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

André Negrão
GSD - INESC-ID
Technical University of Lisbon
(Adviser: Professor Paulo Ferreira)
(Co-adviser: Professor Luís Veiga)

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. Game users require high performance, availability and scalability in order to maximize their gaming experience. Current commercial approaches meet this requirements by confining users to isolated partitions provisioned by powerfull server clusters. In this work, we present a different approach to virtual world partition in which players are allowed to freely interact and move around the game map. We use the Vector-Field Consistency model to reduce the bandwidth requirements imposed both on the servers and clients of the system. Our preliminary results show that our system is able to handle a large number of user in a more efficient manner than other types of architectures.

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[16]. 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[4, 15, 7].

Supporting this new form of game playing is a challenging task. First of all, high performance levels are required in order to provide the highly interactive experience demanded by players. Second, these games must scale to an increasingly large number of users and game environment. Third, it is fundamental to provide constant availability, so users can play whenever they want 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 scalability restrictions imposed by hardware limitations a game's virtual world is either duplicated or statically partitioned into several mini worlds[13], assigning each duplicate/partition to a different, independent server.

Although widely used, this approach presents several obvious drawbacks. First, with these strategies users are confined to a single server at each moment and are unable to 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., door, portal, tunnel) specially designed for that purpose. Scalability is, therefore, achieved at the expense of user interactivity.

This paper describes our work aiming at designing a scalable, efficient and highly available distributed infrastructure to support MMOGs that does not rely on the static partitioning schemes used by commercial games to achieve scalability. Instead, we prescribe an architecture in which the game is presented to its players as a single large virtual world where players can freely interact, without further limitations than those imposed by the logics of the game.

In the past years, several research/academic works have proposed both peer-to-peer(P2P) [8, 3, 10] and distributed client/server [2, 5, 1, 9, 6] alternatives to the commercial approach. In P2P systems the game management is handled by the players themselves who directly exchange update messages with each other, offering computational power, bandwidth and adaptability without the need for dedicated servers. However, P2P approaches depend completely on game users, whose resources are scarce when compared to dedicated servers. As a result, bottlenecks can arise and, therefore, P2P systems can not guarantee the demanded performance levels.

In Client/server (C/S) solutions, on the other hand, multiple servers (dedicated to the task of game management) cooperate in order to efficiently balance computational and network load. Servers collaborate by replicating[9, 6] or partitioning[2, 5, 1] the game state among each other. In replicated systems, each server holds a complete copy of the virtual world, but directly manages only a subset of the total number of players. Partitioned systems, on the other hand, divide players by partitioning the virtual world into disjoint regions, each assigned to a different server. In both 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, contrary to the commercial approach.

Regardless of the architecture, these systems rely on Interest Management[12] techniques to improve performance. IM minimizes bandwidth requirements by sending to clients only updates to objects that are within the players *area of interest* (AOI). An area of interest can be, for instance, the area surrounding the player[2, 5]. Although these strategies considerably reduce bandwidth, they follow an all-or-nothing approach that fails to capture the user's interest: every update to objects within an AOI is sent to the player and none of the ones outside is. As a result, players may experience sudden visibility losses and end up seeing objects abruptly appearing (disappearing) on (from) their area of interest.

A different approach was proposed by the Vector-field Consistency (VFC)[14] 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. With this, it is able to eliminate the abrupt visibility loss characteristic of other IM strategies by gracefully degrading the player's view. However, because it was designed for small wireless mobile network games, VFC employs a centralized architecture that greatly limits its scalability.

In this work we propose VFCLS, a distributed client/server architecture that uses as its interest management scheme a version of VFC modified to achieve improved scalability. In our system, the virtual world is partitioned into different, but not independent, regions handled by distinct servers. Players can move around the virtual world, transparently switching between regions of different servers and are able to seamlessly interact with players located in other regions. The server organization enforces VFC through a subscription protocol that allows servers to apply VFC for players located in other, contiguous or not, regions.

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.

