pool (managed by the Server Communication Module) and immediately replies to the client; sometime later one of the threads of the pool extracts the queued update and sends it to the servers that subscribed that object. The information about which servers subscribed to the object is retrieved from the Subscription Manager's subscription table.

4.2.4 Distributed Vector-Field Consistency

Our distributed version of Vector-Field Consistency inherits and extends many of the original data structures used by VFC.

Auxiliary Data Structures

The consistency requirements of a player is represented by class Phi. This class stores the player's consistency zones (in the form of an array of integers corresponding tho the zones' radius) and degrees (stored on a bidimensional array). Phi also contains references to the objects owned by the player.

TO maintain information about the consistency state of each client we use an Hashtable, indexed by clientId, that maps the client in a list of objects of class AOIInfo. AOIInfo is the class that represents the current state of an object regarding a particular player's view. It contains a reference for the object it refers to, the number of missing updates regarding the player's view, the value of the last update of the object and a boolean field that indicates if the object is dirty.

In VFCLS owned objects and subscribed objects are stored in different hashtables.

updatesReceived

The updatesReceived method is called, as it name indicates, whenever an update is received by the server. It is a small method that implements the *update-received* function of VFC. Its operation is simple: for each player's view, update the AOIInfo entry that corresponds to the object in question by incrementing the number of missing updates and setting the object as dirty.

computeUpdatesToDisseminate

The method computeUpdatesToDisseminate implements the function *triggered-update* of VFC: for each client located on the owned and subscribed player hashtables and for every object that is on the server's local objects pool check if that object is dirty (by inspecting the object's AOIInfo). If it is dirty then check if it should be sent to the player, i.e., if it is in violation of the consistency requirements of the player. If the objects has to be sent to the client, its AOIInfo is reset - the number of missing updates is set to zero and the value of the object is set to the current value.

4.2.5 Remote Interfaces

As we mentioned before, the nodes of our system communicate through remote method invocation.m The following remote interfaces were defined:

- IVFCLSClient: Client interface for incoming messages. Receives state update and server transfer (issued when a player is transferred to a new designated server) messages from VFCLS servers. State updates transfer an array of DataUnit objects to the player while a server transfer message sends the remote interface of the new designated server.
- IVFCLSServer: Server interface for client issued messages. Receives server lookups, player joins and player updates. Server lookup messages are issued when a player that wants to join the network wishes to find its designated server and trigger the lookup protocol described in the previous Chapter. Thus, the server lookup message carries only the position where the player's avatar initiates the game. Server joins are issued (after the server lookup) to the player's designated server and contain the player's remote interface as well as information about its objects and consistency (VFC) requirements. Player updates messages simply transfer to the server the DataUnits locally updated by the client.
- IVFCLSServerToServer: Interface for messages sent from other servers of the VFCLS server network. Includes join requests, player transfers, object subscription messages and updates from subscribed objects.

4.3 Game Programming Interface

Figure 4.1 shows the whole system perspective of a game designed using VFCLS. As we can see, the system includes not only the VFCLS server infrastructure (VFCLS Core) described in the previous Chapter, but also a client-side VFCLS application (VFCLS Client) and a VFCLS API that provides the means for the game application (both the client and the server) to interact with our system.

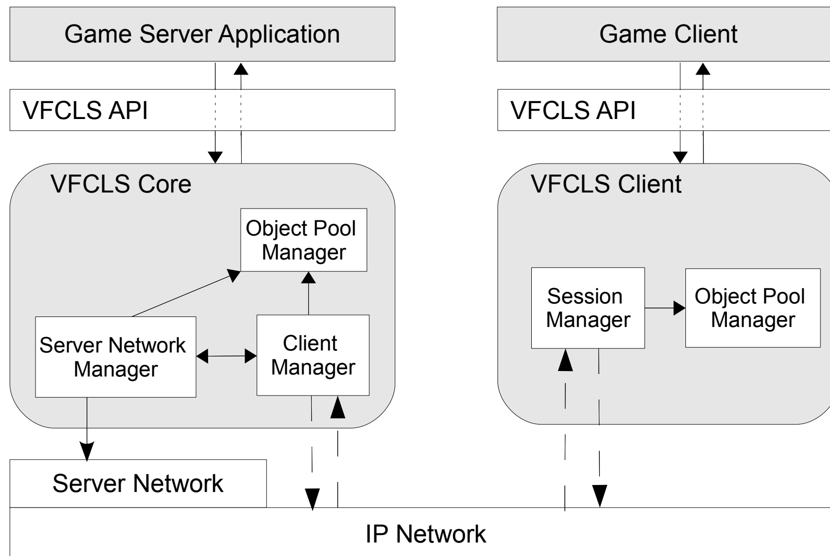


Figure 4.1: VFCLS integration with game applications.

Game applications interact with VFCLS through set of functions defined in VFCLS API. The API provides the following four main functions:

Object Registration For a game to use VFCLS it has to register its objects for VFCLS to manage. At the client-side the game client must register the players' avatar on the VFCLS client before it is used. Likewise, at the server-side, the game server application may potentially (depending on the game's design) register objects corresponding to computer controlled characters (NPCs). To support objects controlled by the game server application the "Client Manager" of VFCLS Core has a "Server Objects Manager" (not shown due to space constraints). This component is similar to the "Session Manager" in that it intermediates every communication with the server application, just as if it was a remote client - it receives the updates from the server application, handles object registration messages for NPCs and enforces VFC to those objects.

As we mentioned before, VFCLS represent objects as instances of class DataUnit. As a result, for an object to be registered it also has to be a DataUnit; this way, programmers must create their games' objects (at least those that are to be shared in the virtual world through VFLS) as instances of DataUnit class or any personalized DataUnit subclass designed by the client.

Object Update After registration, objects can also be updated according to game logics. When an object is locally updated by a client, the game client explicitly informs the VFCLS Client via a UserUpdate API message. As a result, the update is sent to the client's designated server. If the update concerns a server controlled NPC, then it is the server application that informs (via a ServerUpdate API message) the "Server Objects Manager".

Update Notifications Through the API, game clients (and server applications) can be informed when a state update message is received from a server. For this purpose, when the application starts, clients must register themselves as *update listeners* using a *RegisterStateUpdateListener* function provided by

the VFCLS API. This notification can then be used for various purposes by the application running on top of VFCLS: clients can perform graphics rendering to show the updated version of the virtual world to their player; the server application can use the newly received information to decide which should be the next action of an NPC.

