VFC4FPS - Vector-Field Consistency for a First Person Shooter Game

Bruno Loureiro
bruno.loureiro@ist.utl.pt

ABSTRACT

Multiplayer online games are increasingly more popular. Keeping the game state updated and consistent among all players in soft real-time is critical. Sending the complete game state to all players does not scale with the number of players. One way to increase the game scalability is by reducing it's network traffic, and one way to reduce network traffic is by exploiting the player's sensory limits. However, current solutions typically use an all or nothing filtering, where a player only receives updates of objects inside his sensory zone. In this work we use the Vector-Field Consistency. VFC offers a progressive consistency reduction. They do this by using multiple zones, each with a set of consistency requirements, which are reduced with the increasing distance. We intend to obtain a network traffic reduction of at least half of the original traffic of an online game. To that effect we use the Cube 2: Sauerbraten, a First Person Shooter game. In this kind of game, players have a limited view of the virtual world. With that in mind we added the concept of Field of View to VFC in order to improve performance. Results show that is possible to significantly reduce network traffic without harming consistency and playability.

Keywords

Multiplayer Online Games, Optimistic Consistency, Interest Management, Spatial locality

1. INTRODUCTION

In recent years, the popularity of online multiplayer games has been growing rapidly. Among the reasons for the growth is the increasing penetration of broadband internet. A type of game that emerged with these new possibilities was the massively online multiplayer game (MMOG). MMOGs are characterized by high numbers of simultaneous players sharing a huge persistent virtual world.

A popular category of online games that do not fit in the size of MMOGs are the First Person Shooters (FPS). FPS have fast paced interactivity with emphasis on dexterity and reaction times of players. Compared with the hundreds or thousands of simultaneous players that participate in a MMOG, the number of players in a FPS is around the dozens (typically between 16 and 32). FPS differ from MMOGs by allowing servers to be hosted by players on their personal computers. However, the number of players supported by these servers is mainly limited by the available bandwidth. Game servers can also be housed on more powerful servers (machines) with more bandwidth, normally provided by communities of players.

In order to provide good performance, each player has local copies (replicas) of the game state of other players (player's positions, shots, etc.). Maintaining these replicas consistent in soft real time to ensure good gameplay without using too much bandwidth is the main difficulty in implementing an online game. Associated with the maintenance of game consistency is scalability, since lower communication efficiency means support for less players.

One solution for dealing with scalability is the used architecture. There are two main types of architecture: Client-Server and Peer-to-Peer (P2P)[5]. In Client-server architectures there is a central server that receives state updates from clients and that propagates these updates to other clients. In P2P architectures there is no central server, the server functions are divided among all the clients involved, which communicate directly with each other. Both architectures have hybrid variations.

In the case of consistency management, there is a technique called Interest Management(IM)[3][15]. IM allows for a reduction of the amount of state updates required. This reduction is possible through exploration of the sensory limits of the players. Each player has an area of interest (AoI) and is only interested in the state updates from objects within that area. The AoI can be based on regions, auras or line of sight visibility. This way a player is only interested, respectively, in state updates of objects within the same region, that lie within a surrounding area or who they can see.

Another technique, that reduces the frequency of messages, is Dead Reckoning[15]. This technique reduces the frequency of player's position update messages through motion prediction, taking into account past movements. It works well for reduced intervals between messages[12].

Finally, at the communication layer, we have a set of techniques used by the games directly or through libraries[5][15][6]. These techniques focus on compression, aggregation and multicast of packets in order to make communication as efficient as possible.

.

This work aims to adapt an existing IM technique and apply it to a real game. The main objective is to increase the scalability by reducing the network traffic, maintaining the playability and consistency. For this purpose we will use the Vector-Field Consistency (VFC)[14] as a basis of our solution. VFC, in contrast to other IM techniques does not apply an all or nothing filter. VFC allows the consistency to be reduced gradually as the distance increases. A secondary objective is the development of our solution in the form of a library. Thus being able to separate game logic from the details of consistency management.

A key challenge for implementing the solution with a real game lies in the need for the game to have the source code available, i.e., the game has to be open source or a commercial game for which the source code was made available. We chose a First Person Shooter (FPS) because there are a lot of games of this kind with these conditions. The FPS chosen was the Cube 2: Sauerbraten[1].

Another key challenge of this work is linked to the nature of the FPS. They offer mainly maps (virtual worlds) of small size while providing a wide range of vision. In order to increase the impact of the solution, the FPS characteristics will be taken into account. This is reflected with the inclusion of the field of view concept in VFC. Thus allowing to define different consistency requirements for objects, as they are inside or outside the field of view of the player.

This document is organized as follows. In section 2 we describe the related word, focusing on the various existing solutions to reduce bandwidth usage. Section 3 describe our system's architecture and section 4 has the implementation details. In section 5 we describe the evaluation and the obtained results. We finalize with section 6, where we summarize this work and present ideas for future work.

2. RELATED WORK

2.1 First Person Shooters

In a FPS, players compete in a virtual world through the internet. Each player controls an avatar (a digital representation of the player). Avatars normally compete against each other.

Each avatar has an associated state, characterized by attributes such as position, health level, weapons, ammunition and armour. This state is changed through interaction with other avatars, objects and through his own movement.