This paper is organized as follows. In section 2 we overview the current state of the art in multiplayer game supporting architectures and interest management. In section 3 we describe the Vector-field Consistency model in more detail. Next, in section 4, we describe the architecture of our system and in section 5 we present details about its implementation. In section 6 we present the results of the evalution of our system and then we finish with section 7 where we summarize our work and introduce our ideas for future work.

2. RELATED WORK

In a massively multiplayer online game (MMOG), a large number of players interact through an extensive virtual world, shared over a wide area network. 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 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 (NPCs[1]). 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'.

2.1 Peer to Peer Game Support

P2P support for multiplayer games has been an active research topic[8, 3, 10] in the past years. In these systems, clients exchange state updates directly with each other, instead of doing so through a server.

SimMud is a P2P game[10] that 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.

Colyseus is another example of a P2P game infrastructure[3]. 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.

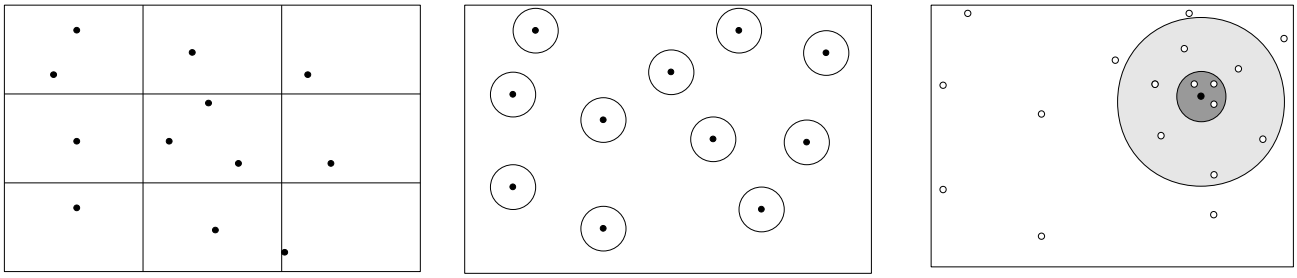
P2P systems put the burden of managing the virtual world and maintaining it consistent on peers that are executed on the players' machines. These machines are considerably limited in both bandwidth and computational power when compared to the dedicated servers used on C/S games. As the number of players 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.

2.2 Partitioned vs Replicated Client/Server

Alternatively to the P2P approach, distributed client/server (C/S) systems use multiple dedicated machines to mediate the game played by client applications. In these systems, the task of managing the game state and updating clients is a responsibility of the servers and clients are left to simply play the game. According to their approach to load balancing, a distributed client/server multiplayer game can be classified as *partitioned* or *replicated*.

In a replicated client/server system[9, 6] (commonly referred to as "mirrored server architecture") each server holds a copy of the complete game world, but is only responsible for a

[1]"Non-player Characters"



(a) Avatars scattered across a virtual world with region based IM

(b) Avatars and their respective auras.

(c) Consistency zones around a pivot.

Figure 1: Different IM schemes.

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 the server they were assigned to.

Alternatively to the replicated approach, partitioned systems achieve load distribution by dividing the virtual world into regions/partitions assigning each to a different server of server network[2, 5, 1]. Although the virtual world is logically divided into partitions, these systems provide mechanisms to make that division unnoticeable to the player, giving him the illusion of being in a large single virtual world.

2.3 Interest Management

IM is motivated by the observation that players are not equally interested in all objects (other players, items,...). Usually, they are more concerned about objects located closer to 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[11]:

- RegionsRegion based IM is the most straightforward approach to interest management (Figure 1(a)). In this approach 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.
- Auras 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 an avatars's aura the player can see it. These carachteristics 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.