VFCLS also provides player update notifications to the server application. Just like state update notification, the game server needs to register itself as a player update listener through an API function (in this case *RegisterPlayerUpdateListener*). This allows game servers to perform application specific operations, like checking the validity of the received update or even incorporating anti-cheating mechanisms.

Object Pool Querying The API also provides functions for applications to query the local object pool. This allows, for example, game servers to perform validation and anti-cheating periodically, instead of every time an update is received.

4.4 Simulation Infrastructure

To evaluate our VFCLS prototype we designed a simulation infrastructure that is able to simulate different types of interest management schemes and system architectures (server-side simulation) as well as different types of game clients (client-side simulation). The simulation is driven by class *SimulationDriver* which, as the name suggests, drives the simulation, i.e, reads configuration files, loads the classes necessary for the simulation and performs simulation steps in sequence.

Interest Management Simulation

Our simulation infrastructure is able to simulate different types of Interest Management models. To implement an IM algorithm it is only necessary to write a class that implements abstract class ACMB (from "Consistency Management Block") and, as a result, implements the methods updateReceived and computeUpdatesToDisseminate. Naturally, other methods and auxiliary data structures can be defined, but only to be used internally by the two previous methods (unless the programmer is also simulating a personalized architecture, in which case he can write both the IM and the architecture and define their interaction at will.

The IM to simulate and its parameters are defined in a configuration file. The configuration file path is passed as a command line argument to the SimulationDriver. The file must have (among other arbitrary parameters) a *IMClass* entry with the fully qualified name of the class that implements the IM model, so that the SimulationDriver can load it, using Java reflection. The file is then passed to the constructor of the CMB class.

Architecture Simulation

The simulation system also allows the simulation of different Client/Server architectures. To simulate an architecture our simulation infrastructure requires that the programmers define a class that overrides abstract class AServer. This class' constructor receives as arguments a class implementing the IM model

and the path of the configuration file containing the information necessary to setup the architecture. The implemented Server class also has to implement the method *startSimulation* which, as its name indicates, starts the simulation of the server architecture. For the simulation to work it is also necessary to provide an implementation for the remote interface IVFCLSServer, so that clients can communicate with the server.

Client Simulation

Our system also provides support for client-side simulation. Our approach to client simulation consists in simulating multiple threaded clients in a single process. Client simulation is oriented by a (client-side) SimulationDriver that receives as command line argument a configuration file (compliant with the Java Properties model) containing (among other possible parameters specific of the client simulation approach) the name of the class that implements the clients to simulate and the number of clients to simulate. The SimulationDriver supports multiple configuration files as arguments and, thus, can simulate more than one type of clients per simulation.

The class defining the client behavior must override class AClient whose constructor receives the configuration file passed as command line argument to the simulator. It must also override the abstract method *startSimulation*. To enable the simulation it is also necessary to provide an implementation of the IVFCLSClient interface.

4.5 Summary

In this section we presented the most important implementation details of our system. Among other things, we explained some of VFCLS's data structures, provided more information about the implementation of specific algorithms and described the architecture of our simulation infrastructure.

Chapter 5

Evaluation

In this section we present the experimental results of our evaluation of VFCLS. To evaluate our system we used our simulation infrastructure to simulate and compare different types of architectures (namely Centralized and Replicated C/S), as well as different Interest Management models, in particular auras.

To simulate clients we developed a simple game in our simulation infrastructure. In this game, automatic clients move their objects - small circles - in straight lines along the game map, periodically changing the direction of their trajectory. For the purpose of evaluation we consider that the size of each object is 200 bytes.

Our experiments were conducted varying different factors like the total number of players, the density of players, and the number and configuration of servers. The tests were performed on two machines equipped with Intel Core 2 Quad processors and 8.0 GB of main memory running on Linux Ubuntu distribution. The two machines are connected by a high speed Gigabit LAN.

In the following sections we present and analyze the experimental results obtained with the described simulation environment. We start with the results and analysis related to VFC and its comparison with auras. After that we present and discuss the results related to the architectural component of the work.

Variation	Description
Aura1	Aura with radius of 40 units
Aura2	Aura with radius of 80 units
Aura3	Aura with radius of 120 units
VFC1	VFC with two zones with radius [40, ∞] and respective K vectors [3,0,0] and [50,10,500]
VFC2	VFC with three zones with radius [40, 80, 120] and respective K vectors [3,0,0], [10,10, 0] and [50,10,500]
VFC3	VFC with two zones with radius [120, ∞] and respective K vectors [3,0,0] and [50,10,500]

Table 5.1: Description of the different parameters variations.

5.1 Interest management evaluation

The first set of tests we describe analyze the performance of VFC and compares it to auras and an alternative where no interest management is applied, regarding incoming client side bandwidth and efficiency. We compared these alternatives on several experiments varying the number of players, the size of the game map and also the parameters of VFC and the radius of auras.

Table 5.1 describes the parameter variation used on the experimentation. It should be noted that VFC2 is the only VFC configuration that does not have an outer zone Z_n.

Figure 5.1 shows the results of bandwidth variation, at the client side, for each of the three AOI schemes, when applied to a 1000 x 1000 with variable number of players (50, 100, 50).

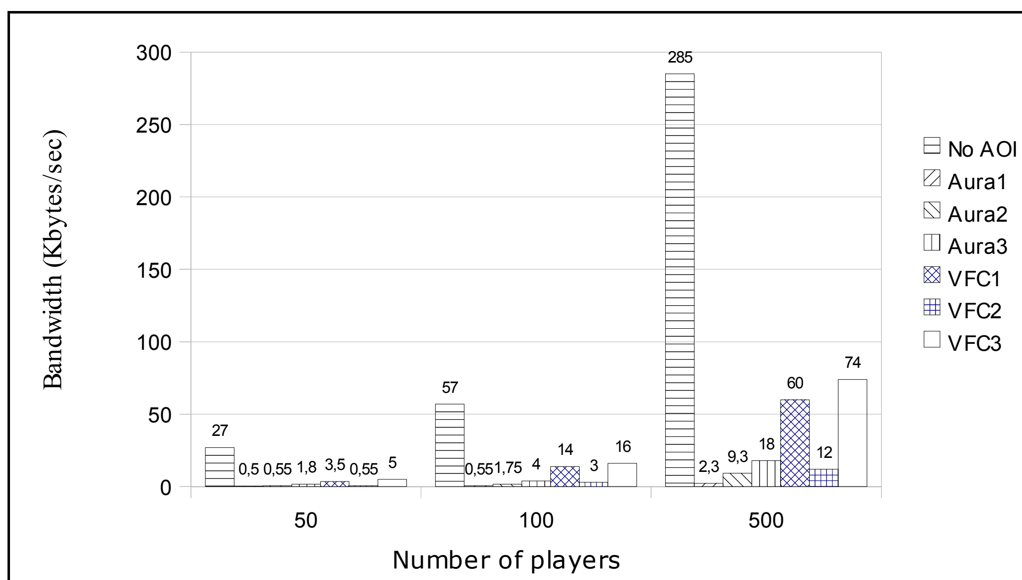


Figure 5.1: Client-side bandwidth requirements.