Since the local state of each player consists of many remote objects, a naive solution to keep them updated would be asking all players for their updated status in each frame. This solution is impractical because the communication latency is higher than the frame processing time (typically 16 ms for 60 frames per second).

In practice, games keep replicas of each client's remote objects. Games then use the local state of replicas for processing frames. Replicas may not reflect the actual state and are updated periodically based on a consistency model. Due to the fast paced nature of FPS games, players tolerate latencies up to 100ms[11].

2.2 Architectures

Architecture refers to the way nodes communicate. Nodes are divided into clients and servers. There are two main architectures. The most common and simplest is the Client-Server (figure 1). In this architecture clients communicate only with the servers. A Client-Server architecture can have multiple servers to balance the load[4]. Peer-to-Peer is another architecture (figure 2), where nodes are called peers and have client and server functionality. The game processing is divided between all nodes and communication is done directly between them.

2.3 Consistency

The replication of information keeps copies (replicas) of information on several computers. Performance is obtained due to local access of information being faster than accessing the same information remotely. However, in order to keep local replicas consistent, replication mechanisms are needed. The main difficulty is managing changes to the same replica by different clients at the same time, and how quickly that consistency is maintained in order to guarantee game playability.

Replication can be managed in two ways: pessimistic and optimistic replication[13]. Pessimistic replication locks access to the replicas during an update. This keeps all replicas consistent between each other. Optimistic replication, on the other hand, lets replicas diverge between clients. But in order to guarantee game playability it has to have a way to limit how much the replicas are allowed to diverge. Divergence can be limited by a maximum time limit[1]. In TACT[17], besides limiting the divergence through time limits, we can use the number of local updates to decide when to propagate the updates.

2.4 Interest Management

A player does not perceive the overall state of the game at once, Interest management (IM) uses these sensory limits to reduce communication. This way, objects relevant to the player are update more frequently than other objects. Interest management can be region based, aura based or visibility based.

In region based IM (used mainly in P2P architectures), players only receive updates of objects in the same region[7][16][9] (figure 3).

In aura based IM, players only receive updates of objects inside an area around the player[3] (figure 4).

In visibility based IM, players only receive updates for objects they can effectively see[3][10] (figure 5).

2.5 Dead Reckoning

Another approach to reduce the bandwidth is to send updates less often. However, it is necessary that the rate reduction is not detrimental to gameplay. Dead Reckoning[12] is used to predict the movement of players until there is a new packet. This prediction is based on previous packets.

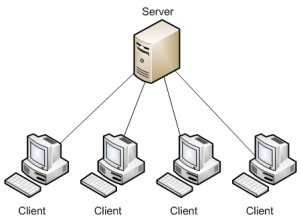

Figure 1: Example of a Client-Server architecture.

[1]Cube 2: Sauerbraten, http://sauerbraten.org/

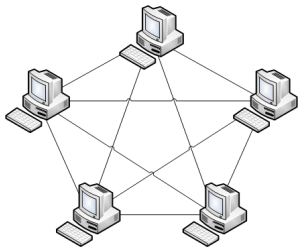

Figure 2: Example of a P2P architecture.

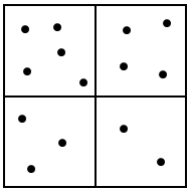

Figure 3: Example of a region based IM.

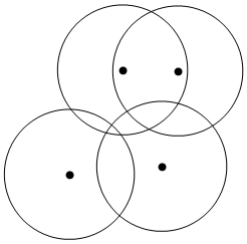

Figure 4: Example of a aura based IM.

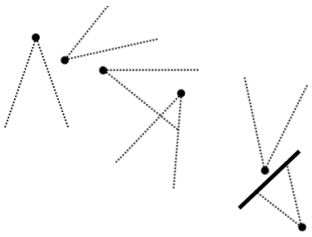

Figure 5: Example of a line of sight visibility IM.

2.6 Network Layer Techniques

There are a group of techniques that can be used at the network layer to reduce bandwidth. A server sends packets to each of it's clients. In many cases these packets are the same differing only in the destination. Using multicast[5] can greatly reduce the bandwidth by sending only one packet by multicast and still reach every client. Another way to reduce bandwidth is by optimally use each packet. Instead of sending many small packets one could aggregate[6][15] small packets in one bigger packet in order to reduce header overheads. Reducing the number of bits needed to represent information is another technique that can be used. Another technique, that can be applied to string based information consists in indexing frequent substring by use of codes. The last technique is lossless compression[5]. This can be achieved by algorithms such as Huffman or by sending only deltas (modifications) of the information.

2.7 Donnybrook

Doonybrook[10] was thought for low bandwidth environments. It uses a P2P architecture. Doonybrook aggressively explores the limited perception of the virtual world by an avatar. Each player has a bandwidth limit which is distributed between the objects in which the avatar has more attention.

2.8 RING

RING[8] uses precomputed visibility between rooms to decide which avatars are visible for each player and exchanges updates only between visible avatars. This system is good for environments with high occlusion.

2.9 A^3

A^3[2] is an algorithm that combines a circular aura with a field of view. They do this to avoid inconsistencies when an avatar turns around abruptly. The frequency with which an avatar receives updates from others is reduced linearly with an increasing distance.

2.10 Vector-Field Consistency

Vector-Field Consistency[14] introduces the concept of multiple concentric circular zones with decreasingly consistency requirements (figure 6). The VFC is an optimistic consistency model that allows replicated objects to diverge in a limited way.

