Proj48534 - A Peer-to-Peer Support for Multiplayer Games

Abstract

The successful commercial MMOGs, at the moment, are implemented using some variant of the client-server model. This model offers simple design, good security and fast detection and treatment of cheating. However, it lacks scalability. The cost of maintaining the server becomes excessive with the increased number of customers. One approach to deal with this situation is distributing the MMOG simulation. The changes in the virtual world of the game would be processed by the machines of the clients, without interference from the server. This distribution can be achieved using the peer-to-peer model. In this paper, we describe a network engine, called Proj48534, which supports partially decentralized MMOGs, ensuring scalability and flexibility, while guaranteeing the basic properties of a client-server model, such as security and consistency. We first examine in more details the existent distributed models. After that, we discuss about the Proj48534 project and its characteristics. Finally we present a simulation of the architecture in comparison with the traditional client-server model.

Keywords: Peer-to-peer, Descentralization, Scalability, Multiplayer Games, Hybrid Topologies, Instanced Game, Distributed Systems

Author's Contact:

1 Introduction

Multiplayer games are a very popular game genre, due to their highly interactive nature. Among those games, a class that is growing in popularity is MMOGs (Massively Multiplayer Online Games). MMOGs are real-time multiplayer games played through the Internet, where a great number of players (usually thousands) play within a persistent-state virtual world. Successful examples include EverQuest [Sony Entertainment 1999] and World of Warcraft [Blizzard Entertainment 2004].

The most common network model used in MMOGs is client-server. In this model, the client only sends its data to the server-side (that can operate on a single machine, a cluster or a grid) and receives frequent updates from the server. The server process all the data from the clients, and broadcasts all the results that occur on the virtual world back to them. Some advantages of this model are: being simple in design; cheating can be efficiently detect and stopped; retaining control over access to the game; and being a predictable model.

The main disadvantage of the client-server model is the lack of scalability. The cost of maintaining the server-side becomes excessive with the increase in the number of clients. When talking about commercial games, the usual approach is to continuously expand the number of machines on the server-side. That is a reasonable solution, however it is not suitable for projects on a budget such as those from small companies or research groups.

One way of dealing with the above problem is to fully or partially distribute the MMOG simulation. All the changes on the game world would be registered and dealt within the client-side, with no interference of the server-side. Security and world state consistence are issues on that model, because there isn't a point where the changes in the virtual world can be evaluated and considered legal or illegal. The main challenge of our project is thus to provide a partially decentralized support model for MMOGs, while retaining the basic properties of the client-server model such as security, consistency and scalability [Schiele et al. 2007].

In this paper a hybrid model is proposed, combining the security and consistency of the client-server model with the scalability and flexibility of the distributed model. In sections 2 and 3 we discuss about Client-server and Peer-to-Peer models. The Proj48534 architecture, is explained in section 4. Section 5 presents the simulations we made to compare client-server and peer-to-peer models. Section 6 presents conclusions on the topic.

2 Client-server model

The client-server paradigm is undoubtedly the most currently used in the implementation of Internet real-time multiplayer games. Commercial examples of these games include Doom, Quake [id Software 1996] and Counter-Strike [Valve 2000], all them 3D real time action games. In any client-server game, massively multiplayer or not, the players interact with the environment and with other players through a client program. Each client has a network connection only with the server. The server is responsible for receiving the information from each client and passing updates to the other clients. There are several different ways the communication protocol between the client and the server can be implemented: some of them provide more security, while others provide efficiency.

Regardless of the technique used, the server is responsible for maintaining an updated state of the game. Since the game is a real time simulation, it is necessary for the state of the game to be frequently recalculated. In general, the server is configured to update its state in a fixed and relatively small frequency, and to periodically send update messages to all clients. Due to that, the server must have great processing power (CPU and, indirectly, memory) and enough bandwidth (network).

A distributed game can use the centralized simulation to accommodate thousands of simultaneous clients. In practice, however, it can be difficult to have in a single machine all the processing power required to perform the simulation in real time. One solution to this problem, commonly used in commercial MMOGs such as EverQuest and PlanetSide [Sony Entertainment 2003], is to divide the virtual world and simulate each division in an individual machine as, for example, a particular computer in a cluster of computers. But even using the clusters to mitigate the problem of the server processing cost, the problem of network consumption still remains. The MMOG server, a single machine or a cluster, will need an Internet connection with low latency and very high bandwidth.