In this first graphic we can perceive the real importance of interest management. By analyzing the bar corresponding to NoAOI and comparing it to the remaining bars it is easy to see that without interest management bandwidth requirements grow fast with the number of objects (in our experiments we consider only objects corresponding to player's avatars and, thus, the number of objects matches the number of players). It is easy to conclude that without applying interest management it is not possible to scale games to large environments.

The importance of IM is even greater considering that games are growing not only in number of users, but also in complexity and realism. As a result, modern games tend to have more and more NPCs[1] and the size of game objects - both player controlled and NPCs - is increasing to accompany the growing realism.

In comparison to the NoAOI alternative every IM configuration yields considerably better results, even those corresponding to VFC with unbounded outer zones (VFC1 and VFC3). However, it is also clear that these two configuration perform considerably worse than auras and the bounded VFC configuration.

This evidence is confirmed by Figure 5.2, which shows the same results as Figure 5.1, but without

[1]Recall that NPC stands for "Non player characters"

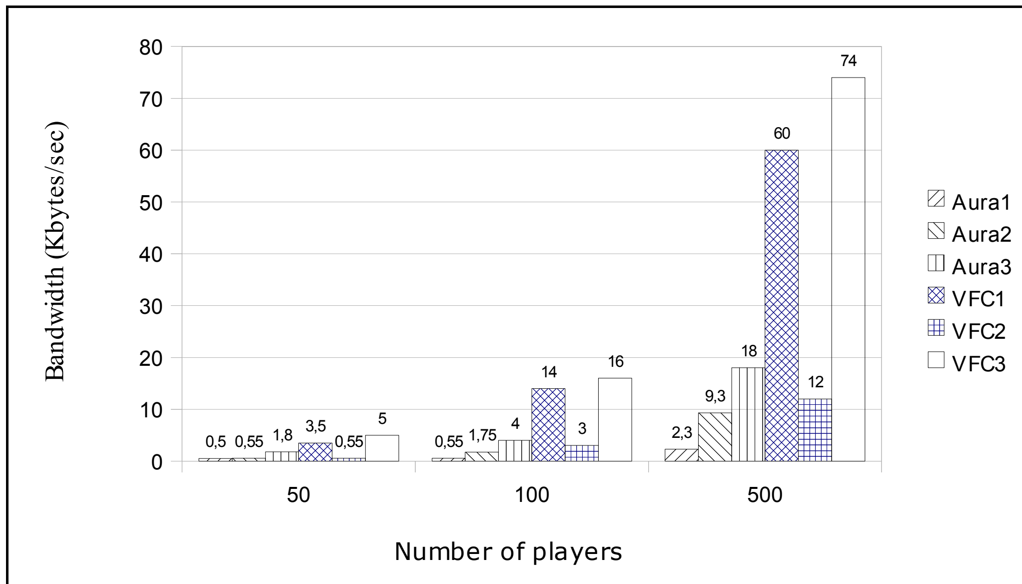


Figure 5.2: Client-side bandwidth requirements: results without the NoAOI configuration.

the "NoAOI" configuration. By observing the graphic it is clear that although VFC1 and VFC3 perform better than the NoAOI alternative the bandwidth required is still too high and, more importantly, increases too much as the system grows in number of players, specially when compared to the remaining bounded configurations.

Figure 5.3 presents results analogous to those of Figure 5.1, but considering two cases, one with 500 and the other with 1000 players, with a larger map of 5000 x 5000. Once more we can identify the impact of not using any interest management scheme on bandwidth. And, as before, we can see that unbounded VFC configuration still results in high bandwidth and, more importantly, are not very scalable, because they grow fast with the number of users.

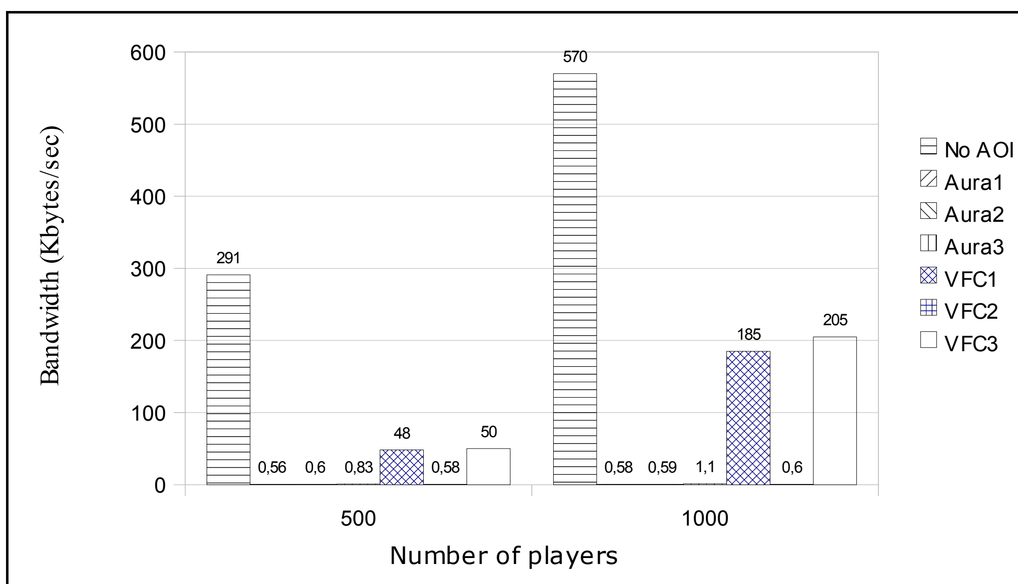


Figure 5.3: Client-side bandwidth requirements with 500 and 1000 players scattered across a 5000x5000 map.