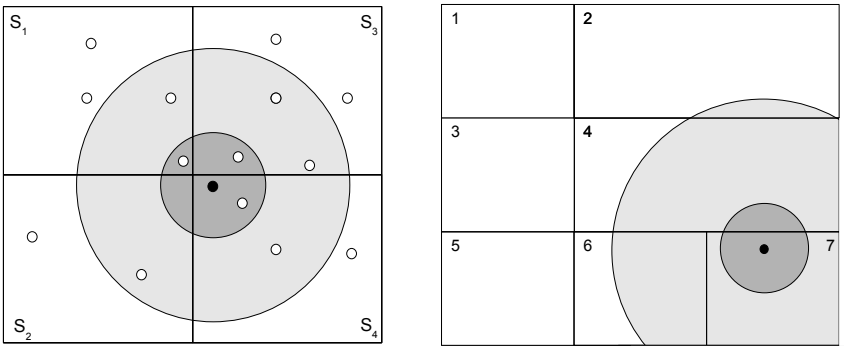
Despite providing bandwidth reduction, current IM mechanisms are too rigid - updates in an area of interest (region or aura) are visible, all others are not. This inflexibility can lead to some undesired game situations. Consider, for instance, a player moving in an open scenario. Because consistency 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[9], resulting in a cut in performance due to increased bandwidth and computation requirements.

Alternatively to the common IM strategies, Vector-Field Consistency (VFC) [14] defines multiple *consistency zones*, each with a different *consistency degree* (Figure 1(c)) that weakens as the distance to the player increases. Each degree defines the maximum divergence allowed between the player's view of an object and the object's actual state. Unlike the other IM schemes, *consistency degrees* are defined based on three criteria: the time elapsed since the last time the object was updated; the number of updates to an object that were not sent to the player; and an application-specific function that captures the difference between the state of the object on the player's view and its actual state. As a result, a VFC AOI can cover a larger area while avoiding bandwidth increase and is, thus, able to efficiently tackle the shortcomings of auras and regions. However, VFC was designed for mobile had-hoc networks as a centralized model, executed by a single server, which is not suitable for large scale networks.

3. VFC: LARGE SCALE

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)[14] 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 "Large Scale Vector-Field Consistency" (VFCLS).



(a) Even partitions. (b) Partitions with different dimensions.

Figure 2: VFCLS main concepts: Virtual world partition and Vector-Field Consistency.

Our approach to achieve scalability consists in partitioning the virtual world into dynamically sized rectangular regions/partitions and assign each one to a different server of a distributed server network. Clients (players) are assigned to one of those servers 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.

Servers in our infrastructure are organized in a peer to peer network and are connected to the servers responsible for their direct neighbor partitions. They may also need to know and communicate with servers that are not their direct neighbors. This is necessary because a player's area of interest (AOI) 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 favours 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.

3.1 System Architecture

VFCLS comprises the following 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".

Client Manager. The Client Manager administers client data and enforces VFC through two components, the "Session Manager" and 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 protocol, which was extended in order to support inter-partition interaction.

3.2 Consistency Model

VFCLS represents the virtual world as an N-dimensional space populated with game objects (e.g., avatars, items, NPCs). Each game client of VFCLS has a local *view* of the virtual world, comprising relevant objects, that can have bounded inconsistencies with relation to their primary replicas, which are distributed between the servers. In each view, the consistency of objects depends on their distance to a *pivot* object (e.g., the player's avatar).

Pivots are associated with *consistency zones* – concentric, ring shaped areas defined around the pivot object – 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. As a result, an object located closer to the pivot is required to be more often refreshed.

Consistency degrees are defined by 3-dimensional *consistency vectors* (κ) that bound the maximum divergence of objects inside a *consistency zone* with relation to their primary replica. Each dimension corresponds to a scalar value that bounds divergence regarding the following criteria:

- *Time* (θ): Specifies the maximum time (in seconds) an object can stay without being refreshed with its primary replica's latest value (staleness).
- *Sequence* (σ): Specifies the maximum number of updates to the primary replica that are allowed not to be

applied to the object (missing updates).

- *Value* (ν): Specifies the maximum divergence between the contents of the local copy of an object and its primary replica (divergence impact). *Value* is an application-dependent criterion that defines the maximum percentage difference between an object and its primary replica. *Value* is calculated by a function specially defined by the application's programmers or automatically calculated for numerical data.

3.2.1 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. This approach contrasts with the limiting strategies tipically used by commercial games.