The levels of consistency associated with an avatar are specified by three-dimensional vectors. Each vector κ is connected to one consistency zone. Zones are defined around a

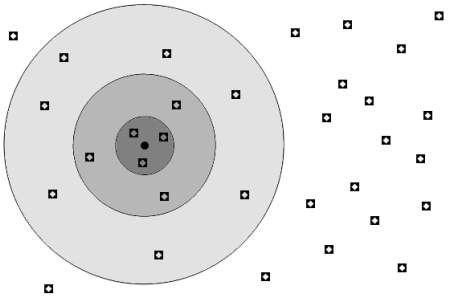

Figure 6: Example of VFC's consistency zones.

pivot. A pivot can be an avatar or other object. Objects within the same zone are subjected to the same consistency level defined by the vector κ. This vectors specify the limits of which a replica is allowed to diverge. The three dimensions of the vector are time, sequence and value.

- Time: defines the maximum time (in seconds) that a replica is allowed to diverge.
- Sequence: defines the maximum number of updates that a replica can ignore.
- Value: defines the maximum difference between the content of an object and is measured in percentage.

When one of this dimensions is exceeded means that the replica needs a new update.

3. ARCHITECTURE

3.1 Overview

We decided to use Vector-Field Consistency as the basis of our work. We chose VFC for their flexibility and progressive consistency reduction.

However, we believe that VFC can be improved to better support FPS. VFC consistency zones have the same consistency level for the 360° surrounding the avatar. However, the visibility of an avatar is limited to a field of view that is only part of the 360°.

The introduction of the field of view helps strengthen the consistency level in the avatar's field of view, while simultaneously decreases de consistency level for objects "behind" the avatar. Due to this adaptation of the VFC to the context of FPS, we named our system VFC4FPS (Vector-Field Consistency for First Person Shooters).

The game to which the VFC4FPS was applied to was the Cube 2: Sauerbraten. The reasons behind this choice were being open source, implemented in C++, popular, having large maps and an in-game map editor.

3.2 VFC Architecture

VFC was designed to be used as a library. Thus relieving the game programmer from the communication details. Through the API provided by the library, the programmer can parametrize the consistency requirements. VFC uses a Client-server architecture (figure 7).

Clients keep replicas of all shared objects (secondary object pool). The server keep the main replicas (main object pool). A client is free to read the replicas, but when a update is made to a replica it must be propagated to the server.

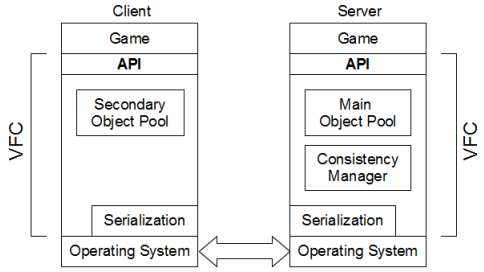

Figure 7: VFC architecture.

This propagation is not immediate. Both the client and the server send replica updates in a periodic fashion (rounds).

The server contains a consistency management block, which is responsible for deciding which updates are sent to each player at each round. This is done by a VFC function named round-triggered. Using each client view (set of consistency parameters), it decides if an object should be sent to the client.

In order to support the round-triggered function, VFC uses data structures to store the number of updates that each object has received since the last send to the player. It also stores the last time de object was sent and the value that the object had. This data is necessary to see if any of the three parameters of the κ vector was exceeded. In which case an update is needed.

3.3 VFC4FPS

VFC4FPS introduces the field of view as a new parameter on a player view. It does this by adding an array that contains the values for the angles that constitute the fields of view. The array that contained the κ vectors gains a new dimension, and is now indexed by zone and FoV in order to obtain the κ vector. This means that a zone can have, for each zone, one κ vector for each FoV.

Another modification made by the VFC4FPS was to change the units of the time dimension from seconds to milliseconds due to the high frequency with which updates are sent.

In the table 1 we can see an example of a view with three zones, three FoVs and the respective κ vectors.

3.3.1 VFC4FPS Architecture

VFC4FPS maintains the basic architecture of VFC (figure 8). One of the changes was in the round-triggered function. This function was altered to use the orientation of the pivot to determine in which FoV an object was located. This was done in addition to the zone determination. With these two determinations was then possible to obtain the κ vector with which the consistency state of an object was then compared.

Another change was the introduction of an compression module after the serialization stage in order to reduce the size of the updates.

3.3.2 Cube 2 Communication Model

Cube 2: Sauerbraten has two types of communication: object updates and events. Object updates are periodic and contains the avatar's state. Events are immediate and consists of messages that change the global state of the game.

In Cube 2: Sauerbraten all shared objects are avatars. Guns, ammo, armour and health are controlled by events. Object updates contains all the state and can be mapped to the VFC4FPS.

Events, however, can't. This is because events don't contain state, they cause changes in the global state. This fact

FoV \ Zone	150	250	∞
90°	[30, 1, 5%]	[60, 2, 10%]	[90, 3, 15%]
150°	[45, 2, 10%]	[75, 3, 15%]	[105, 4, 20%]
360°	[90, 3, 15%]	[120, 4, 20%]	[200, 5, 25%]

Table 1: Example of a view.