By observing Figure 5.3 it is also possible to verify that, for the bounded configurations, the difference between the resulting bandwidth with 500 and 1000 players is virtually null. These results stem from the fact that, unlike unbounded schemes, in aura based strategies the number of players inside an AOI is not dependent on the total number of players in the game, but rather on player density, which depends both on the total number of players and on the size of the virtual world. Because of this it is even possible that in environments with more users the necessary bandwidth is lower than that of a game with less users but with a smaller virtual world. It is possible to see that by observing the bars corresponding to 500 players in Figure 5.1 and comparing it with the bars corresponding to 1000 players in Figure 5.3; the bandwidth of the former is much larger than that of the latter.

Figure 5.4 further shows the impact of player density by varying the size of the virtual world while maintaining the same number of 500 players. In it we can see that increasing the size of the virtual world results in lower client-side bandwidth usage for auras and VFC, while if no IM is applied the size of the map has no influence.

It is also interesting to note that the bandwidth reduction of VFC1 and VFC3 between the 1000x1000 map and the 5000x5000 is similar to the reduction of the other IM schemes for the same map variation. This fact indicates that the mentioned bandwidth reduction of VFC1 and VFC3 is caused by the decrease in the number of players located on the inner zones of those configurations.

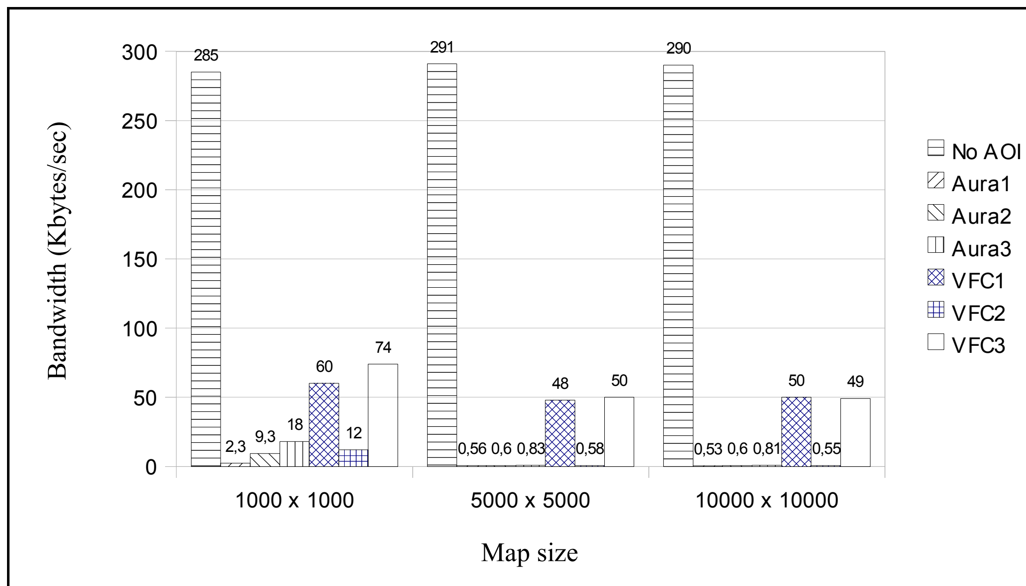


Figure 5.4: Client-side bandwidth requirements: influence of player density.

Bounded VFC vs Auras

The previous section showed that aura based bounded IM schemes scale better than unbounded ones. Now we will compare and analyze two different types of bounded IM schemes, auras and bounded VFC. We continue to use the configurations defined on Table 5.1 but we now focus solely on auras (Aura1, Aura2 and Aura3) and VFC2.

By looking at Table 5.1 we can see that the radius of Aura1, Aura2 and Aura3 are, respectively, 40, 80 and 120 and that VFC2 has three consistency zones with radius also with radius 40, 80 and 120. Figure 5.5 illustrates the differences between these configurations.

Figure 5.5: AOI size and variations.

Figure 5.6 shows the bandwidth differences between Aura1, Aura2, Aura3 and VFC2 in a context with variable number of players like that of Figure 5.1 - 50, 100 and 500 players in a 1000x1000 virtual world. The first thing we conclude is that, as expected, the performance of auras decreases as the radius increases, because the number of objects inside the AOI is higher. It happened on this experience as it would happened in every other that varies the radius of auras, depending only on the distribution of players - if the radius (and, therefore, the visibility of the player) increases, so does bandwidth, because the radius is the only parameter auras can vary.

In VFC, on the other hand, varying the radius of consistency zones does not necessarily means that the bandwidth will increase. Because VFC has other parameters that can be configured it is possible to increase the range covered by VFC zones while maintaining, or even reducing, bandwidth. This way, it possible to enlarge player's visibility with little or no impact on bandwidth, although at the potential cost of fidelity (but in a way that does not disrupt the playing experience).

Observing figure 5.6 we can also see that VFC2 only performs better than Aura3. To fully understand the meaning of these results, however, we have to analyze them in light of Figure 5.5. One of the things it lets us know is that Aura1 corresponds to the inner zone of VFC2. Hence, it is only normal that the resulting bandwidth of VFC2 is higher than the one generated by Aura1. It should be noted, however, that VFC2 corresponds to an example VFC consistency vector defined for our experiments. As we mentioned before, a different consistency vector, one that would relax even more the consistency requirements, could perform better than VC2 and possibly yield bandwidth results similar to Aura1.

When comparing Aura2 - which corresponds to the middle zone of VFC2 - with VFC2 it is clear that the bandwidth required by the former is closer to that of the latter than Aura1 was. Again, it is with no surprise that we verify these results and, once more, it is important to understand that a different configuration of VFC2 could produce opposite results.

Finally, the information in Figure 5.6 shows, as we mentioned before, that the resulting bandwidth of Aura3 is higher than VFC2. More importantly, these results happen despite the fact that Aura3 and VFC2 cover the same exact area of the virtual world. This is possible due to the high flexibility VFC exhibits, which allows it to be easily tuned with unlimited configuration possibilities.