3 Peer-to-peer model

There is a growing interest in research and development of peer-to-peer architectures for MMOGs. The models proposed seek decentralization as a way of increasing scalability and reducing dependency on nodes in trusted areas. Other benefits include the elimination of central points of failure, as well as the increase of responsiveness.

The Communications Architecture for Massive Multiplayer Games, proposed by [Fiedler et al. 2002], was the first architecture to address the scalability problem by proposing a solution based on the publisher-subscriber paradigm. In this paradigm, the virtual world is divided into smaller pieces, usually called "cells", and each participant can choose to sign (or participate) in only a few cells. Thus, each subscriber of a cell only needs to exchange update messages with subscribers of the same cell. These architectures also deal with other fundamental problems, such as the problem of responsiveness to commands generated by each participant.

There are other proposals that do not address the cheating problem, such as Hydra [Chan et al. 2007], that focuses on guaranteeing the consistency in the messages committed when nodes fail; and Mediator [Fan et al. 2007], which address the scalability and performance problems by proposing a solution based on a super-peer

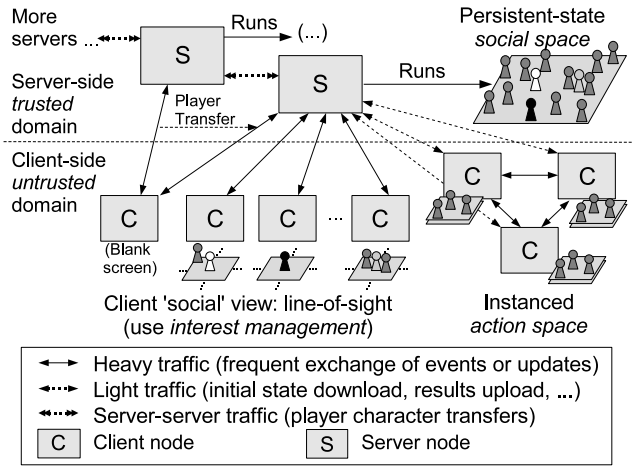

Figure 1: *The instanced 'social vs. action' MMOG*

network and a reward scheme: peers that contribute more can use more resources.

It can be noted that recent proposals for MMOG support based on peer-to-peer overlays are becoming more aware of issues in actually deploying a peer-to-peer MMOG on the Internet, as they show more concern about peer-to-peer problems, like hostile service providers and bandwidth limitations, for instance. There are many pure peer-to-peer and hybrid model proposals. However, some of these proposals don't offer any kind of deterrent mechanism against cheating players, which is an essential feature for implementing an online massively distributed and persistent-state game. In the next section we will show how our model offers scalability and also cheat-resistance so that its commercial application is viable.

4 Proj48534 project

The Proj48534 project is a distributed simulation model, and also a library (in reality, a stack of C/C++ libraries) that implements the instanced game model, which will be described in section 4.1. The instanced game model is the price to pay for a simple approach in unifying security, consistency and scalability in a decentralized MMOG. The objective is to drastically reduce the processing and communication cost in the server-side.

4.1 The Instanced Game Model

Several MMOGs, like PlanetSide and World of Warcraft, offer to the player the illusion of one or several large and contiguous virtual spaces where all the gaming takes place. Those spaces are divided in segments, and each server machine or group of server machines is responsible for one segment. A warping system is necessary to tie all segments together, allowing the player to move from one space to the other transparently.

The game Guild Wars introduced a new game model, which will be referenced to as the instanced game model [Cecin et al. 2006]. There are two virtual spaces on Guild Wars: the social space and the action space. The first kind is a medium-sized virtual space where a significant number of players can socialize, trade virtual goods and organize game sessions. Because of the nature of this space, low consistency requirements are necessary. Game sessions happen on the action space, which is a small-sized virtual space where a small number of players gather to play a game session. Higher consistency requirements are necessary on that space because of its fast and dynamic nature.