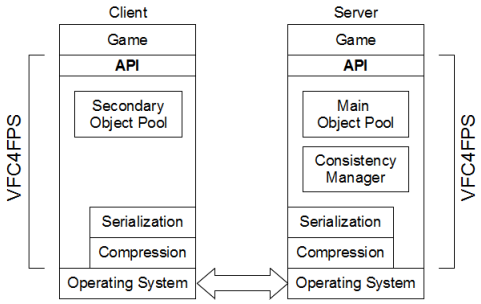

Figure 8: VFC4FPS architecture.

impedes the mapping of events to VFC4FPS. This is because events can't have it's frequency reduced, at most they can be filtered. But then, there would be clients whose global state differed from the other clients.

3.3.3 *Client Join*

When a client connects to a server, it does so like in the original game. Simultaneous, a connection is made between the VFC4FPS client and the VFC4FPS server.

The game client then registers it's avatar's object with the VFC4FPS client and it's view. These are then sent to the VFC4FPS server.

3.3.4 *Client Update*

A client processes frames many times per second (typically 60). In each frame the player's avatar state is altered due to interaction with the virtual world.

It's the VFC4FPS client's job to periodically send updates of his local replicas to the server. Objects are only sent to the server if there are changes after the last send. This avoids sending redundant information if there are no changes.

3.3.5 *Server Update*

The main function of the VFC4FPS server is to process the received updates from the clients and decide, using the round-triggered function, which updates to send to each client.

As reported earlier, events are not managed by the VFC4FPS. However, some events can be filtered. This is the case of events for shot effects and sound triggers. These events have spacial locality and are only relevant for a subset of players.

When one of these events are received by the original Cube 2 server, it calls the VFC4FPS server in order to obtain the clients to which these events are relevant, filtering these events for those who are not.

3.3.6 *View Changes*

VFC4FPS offers the possibility to change any parameter of a view. This is particularly useful in the use of zoom by a player. When a player zooms in, the field of view is reduced. This can be matched in the view. Due to the higher range of vision in this situation, the radius of the zones is duplicated to maintain gameplay. When a view is changed it is send immediately to the VFC4FPS server, so it can accurately manage the player's consistency.

4. IMPLEMENTATION

4.1 Development Environment

VFC4FPS was developed in a library form. The library was implemented in C# in the .Net platform from Microsoft[2]. Cube 2: Sauerbraten is implemented in C++. The use of the VFC4FPS functionality is done via the library's API. In order to simplify the implementation and leverage the interoperability offered by the .Net platform, the game's C++ code was compiled as managed C++. Thus allowing to directly interface the game and library.

4.2 Interfaces for Shared Objects

Not all the objects in a game have the same characteristics. For this reason VFC4FPS offers a hierarchy of interfaces that objects can implement (figure 9).

First we have the ISharedObject interface. This interface is useful for objects without spatial locality, i.e., objects that represent the overall state of the game. A ISharedObject contains a field that identifies the object (Id) and methods that allow cloning an instance of the object (clone) and determine whether two instances are equal (Equals).

Next in the hierarchy we have the IPositionableObject interface. This interface applies to objects that have spatial locality. This interface contains three fields for specifying the position X, Y and Z in the virtual world. It also contains a method to calculate the distance between two objects (DistanceBetween) and another to compare the variation of the parameter value (CompareNu) specified by the vectors *kappa*. The implementation of the method CompareNu specifies how the value is calculated.

Finally, we have the IOrientableObject interface. This interface adds the fields that define the orientation of the object (OrientationX, OrientationY, OrientationZ). The orientation must be in the form of a normalized vector. This interface is applicable to objects that have a limited view of the virtual world.

4.3 VFC4FPS Client API

AddObject Adds a new object to the secondary object pool. The object is immediately sent to the server main object pool. The object must be an instance of a derived class from one of the interfaces specified in section 4.2.

ConnectToServer Establishes a connection between the VFC4FPS client and server. In the case of a successful connection, returns a client identifier.

DelObject Removes an object that is owned by the client. It sends an immediate request for the server to remove

[2]http://www.microsoft.com/net/

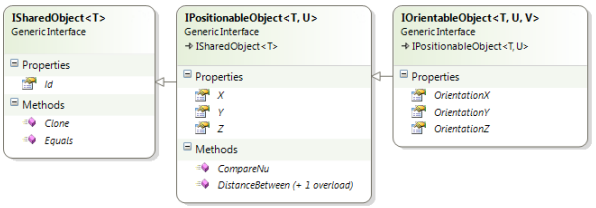

Figure 9: Interface hierarchy for shared objects.

the object from the main object pool.

DisconnectFromServer Disconnects the VFC4FPS client from the VFC4FPS server.

GetObjectRef Returns a reference to an object that matches the identifier provided. Changes to the object are made directly to the secondary object pool.

GetPhi Returns the client's view for manipulation.

GetUpdatedIDs Returns the identifiers for the objects that were updated from the server since the last call of this function. Avoids unnecessary processing by the client.

SetClientID Optional function that can change the client's identifier.

SetPhi Adds or updates a client's view and immediately sends it to the VFC4FPS server.

IsConnected Returns the state of the connection between the VFC4FPS client and server.

4.4 VFC4FPS Server API

Init Initializes the server in a specific port.