Table 5.1 contains some examples of different VFC configurations that have the same or similar

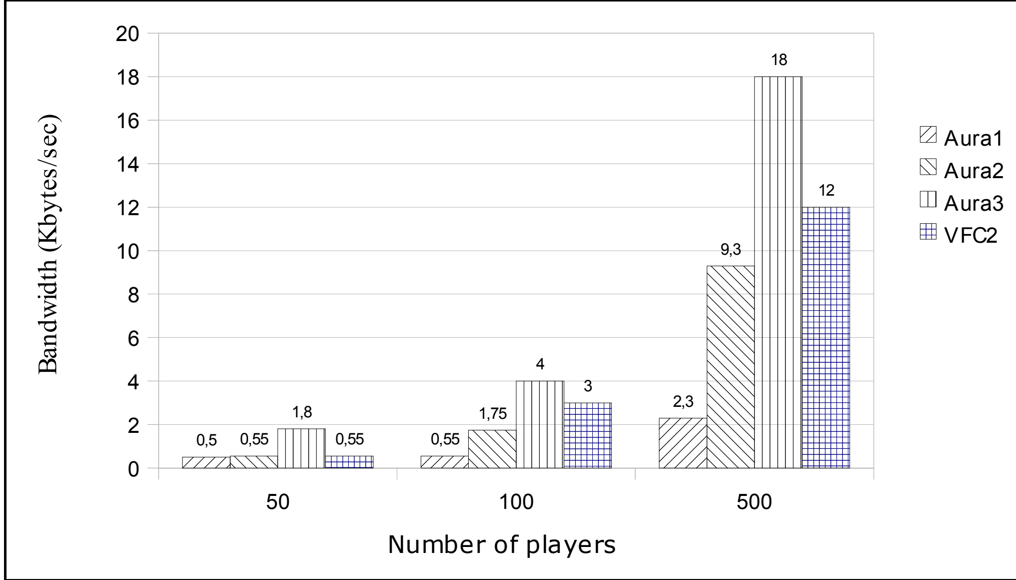


Figure 5.6: Client side bandwidth requirements: auras versus VFC.

bandwidth performance as Aura3 for 100 players in a 1000x1000 map. These results are even more enlightening knowing that the bandwidth required when using an aura with radius of 200 is about 5760 bytes. These results make the flexibility of VFC evident. We presented a few configurations just to exemplify, but there are many other possibilities. Furthermore, besides this parametrization malleability, another strength of VFC is that it can emulate other IM schemes obtaining the same or even better results.

5.2 Architectural evaluation

To evaluate the architectural component of our system we compared it (using our simulation infrastructure) with a centralized (VFC's original architecture) and a replicated architecture. In our evaluation we varied both the number of clients and the number of servers for each of the two distributed architectures (VFCLS and the replicated one). Table 5.3 describes the settings used for evaluation. Hereafter we refer to the different architectures by the names defined in the table. In our evaluation we simulated clients with the following VFC specification: three consistency zones [120, 200, 500] and respective consistency vectors κ = [(3,0,0), (10,5,100), (50,10,500)]. The virtual world consisted of a 5000 x 5000 map.

Table 5.2: Different VFC configurations that yield the same bandwidth as Aura3.

	Zones	Kvec	Bandwidth (Bytes/sec)
1	[40, 250]	{[3,0,0], [50,10,500]}	2150
2	[40, 150, 350]	{[3,0,0], [50,10,500], [80,20,500]}	2800
3	[80, 200]	{[3,0,0], [50,10,500]}	2315
4	[80, 150, 250]	{[3,0,0], [50,10,500], [80,20,500]}	2640

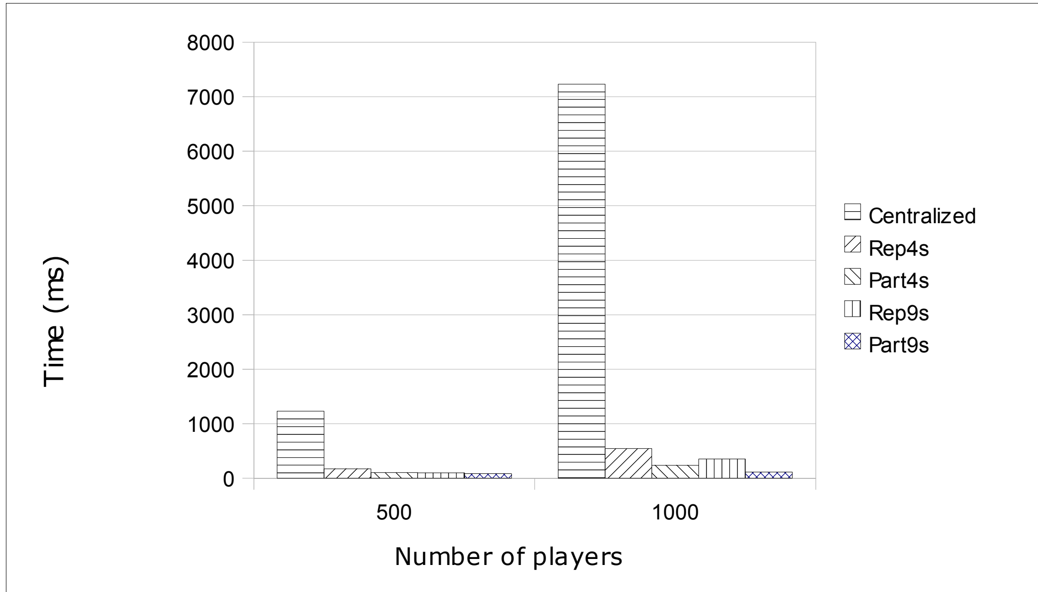


Figure 5.7: Execution times of function round-trigger for the different architectures.

5.2.1 Performance

We start by presenting the results regarding the performance of our system. To measure performance we analyze the execution time of function *round-triggered* of VFC. This way, we can also analyze the impact of the performance of our system on the game's playability - the more often a system is able to update its clients, the more interactive the game is.

Figure 5.7 shows the results of this evaluation setting considering 500 and 1000 players. The most important (and only) thing that we can conclude by the observation of the Figure is the infeasibility of single server architectures for large scale environments and the necessity of employing distributed alternatives. It is possible to see that for a larger environment of 1000 players the single server cannot guarantees the desired levels of interactivity and is only able to update clients every 7 seconds. This, obviously, results in a poor gaming experience.

To analyze the differences between the two distributed architectures (our VFCLS system and the simulated replicated system) we must take a look at Figure 5.8 that shows the same results as Figure 5.7 but without including the results of the centralized implementation.