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 minimizing server synchronization requirements we designed a subscription protocol in which servers communicate only when strictly necessary, to ensure the correct application of VFC.

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. We consider that two players interact when one is whithin the other's AOI, whether they are actually (at the application level) interacting or not. Because VFC allows game programmers to freely define consistency zones, we understand that, in our system, a player's AOI can cross multiple partitions, including non-neighbor ones. 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 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 player's 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.
2. After all checking is done 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 their 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 the 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 is explained in section 3.3, servers apply VFC both to their *owned* (i.e., the objects located on the server's partition) and *subscribed* (i.e., the objects located on other partitions, but that have subscribed to the server) players. 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 the server 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.

3.2.2 Player Transfer Protocol

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 transfer a player's data before the player actually moves to a new partition. 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.

The actual transfer of a player from his current server S_i to the new 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 transfer notification.

3.3 Distributed Vector-Field Consistency

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

Our Distributed VFC algorithm ensures that players receive updates even from players located on different partitions by having servers enforcing VFC for both their owned and subscribed objects, although considering only their owned objects when checking which objects are within a pivot's *consistency zones*. Considering Figure 2(a), 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 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.

3.3.1 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, from a player, then it concerns an *owned object*. Hence, the server updates its information about each player's (considering both owned and subscribed) current state of consistency to reflect the update received.

- 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.

3.3.2 Updating Clients

The process of updating clients 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. After that, 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 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.

4. IMPLEMENTATION

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 far as VFCLS is concerned, the virtual world is a bounded area populated with DataUnits. A DataUnit (DU) is an object that representes 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. 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.

4.1 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 diferent users.

As explained in the previous section, 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 Distributed Vector-Field Consistency

Our distributed version of Vector-Field Consistency inherited and extended many of the original data structures used by VFC. 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 reference to the objects owned by the player.

To maintaing information of 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.

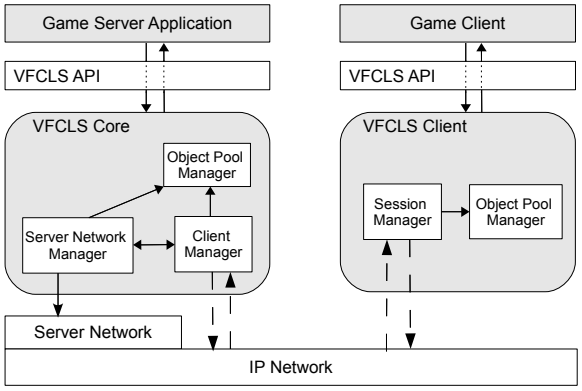


Figure 3: VFCLS integration with game applications

4.3 Game Programming Interface

Figure 3 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.

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 pottentially (depending on the game's design) register objects corresponding to computer controlled characters (NPCs).

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.

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.

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.

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 three zones with radius [40, 80, 120] and respective K vectors [3,0,0], [10,10, 0] and [50,10,500]

Table 1: Description of the different parameters variations

5. EVALUATION

To evaluate our system we designed and implemented a 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 the objects is 200 bytes. The tests were performed on two machines equiped 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.

5.1 Interest Management Evaluation

Table 1 describes the parameter variation used on the experimentation. By looking at the table we can see that the radius of Aura1, Aura2 and Aura3 are, respectively, 40, 80 and 120 and that VFC1 has three consistency zones with radius also with radius 40, 80 and 120. Figure 4 illustrates the differences between these configurations.

Figure 5 shows the bandwidth differences between Aura1, Aura2, Aura3 and VFC1 in a context with variable number of players - 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. In VFC, on the other hand, varying the radius of consistency zones does not necessarily means that the bandwidth spent 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

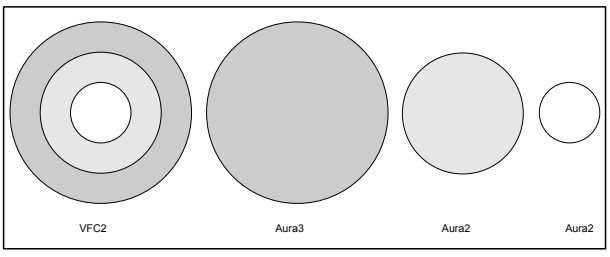


Figure 4: AOI size and variations.