ClientIDsAffectedByPositionableEvent It is through this function that the game server carries out the filtering of events. It has two implementations, one that takes a position vector of a punctual event and a maximum radius in which this event is valid. Another that takes two position vectors, useful to represent line segments, and the highest index of the consistency zone in which this event is valid. It then returns the clients to which the event is relevant.

GetObject Returns a copy of the desired object. Any change to it's status is not propagated to the main object pool.

4.5 Data Structures

4.5.1 ObjectPool

This class corresponds to the implementation of the main and secondary object pool. It consists of two dictionaries. One dictionary stores instances of ISharedObject. The other stores the client who owns each ISharedObject. Both dictionaries are indexed by the identifier of the objects.

4.5.2 ClientStateData

This class is responsible for storing the current state of the consistency of each ISharedObject of each client. It stores this information using dictionaries. The information contained in these dictionaries is the last temporal instance that an object was sent to a client, the number of updates received since it's last send and a copy of the instance of the last sent object (for later comparison to the value parameter of the vector κ.

4.5.3 CubeOriObj

This class corresponds to the implementation of the IOrientableObject interface of the object that represents the avatar's state. In addition to the fields required by the interface, it contains the yaw and pitch of the avatar, falling speed in the three axes (fallingx, fallingy, fallingz), movement velocity in the three axes (velx, vely, velz), physical state (physstate), flags and selected gun (gunselect).

4.5.4 KVec

This class corresponds to the implementation of the vectors κ. It consists of two integers to represent time (theta) and sequence (sigma), and a float to represent the value (nu). To represent an infinite value to any of the parameters, the value (-1) is used.

4.5.5 Phi

This class corresponds to the implementation of views. It consists of two arrays of integers, one for specifying the dimensions of the zones and one for the viewing angles. It also contains a bi-dimensional array of KVec instances. This array is indexed by zone and field of view. This class also contains a list of integers to represent the pivots of the view. Each integer corresponds to an object identifier.

4.6 Communication

The communication between VFC4FPS client and server (figure 10) is done through .Net remoting. Events are managed by the original Cube 2 system. Object updates are managed by the VFC4FPS. Object are serialized in binary form and sent using the UDP protocol. This is because the .Net remoting protocol generates too much network traffic. However, a serialized object is considerably bigger than the game's original objects. For that reason it was used two methods of compression to reduce the object size.

4.7 Serialization and Compression

A serialized object contains meta-data which takes up much space. However, only a subset of bytes, those of the fields, are the only ones that change. In order do eliminate the redundant information we used a delta compression system.

After that we still had objects bigger than the original game. We then implemented a bitmap based compression to further reduce the object's size.

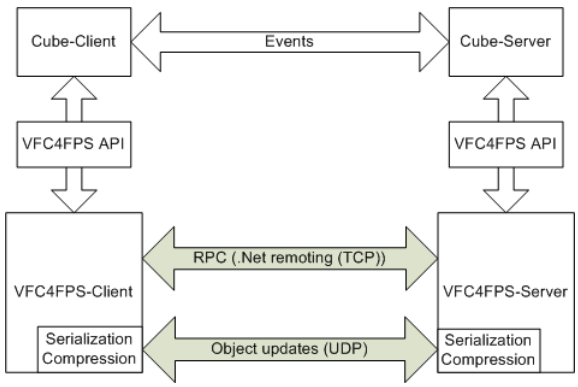

Figure 10: Communication architecture.

4.7.1 Delta Compression

In the figure 11 we can see an instance of the CubeOriObj class in it's serialized form. Highlighted in grey is the region of bytes that corresponds to the size of the fields of the class. Using introspection we can determine de total size of the fields. Using this size we can calculate the offset where these bytes start. These bytes constitutes the delta of the class. This method was devised from empiric observation, and can have unexpected results when applied to other circumstances. One limitation is the use of variable fields, like strings and arrays. But in the context of the VFC4FPS and Cube 2 it works.

4.7.2 Bitmap Compression

The bitmap compression has the goal of eliminating useless bytes, in this case, null bytes. This method of compressing an array of bytes is to use a bitmap to mark if the position in the array has a null value or not. Since each bit corresponds to a position in the array, a byte can index eight positions in the array.

In short (figure 12) this algorithm creates a bitmap of the array, and marks each bit with zero or one depending on the byte value. This produces a bitmap and an array without null bytes. these two are concatenated and a byte is added with the total size of the bitmap, necessary for decompression.

4.8 Bots

To make assessments of network traffic it would take a great many number of players and infrastructure. Instead, our assessments were made through the use of avatars controlled by the artificial intelligence (bots) of Cube 2: Sauerbraten.

However, in order to run multiple bots per computer it was needed to disable the graphical processing. Due to dependences with the game logic, was not possible to completely disable de graphical output. Instead we disabled textures, sounds, light maps and some models. This way we obtained a lightweight bot controlled client.

5. EVALUATION

5.1 Views

The performance of VFC4FPS is mostly associated to the views specified. Measurements were made using three different views: normal view, zoom view and view with only one FoV (VFC view). The normal view (table 2) contains the appropriate parameters to the normal view of the avatar.