The analysis of the Figure lets us see that our VFCLS prototype outperforms the replicated architecture considering both the 500 player context and the 1000 players context. Considering the 500 players setup, VFCLS with four servers only outperforms the equivalent four server replicated architecture, while

Name	Description
Centralized	Single server centralized architecture
Rep4s	Replicated architecture with four servers.
Rep9s	Replicated architecture with nine servers.
Part4s	Four servers VFCLS
Part9s	Nine servers VFCLS

Table 5.3: Description of the evaluated architectures.

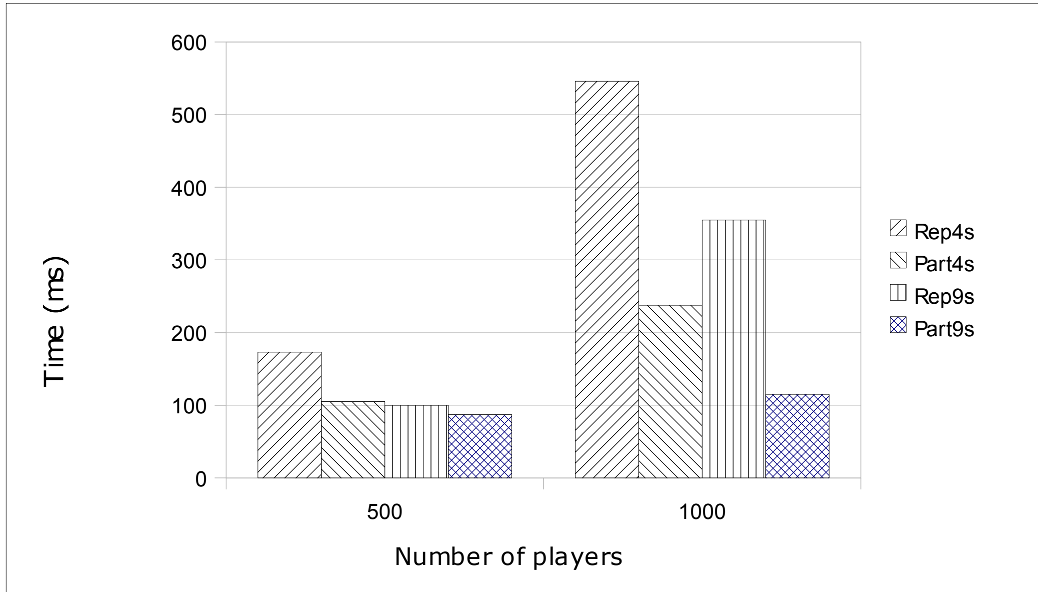


Figure 5.8: Execution times of function round-trigger for the different architectures: VFCLS and replicated architecture highlight.

the nine servers replicated system achieves better results than *Part4s*, but still worse than the nine server VFCLS. The differences between the four architectures, however, are not meaningful, and either of them could provide a good interactive experience to its 500 players.

It is by analyzing the 1000 players context that the performance of VFCLS stands. The graphics show that both the four and the nine server VFCLS setup outperform the two replicated architectures, with each reducing by more than half the execution times of their replicated equivalent (in terms of number of servers). Moreover, both can still provide a highly interactive to its users, as the execution times are low. It is also encouraging to see that the difference between the execution times of *Part9s* in the the 500 and 1000 player scenarios is not meaningful, which indicates that VFCLS has potential in what concerns scalability. To confirm this, however, it would be necessary to perform more stress testing, with a larger virtual world and a higher number of players and servers. However, due to the limitations of the available hardware, those tests were not possible to execute.

5.2.2 Server To Client Bandwidth

In this section we analyze the bandwidth required for a server to enforce VFC to its clients. The results regarding bandwidth are shown in Figure 5.9. At first glance it looks like the performance of VFCLS regarding bandwidth is poorer than the performance of the replicated architecture: with 500 players *Part4s* is the setup that requires more bandwidth; with 1000 players, the more bandwidth demanding is *Part9s*. However, to fully understand the information of Figure 5.9 it is necessary to also analyze the results of the previous performance analysis.

Because VFCLS (both the four and the nine servers configuration) achieves faster execution times of *round-triggered* function, it is able to issue an higher number of rounds messages per second. As a result, it performs VFC enforcement more often than the other (slower) architectures. For instance, *Rep4s* can only perform consistency enforcement about once per second in the 1000 players context, while *Part9s* can do so almost five times (considering that rounds are issued every 100 milliseconds, as was the case of

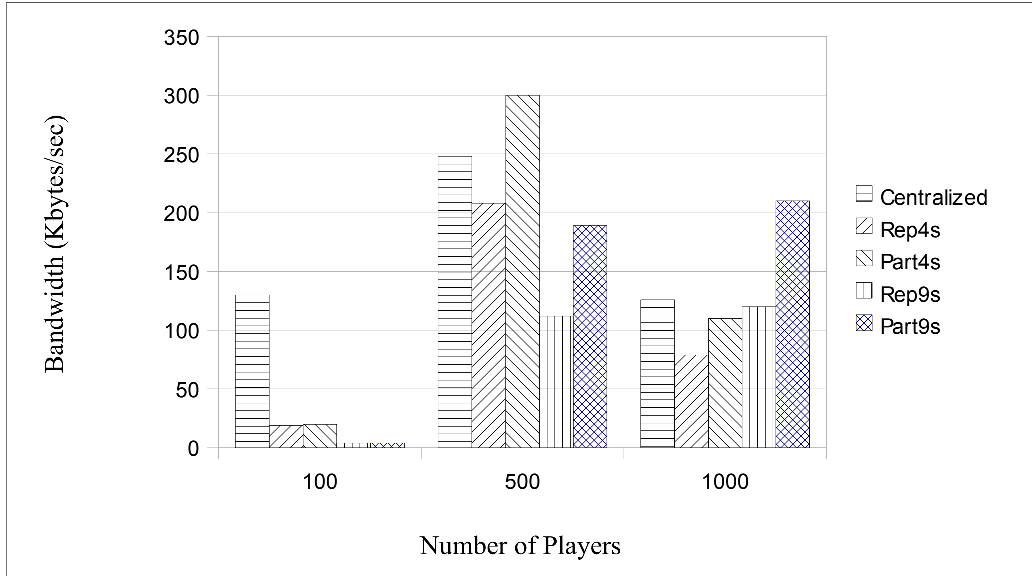


Figure 5.9: Per-server bandwidth requirements of the different architectures.

our testing). Therefore, VFCLS is able to send messages more often to its players, which results in the higher bandwidth requirements. However, this is not a drawback of our system; instead, it means that VFCLS can provide a highly interactive experience that the replicated architecture cannot.