The relationship between the two spaces is as follows. When a game session ends, the players involved in it return to the social space. All the traits of the characters (e.g. virtual world money, experience points and statistics) are updated with new information from the session. The social space is designed as a contiguous world, as mentioned in the beginning of this section. The action

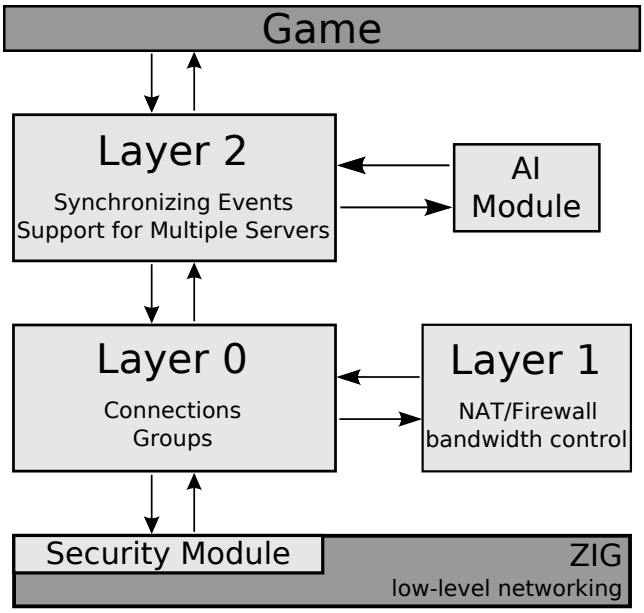

Figure 2: *Proj48534 Architecture*

spaces are created dynamically to support temporary game sessions with a small number of players. This space is destroyed after the session ends.

Guild Wars, as far as we can tell, follows the client-server model. Our proposal, which is illustrated in Figure 1, is to use the instanced game model with a hybrid network architecture model. Since the interactions on the social space (chatting, trading) don't require high consistency, and there is a necessity for validation of who can play (game accounts, passwords), the server-side will be responsible for coordinating this part of the game. So even a server-side with modest processing capacity and network bandwidth could manage the social space if game quality is scaled down accordingly, for instance by sending less frequent updates to clients. The game sessions would be processed only on the client-side machines, avoiding unnecessary processing and communication cost on the server-side. The clients would form groups, wich would operate within a peer-to-peer dynamics.

4.2 Proj48534 architecture

The functionality promised by our network engine is delivered by a set of libraries. Those libraries are organized in layers, as shown in Figure 2. The next sections describe the implementation of the three layers, followed by an overview of security and artificial intelligence (AI) techniques which are employed.

4.2.1 Layer 0

The first layer in the Proj48534 architecture is responsible for forming the mesh of point-to-point connections. The client's connections with the servers and with other clients are established and managed in this layer. The basic network functions for connection establishment, such as socket creation, connection and packet sending and receiving, are given by a low-level API provided by the ZIG library [Cecin 2007] which basically implements a protocol that resembles SCTP [Stewart et al. 2000] over UDP.

The Layer 0 server is responsible for managing connections with all clients. Naturally, it maintains a list of all clients that are connected to it along with a logical communication endpoint object for each client. When started, the server spawns a socialization space called the social space object. All clients that are not in an action space are bound to the social space. All communication between clients that are in the social space is intermediated by the server.

The server can create new groups called action spaces, and it can destroy such existing groups. In the action spaces, clients communicate directly, and this requires all clients to already have the

ability to send UDP packets to each other. The group object also allows clients of that group to broadcast messages to the whole group. Connection between group members is carried out through Peer objects. The clients, when in groups, maintain a list of Peer objects which is pretty much the same as the list of Client objects in the server, allowing peers to manage their connection with each other group member.

Layer 0 is also responsible for detecting connection timeouts. If a communication channel (client-server or peer-to-peer) has been without traffic for a specific amount of time, it is assumed to be dead and is terminated, generating a callback for the upper layer. To prevent alive links from dying, Layer 0 employs keep-alive messages.

4.2.2 Layer 1

Layer 1 of the Proj48534 architecture is responsible for handling connection-related problems between the peers within a group. This layer will ensure that all peers are able to send and receive messages from each other.

We see two main problems that can cause a connection failure between two peers: NAT/Firewall block and bandwidth insufficiency. When the Layer 0 indicates that a message must be send from one peer to another, Layer 1 must verify if there is a direct connection between the peers. If this connection doesn't exist, an alternative path must be discovered, using intermediate peers. After the path is discovered, the proper routing of the message can take place.

Among all peers within a group there are different bandwidth capabilities. It is very likely to occur a situation where a peer doesn't have enough upload bandwidth to deal with the number of messages that it has to send. Layer 1 will identify such situation and, instead of sending the messages directly to the destinations, send them to a peer with good upload bandwidth capability, holding him responsible to send the message.