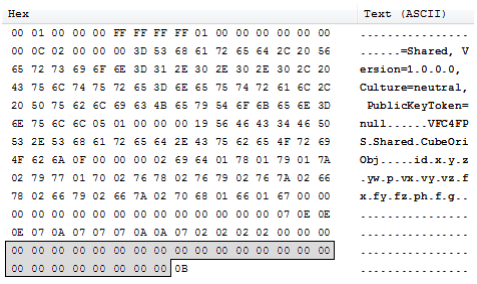

Figure 11: CubeOriObj serialized.

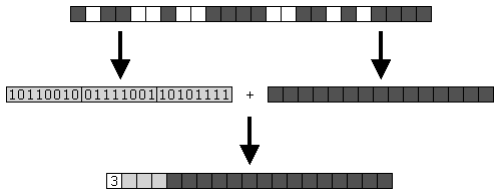

Figure 12: Bitmap compression.

FoV \ Zone	300	600	∞
100°	[30, ∞, 0]	[60, ∞, 3%]	[90, ∞, 5%]
140°	[60, ∞, 6%]	[90, ∞, 9%]	[130, ∞, 13%]
360°	[90, ∞, 9%]	[200, ∞, 20%]	[300, ∞, 30%]

Table 2: Normal view.

In the case of Cube 2: Sauerbraten, an avatar has a 100° field of view. Besides the 100°, in this view we see a field of view of 140°, this is to avoid momentary inconsistencies when doing a quick spin. These 140° represent 20° either side of the normal field of view.

The zoom view (table 3) contains the parameters used in the case where the avatar has the zoom activated. When an avatar has zoom, the field of view is reduced from 100° to 35°. The zones's radius were doubled in relation to the normal view. Again, the field of view has 20° to each side, which results in the 75°. The *kappa* vectors remain the same as the normal view.

The VFC view (table 4) is equal to the normal view, but contains only one field of view that covers all 360° around the avatar. This view was used to provide a comparison between the VFC4FPS and the VFC, and prove that the inclusion of FoVs allows greater performance.

5.2 Quantitative Evaluation

5.2.1 Server Network Traffic Reduction

To measure the reduction of traffic generated we resorted to the use of 48 bots on different maps. The choice of the number of bots is linked to the normal number of players normally seen in online servers. This number is at around 24 players. Since the aim is to reduce traffic by half, it was decided to double the number of "players" (bots) to determine if the gains were close to the traffic generated by 24 players.

All measurements were made in game mode ffa (free for all, with various types of weapons) in order to have more events of shooting and better demonstrate the impact of the event filter. Measurements were recorded only from the

FoV \ Zone	600	1200	∞
35°	[30, ∞, 0]	[60, ∞, 3%]	[90, ∞, 5%]
75°	[60, ∞, 6%]	[90, ∞, 9%]	[130, ∞, 13%]
360°	[90, ∞, 9%]	[200, ∞, 20%]	[300, ∞, 30%]

Table 3: Zoom view.

Zone / FoV	300	600	∞
360°	[30, ∞, 0]	[60, ∞, 3%]	[90, ∞, 5%]

Table 4: VFC view.

time when all 48 bots were connected to the server. The duration of each measurement was 10 minutes. This value was chosen to obtain stable average values and for being the normal time length of a map.

A server has inbound and outbound traffic. The inbound traffic is low and constant. The highest traffic is outbound. On those grounds we only present measurements of the server's outgoing traffic.

Figure 13 presents the results using the normal view. These graphs show the rate of outgoing traffic in bytes per second for the two communications channels for the two solutions (Original and VFC4FPS). As can be seen, the traffic rates are stable over time. We can observe a reduction of more than half the network traffic for object updates. Something else we can observe is the low traffic generated by events. The small difference between the original events and the "VFC4FPS" events is due to the filtering applied.

Figure 14 presents similar results to the previous paragraph, but this time with the zoom view. The use of this view lies in the fact that the zoom in this game is a visual feature that bots do not use. Therefore, in order to confirm that the use of zoom does not affect the reduction of traffic, it was considered the worst case, where all the bots have the zoom activated. As expected, increasing the radius of the zones along with the reduction of field of view did not impair the performance of VFC4FPS compared to the normal view. The main difference with the results of the normal view is the smaller difference between the traffic events. This is due to the fact that the zones are wider, which reduces filtering.

We end with the same measurements, but this time with the VFC view (figure 15). We can clearly see that the use of the field of view in VFC4FPS was justified.

Table 5 presents the average values for the ratio between the original game network traffic and the VFC4FPS traffic for different views and maps. There is a trend for the ratio to increase with the size of the maps. This is justified by the possibility of avatars to be more dispersed than in small maps. Numerically, we can conclude that the reduction achieved through the normal view and the zoom view is equal. We can also conclude that without the inclusion of

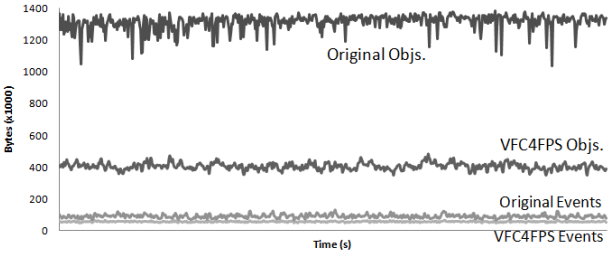

Figure 13: Server outbound traffic during 10 minutes in the flagstone map using the normal view.