5.2.3 Inter-server Communication

Tables 5.5 and 5.4 show the average number of message exchanged between servers in a game context with 100 players. Table 5.4 shows the average number of player updates received by a server, along with the average number of server synchronization messages received and the average number of servers involved in the process, with the exception of the server that triggers synchronization for replicated architectures. Table 5.5 shows the messages exchanged between VFCLS server: object subscription, player transfer, server synchronization and the average number (NServers) involved in it, along with the average number of player update messages received by a server. The results were obtained during a five minute simulation.

Replicated	Player updates	Server Synchronization	Number of Synchronizing Servers
4 Server	95357	285747	3
9 Server	26036	230967	8

Table 5.4: Server communication in replicated architectures.

By looking at the tables it is possible to see that VFCLS, due to its partitioning approach and the subscription protocol, limits the propagation of player update messages between servers. Furthermore, it is possible to see that the network overhead caused by transfer and subscription messages is not very high.

VFCLS	Player Updates	Server Synchronization	Number of Synchronizing Servers	Subscriptions	Transfers
4 Server	93277	104671	1.12	213	115
9 Server	25235	37373	1.27	301	167

Table 5.5: Server communication in VFCLS.

5.3 Summary

In this Section we presented the results of the evaluation of our system. We first analyzed the VFC model by comparing it to a similar Interest Management model and then presented some results about the evaluation of the architectural component of our work.

In this work we performed only a preliminary study of our architecture, leaving some important aspects neglected. First, we tested the system using a limited AOI radius that could only cross a server's neighbor partitions. To fully investigate the performance of our system it necessary to design other test cases that consider larger AOIs and larger number of users. Furthermore, in this work we have not tested the behavior of our system under hotspots. It certain, however, that the results of such a test case would not be satisfactory, because our system does not, currently, perform state repartition to respond to load peaks at the servers.

Chapter 6

Conclusion

Massively multiplayer online games (MMOGs) are played by thousands of world wide distributed users. To accommodate these large environments, game infrastructures are required to provide high availability, performance and scalability. In this document we have discussed how current approaches, both commercial and academic, try to achieve these requirements. We focused on system architectures and interest management techniques currently employed because we consider these two aspects of a game's infrastructure to be fundamental to fulfill user requirements.

In this work we proposed VFCLS, a distributed client/server architecture to support MMOGs. Our solution dynamically partitions the virtual world among several servers, allowing players to freely interact with each other, regardless of the partition they are located in. Our system also allows players to move between partitions (and, hence, servers) transparently. We use Vector-field Consistency (VFC) to reduce the bandwidth required both for servers and clients. Contrary to current interest management strategies, VFC does not impose abrupt changes on a player's area of interest. Instead, it gracefully degrades the player's view as the its distance to objects increases, similarly to human vision.

Servers in our architecture are arranged in a peer to peer network in which each server only has a partial view of the complete server network. We designed a subscription system that reduces server communication by ensuring that a server only knows and interacts with other servers when it is strictly necessary to allow interaction between players located on different partitions.

To test our system we developed a flexible simulation infrastructure that can simulate different architectural settings and interest management models as well as different types of game clients. We used this infrastructure to test VFCLS and compare it to other solutions, both regarding system architectures and interest management. Our results show that VFC can match and even outperform other interest management schemes while improving a game's playability at the same time. Preliminary evaluation of the architectural component of VFCLS also show that our system is a feasible distributed multiplayer game infrastructure.

6.1 Future Work

In this work we have established the grounds for wide variety of future works, including extensions and improvements to the current design and implementation. In this subsection we enumerate some of the

ideas for future work that arose during the dissertation:

- Currently, modifications to the dimensions of a server's partition occur only as a result of a split due to a new server joining. However, a fully dynamic and dependable system should be able to reorganize its layout by dynamically repartitioning a partition as needed, among the existing servers. This way the system is would be able to automatically (i.e., without requiring human assistance) adapt to hotspots and sporadic server load peaks. As future work, we intend to design an efficient state repartition algorithm that can monitor the load on server, identify when a server is overload and automatically respond to it by dividing the overloaded server's partition with one or more of the existing servers of the network;

- In our design of VFCLS we have not addressed fault tolerance. As a result, if a server crashes its data is lost and the players on its partition are unable to play. We intend to add fault tolerance mechanisms to VFCLS so that if a server crashes another one can, automatically, take its place and the remaining servers can update their partial view with the new disposition of the network;

- We also intend to study the feasibility of combining partitioning and replication to achieve the best of both systems - improved performance due to reduced response times (replication) and high scalability due to the effective division of players between partitions;

- As we mentioned before, another objective of ours is to endow VFCLS with an explicit conflict management mechanism that can efficiently detect and resolve conflict;

- Due to time constraints we did not address willing server departure. In a system like ours, a server may have to leave the system for a varied number of reasons (e.g., because it requires maintenance, or because it is required to perform another task not related to the game). In the future, we intend to allow servers to willingly leave the network and transfer its game state to another existing server with a minimum impact to the network;

- Expand VFC/VFCLS to other application contexts, like cooperative editing;

- Study how the subscription protocol can use the objects' consistency requirements to reduce the frequency of server synchronization.

Bibliography

[1] R. Alonso, D. Barbara, and H. Garcia-Molina. Data caching issues in an information retrieval system. *ACM Trans. Database Syst.*, 15(3):359–384, 1990.

[2] P. A. Alsberg and J. D. Day. A principle for resilient sharing of distributed resources. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, pages 562–570, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[3] S. Androutsellis-Theotokis and D. Spinellis. A survey of peer-to-peer content distribution technologies. *ACM Comput. Surv.*, 36(4):335–371, 2004.

[4] M. Assiotis and V. Tzanov. A distributed architecture for mmorpg. In *NetGames '06: Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*, page 4, New York, NY, USA, 2006. ACM.

[5] R. K. Balan, M. Ebling, P. Castro, and A. Misra. Matrix: Adaptive middleware for distributed multiplayer games. In *Middleware*, pages 390–400, 2005.

[6] T. Beigbeder, R. Coughlan, C. Lusher, J. Plunkett, E. Agu, and M. Claypool. The effects of loss and latency on user performance in unreal tournament 2003®. In *NetGames '04: Proceedings of 3rd ACM SIGCOMM workshop on Network and system support for games*, pages 144–151, New York, NY, USA, 2004. ACM.