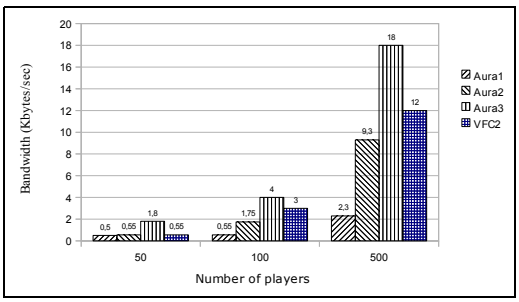


Figure 5: Client side bandwidth requirements: auras versus VFC.

or no impact on bandwidth, although at the cost of fidelity.

Observing figure 5 we can also see that VFC1 only performs better than Aura3. To fully understand the meaning of these results, however, we have to analyze them in light of figure 4. One of the things it lets us know is that Aura1 corresponds to the inner zone of VFC1. Hence, it is only normal that the resulting bandwidth of VFC1 is higher than the one generated by Aura1. It should be noted, however, that VFC1 corresponds to an example VFC consistency vector defined for our experiments. As we mentioned before, a different consistency vector could perform better than VC1 and possibly yield bandwidth results similar to Aura1.

Finally, the information in figure 5 shows that the resulting bandwidth of Aura3 is higher than VFC1. More importantly, these results happen despite the fact that Aura3 and VFC1 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.

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 2 describes the settings used for evaluation. In our evaluation we simulated clients with the following VFC specification: three consistency zones [120, 200, 500] and respective consistency vectors $\kappa = [(3,0,0), (10,5,100), (50,10,500)]$. The virtual world consisted of a 5000 x 5000 map.

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 2: Description of the evaluated architectures.

5.2.1 Performance

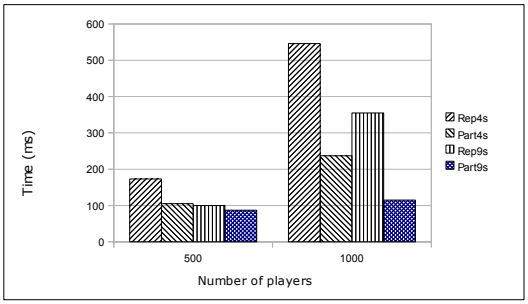


Figure 6: Execution times of function round-trigger for the different architectures: VFCLS and replicated architecture highlight.

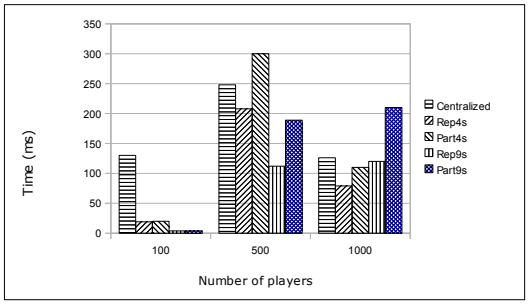


Figure 7: Bandwidth requirements of the different architectures.

To measure performance we analyse the execution time of function *round-triggered* of VFC. This way, we can also analyse 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.

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 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 meaningfull, 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 significative, which indicates that VFCLS has potential in what concerns scalability. To confirm this, however, it would be necessary to perform more testing, with a larger virtual world and a higher number of players and server. However, due to the limitations of the available hardware, those test were not possible to execute.

5.2.2 Server To Client Bandwidth

The results regarding bandwidth are shown in figure 7. 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 7 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 pers 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 miliseconds, as was the case of 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.

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 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 achieve user requirements.

In this work, we propose a distributed client/server architecture to support MMOGs. Our solution partitions the virtual world among several servers, allowing players to freely and transparently move between regions/servers. 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 visibility similarly to the human sight.

7. REFERENCES

[1] 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.

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

[3] 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.

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

[5] 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.

[6] 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.

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

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

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

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

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