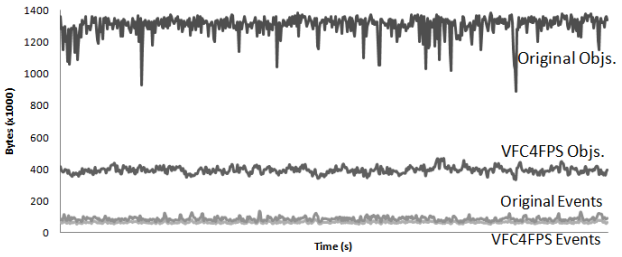

Figure 14: Server outbound traffic during 10 minutes in the flagstone map using the zoom view.

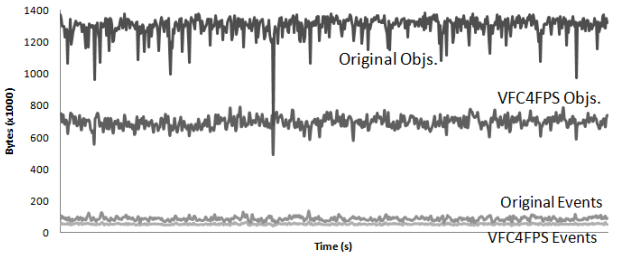

Figure 15: Server outbound traffic during 10 minutes in the flagstone map using the VFC view.

Map		Views		
Name	**Dimension**	**Normal**	**Zoom**	**VFC**
aard3c	250x250	1,7		
academy	250x250	1,5		
aqueducts	875x750	1,7		
arabic	875x750	2		
akroseum	1000x1000	2,2		
dust2	1000x1000	2,5		
campo	1000x1000	1,7		
venice	1000x1000	2,5		
wdcd	1000x1000	2,6	2,6	1,6
redemption	1250x500	2,3		
core_transfer	1250x750	2,7		
face-capture	1500x250	2,5		
damnation	1500x500	2,1		
shipwreck	1500x1250	3		
flagstone	1500x1500	3	3	1,8
hallo	1500x1500	2,7		
ph-capture	1500x1500	2,7		
river_c	1500x1500	2,9		
mach2	1750x750	2,9		
urban_c	2250x1750	2,8	2,7	1,7

Table 5: Average ratios after 10 minutes.

field of view, the reduction is lower.

5.2.2 Client Network Traffic Reduction

Unlike the server, in a client, the inbound traffic is the greatest. Therefore, this section presents the reduction achieved in the client due to the reduction done on the server. The simulation conditions were the same as for the server: 48 bots, 10 minutes, ffa mode and using the normal view.

Using figure 16 and table 6 as basis, we can observe the reduction achieved in a client. On average, regardless of the map, a client has a traffic reduction of more than two times compared to the original traffic. There is some variation in VFC4FPS traffic compared to the original traffic. This is because the reception of updates from the original server had a constant frequency, while in VFC4FPS, avatar movement and position strongly influences the frequency with which a client receives updates for the different objects.

5.3 Qualitative Evaluation

In order to evaluate if our use of VFC4FPS didn't harm the game playability, we conducted a test with real players. Each player would play two versions of the game. One was the original game, and the other the game using VFC4FPS. This test was done in a blind fashion, where players didn't knew which version was which.

Tests were made one player of a time, against 48 bots. Tests were conducted in the urban_c map in the instagib mode (where one shot kills). Each player started by playing 5 minutes in one version to practice. After that he would play for 10 minutes in the same version, after which it was asked for him to switch versions and play for another 10 minutes. The starting version (the practice one) was alternated between players. When the player ended playing both versions, it was asked for him to answer a short questionnaire. The main question was if he noticed differences between the two versions, mainly related to the opponent's movement.

5.3.1 Results

Tests were conducted with 16 volunteers. Their average age was 25 years. Only 3 of the volunteers answered that they encountered differences between the two versions. One of which favoured the VFC4FPS version. This means 88% favourable answers towards VFC4FPS.

We can see how the players experience with FPS games related to their response in the figure 17. Most players had some experience and didn't saw any differences.

Again, most players used the zoom function, and, despite the higher vision range, that didn't affect their perception between the two versions (figure 18).

	wdcd	flagstone	urban_c
Average ratio	2,5	2,9	2,9

Table 6: Average ratios for one client inbound traffic.

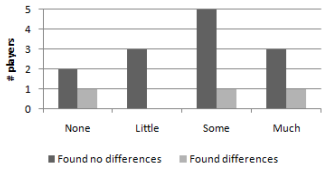

Figure 17: FPS experience distribution.

6. CONCLUSIONS

First Person Shooters have a fast paced action where the precision of the opponents movements is very important. When played online, communication is required to be very frequent to keep the players updated. This translates into a large network traffic, especially outbound traffic in the server in a Client-Server architecture. The high use of bandwidth is the main limiter of the number of players supported.

In this paper we presented several ways to reduce the use of bandwidth in multi-player online games. We covered the topics of architectures, consistency management, interest management, Dead Reckoning and techniques at the network level. And also some academic work.

We suggested VFC4FPS, a library to manage the consistency of FPS. The VFC4FPS is based on VFC, which allows a progressive and controlled reduction of consistency. Our system exploits the context of the limited view of the avatars in a FPS and adds to the VFC the concept of fields of view.