[7] A. Bharambe, J. Pang, and S. Seshan. Colyseus: a distributed architecture for online multiplayer games. In *NSDI'06: Proceedings of the 3rd conference on Networked Systems Design & Implementation*, pages 12–12, Berkeley, CA, USA, 2006. USENIX Association.

[8] Blizzard Entertainment. World of warcraft. `http://www.worldofwarcraft.com`.

[9] J.-S. Boulanger, J. Kienzle, and C. Verbrugge. Comparing interest management algorithms for massively multiplayer games. In *NetGames '06: Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*, page 6, New York, NY, USA, 2006. ACM.

[10] W. Cai, P. Xavier, S. J. Turner, and B.-S. Lee. A scalable architecture for supporting interactive games on the internet. In *PADS '02: Proceedings of the sixteenth workshop on Parallel and distributed simulation*, pages 60–67, Washington, DC, USA, 2002. IEEE Computer Society.

[11] V. Cardellini, E. Casalicchio, M. Colajanni, and P. S. Yu. The state of the art in locally distributed web-server systems. *ACM Comput. Surv.*, 34(2):263–311, 2002.

[12] V. Cardellini, M. Colajanni, and P. S. Yu. Dynamic load balancing on web-server systems. *IEEE Internet Computing*, 3(3):28–39, 1999.

[13] J. Chen, B. Wu, M. DeLap, B. Knutsson, H. Lu, and C. Amza. Locality aware dynamic load management for massively multiplayer games. In K. Pingali, K. A. Yelick, and A. S. Grimshaw, editors, *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (10th PPOPP'2005), ACM SIGPLAN Notices*, pages 289–300, Chicago, IL, USA, June 2005.

[14] E. Cronin, A. R. Kurc, B. Filstrup, and S. Jamin. An efficient synchronization mechanism for mirrored game architectures. *Multimedia Tools Appl.*, 23(1):7–30, 2004.

[15] J. Dollimore, T. Kindberg, and G. Coulouris. *Distributed Systems: Concepts and Design (4th Edition) (International Computer Science Series)*. Addison Wesley, May 2005.

[16] Electronic Arts. Ultima online. http://www.uoherald.com/news/.

[17] T. A. Funkhouser. Ring: a client-server system for multi-user virtual environments. In *SI3D '95: Proceedings of the 1995 symposium on Interactive 3D graphics*, pages 85–ff., New York, NY, USA, 1995. ACM.

[18] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pages 173–182, New York, NY, USA, 1996. ACM.

[19] T. Hampel, T. Bopp, and R. Hinn. A peer-to-peer architecture for massive multiplayer online games. In *NetGames '06: Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*, page 48, New York, NY, USA, 2006. ACM.

[20] Id Software. Quake. http://www.idsoftware.com/games/quake/quake3-arena/.

[21] P. Kabus, W. W. Terpstra, M. Cilia, and A. P. Buchmann. Addressing cheating in distributed mmogs. In *NetGames '05: Proceedings of 4th ACM SIGCOMM workshop on Network and system support for games*, pages 1–6, New York, NY, USA, 2005. ACM.

[22] N. Krishnakumar and A. J. Bernstein. Bounded ignorance: a technique for increasing concurrency in a replicated system. *ACM Trans. Database Syst.*, 19(4):586–625, 1994.

[23] D. Kushner. Engineering everquest. *IEEE Spectrum Magazine*, 2005.

[24] M. Kwok and J. W. Wong. Scalability analysis of the hierarchical architecture for distributed virtual environments. *IEEE Trans. Parallel Distrib. Syst.*, 19(3):408–417, 2008.

[25] H. Lu. Peer-to-peer support for massively multiplayer games. In *INFOCOM*, 2004.

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

[27] T. Minoura and G. Wiederhold. Resilient extended true-copy token scheme for a distributed database system. *IEEE Trans. Softw. Eng.*, 8(3):173–189, 1982.

[28] G. Morgan, F. Lu, and K. Storey. Interest management middleware for networked games. In *I3D '05: Proceedings of the 2005 symposium on Interactive 3D graphics and games*, pages 57–64, New York, NY, USA, 2005. ACM.

[29] K. L. Morse. Interest management in large-scale distributed simulations. Technical Report ICS-TR-96-27, University of California, Irvine, Department of Information and Computer Science, July 1996.

[30] B. C. Neuman. Scale in distributed systems. In *Readings in Distributed Computing Systems*, pages 463–489. IEEE Computer Society Press, 1994.

[31] D. Pittman and C. GauthierDickey. A measurement study of virtual populations in massively multiplayer online games. In *NetGames '07: Proceedings of the 6th ACM SIGCOMM workshop on Network and system support for games*, pages 25–30, New York, NY, USA, 2007. ACM.

[32] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 161–172, New York, NY, USA, 2001. ACM.

[33] A. I. T. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware '01: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, pages 329–350, London, UK, 2001. Springer-Verlag.

[34] Y. Saito and M. Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1):42–81, 2005.

[35] N. Santos, L. Veiga, and P. Ferreira. Vector-field consistency for ad-hoc gaming. In *Middleware*, pages 80–100, 2007.

[36] J. Smed, T. Kaukoranta, and H. Hakonen. Aspects of networking in multiplayer computer games, 2005.

[37] Sony Online Entertainment. Everquest. `http://everquest.station.sony.com/`.

[38] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.*, 11(1):17–32, 2003.

[39] D. Stutzbach and R. Rejaie. Understanding churn in peer-to-peer networks. In *IMC '06: Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pages 189–202, New York, NY, USA, 2006. ACM.

[40] J. Waldo. Scaling in games and virtual worlds. *Commun. ACM*, 51(8):38–44, 2008.

[41] S. D. Webb, S. Soh, and W. Lau. Enhanced mirrored servers for network games. In *NetGames '07: Proceedings of the 6th ACM SIGCOMM workshop on Network and system support for games*, pages 117–122, New York, NY, USA, 2007. ACM.

[42] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding replication in databases and distributed systems. In *Proceedings of* 20th *International Conference on Distributed Computing Systems (ICDCS'2000)*. IEEE, 2000.

[43] H. Yu and A. Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *OSDI'00: Proceedings of the 4th conference on Symposium on Operating System Design & Implementation*, pages 21–21, Berkeley, CA, USA, 2000. USENIX Association.