VFC4FPS was applied to the open source game Cube 2: Sauerbraten with the aim of reducing, at least by half, the use of bandwidth on the server. And thus offer support a greater number of players. We evaluated the performance of Cube 2: Sauerbraten with VFC4FPS in several maps, and the results show that the objective of reducing by half the traffic has been reached and in many cases exceeded. This, without harming the gameplay compared to the original Cube 2: Sauerbraten.

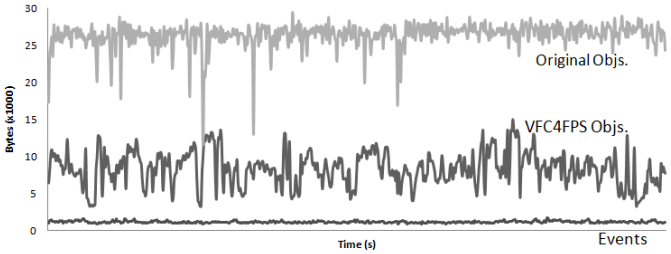

Figure 16: Client inbound traffic during 10 minutes in the flagstone map using a normal view.

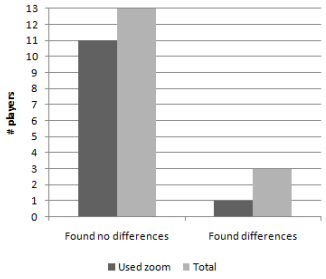

Figure 18: Zoom usage by players.

6.1 Future Work

The good results of this work open the door to some improvements:

- Currently, the implementation of VFC4FPS was very focused on the Cube 2: Sauerbraten characteristics, and some of the features were not complete. As future work, an implementation of VFC4FPS in the form of a completely generic library, without the limitations of the .Net platform would allow integration with a high number of FPS.
- We kept the Client-Server architecture, used by both Cube 2: Sauerbraten and VFC. Something that can be tried is the modification of the VFC4FPS to a P2P architecture.
- Investigate the possibility of including an efficient and generic geometric representation of the virtual world to enable greater gains by using the visibility of each client.

7. REFERENCES

[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] C. E. Bezerra, F. R. Cecin, and C. F. R. Geyer. A3: A novel interest management algorithm for distributed simulations of mmogs. In *DS-RT '08: Proceedings of the 2008 12th IEEE/ACM International Symposium on Distributed Simulation and Real-Time Applications*, pages 35–42, Washington, DC, USA, 2008. IEEE Computer Society.

[3] 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.

[4] E. Cronin, B. Filstrup, A. R. Kurc, and S. Jamin. An efficient synchronization mechanism for mirrored game architectures. In *NetGames '02: Proceedings of the 1st workshop on Network and system support for games*, pages 67–73, New York, NY, USA, 2002. ACM.

[5] J. Dyck. A survey of application-layer networking techniques for real-time distributed groupware. Technical report.

[6] J. Dyck, C. Gutwin, T. C. N. Graham, and D. Pinelle. Beyond the lan: techniques from network games for improving groupware performance. In *GROUP '07: Proceedings of the 2007 international ACM conference on Supporting group work*, pages 291–300, New York, NY, USA, 2007. ACM.

[7] S. Fiedler, M. Wallner, and M. Weber. A communication architecture for massive multiplayer games. In *NetGames '02: Proceedings of the 1st workshop on Network and system support for games*, pages 14–22, New York, NY, USA, 2002. ACM.

[8] 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.

[9] R. Krishna Balan, M. Ebling, P. Castro, and A. Misra. Matrix: adaptive middleware for distributed multiplayer games. In *Middleware '05: Proceedings of the ACM/IFIP/USENIX 2005 International Conference on Middleware*, pages 390–400, New York, NY, USA, 2005. Springer-Verlag New York, Inc.

[10] J. Pang. Scaling peer-to-peer games in low-bandwidth environments. In *In Proc. 6th Intl. Workshop on Peer-to-Peer Systems IPTPS*, 2007.

[11] L. Pantel and L. C. Wolf. On the impact of delay on real-time multiplayer games. In *NOSSDAV '02: Proceedings of the 12th international workshop on Network and operating systems support for digital audio and video*, pages 23–29, New York, NY, USA, 2002. ACM.

[12] L. Pantel and L. C. Wolf. On the suitability of dead reckoning schemes for games. In *NetGames '02: Proceedings of the 1st workshop on Network and system support for games*, pages 79–84, New York, NY, USA, 2002. ACM.

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

[14] N. Santos, L. Veiga, and P. Ferreira. Vector-field consistency for ad-hoc gaming. In *Middleware '07: Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware*, pages 80–100, New York, NY, USA, 2007. Springer-Verlag New York, Inc.

[15] J. Smed, T. Kaukoranta, and H. Hakonen. A review on networking and multiplayer computer games. In *Multiplayer Computer Games, Proc. Int. Conf. on Application and Development of Computer Games in the 21st century*, pages 1–5, 2002.

[16] S. Xiang-bin, W. Yue, L. Qiang, D. Ling, and L. Fang. An interest management mechanism based on n-tree. In *Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, 2008. SNPD '08. Ninth ACIS International Conference on*, pages 917–922, Aug. 2008.

[17] H. Yu and A. Vahdat. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Trans. Comput. Syst.*, 20(3):239–282, 2002.