4.2.3 Layer 2

The upper layer of the Proj48534 implementation is responsible for guaranteeing the simulation consistency on action spaces, providing support for multiple servers and exposing all of the implementation's functions to the game programmer. Due to the simulation of action spaces being executed asynchronously in each peer, there is no intrinsic guarantee of consistency. To partially address that, Layer 2 introduces the role of the super-peer, which is a special peer responsible for receiving, ordering and redistributing some of the game events such as updating player scores and picking up the flag in a capture-the-flag action game, for instance. To avoid overloading the super-peer, position updates or player movement requests (depending on the game's protocol design) are to be sent directly from each peer to every other peer using the unicast full-mesh provided by the lower layers. Not only position updates, but any other frequent, delay-sensitive and weakly-coupled message type should also be sent peer-to-peer.

Depending on the game, the resulting conflicts from a lack of centralized timestamping of movement and similar events, whenever they ensue, can be either handled by each peer individually or left for the super-peer to detect it and issue a special correction message later, as the BZFlag protocol does [Pellegrino and Dovrolis 2003]. Local corrections are an option if each peer is authoritative over its own avatar's position and if it broadcasts position updates for its own avatar. For instance, if a peer's local avatar overlaps with a remote peer's, the local avatar can push its own avatar out of that position. The remote avatar will do the same. The only issue is ensuring that the correction step won't make both peers try to resolve the situation by unstucking their avatars to the same spot again. This local conflict resolution is what we are currently employing in our capture-the-flag action prototype, Hoverkill [Singular Studios 2006].

Besides ordering and distributing some of the events, the super-peer is responsible for the communication of the peer-to-peer action group with the server. The super-peer is the one that reports gameplay results back to the server, whenever necessary.

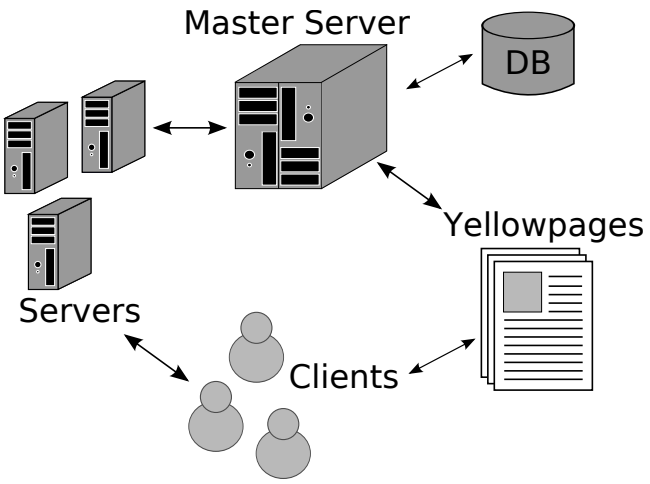

Figure 3: *Multiple Servers Support*

Layer 2 also introduces support for multiple game servers (Figure 3) in the API. The idea is that players can choose their server from a list of servers. Each server will typically host a single social space and when that is full (server capacity is reached) players can connect to other social spaces in other servers. To tie all of the players scattered on different servers together, there is a master server to which all servers are connected. Whenever a player connects to a server and authenticates, his player account is downloaded from the master server to that game server. When the client drops from the game server, his account is uploaded back to the master server. There is a simple Yellow Pages Server proxy that can be used by clients to query the master server for the currently available game servers.

The social space support provided by Layer 2 is a simple client-server API. Any interest management [Morse et al. 1996] in the social space is currently left for the application to implement. We are planning to add something akin to OpenTNL ghosting [Garage Games 2004] into the API which would allow the game programmers to optimize the social space more easily. Another idea is to implement a peer-to-peer mesh for social spaces, but that one would have to be scalable (no full-mesh possible). Since we are aiming at a simple solution, the use of probabilistic broadcast for position updates on the social space is being considered, leaving timestamping and distribution of more infrequent events for the server.

4.2.4 Artificial Intelligence Module

In the previous section, we revealed that we encourage peers in an action space to send authoritative position updates, among other events that might have low-latency requirements and a low probability of having their order of execution affecting the resulting game states too much (weakly-consistent events). This opens up the possibility of cheating.

In this project we tried a somewhat unique approach. Since most events would be still centralized at the super-peer and most action games have one-per-client avatar position updates as their consistency, bandwidth and latency bottlenecks, we just left peers with total control over their positions and with the responsibility for sending them directly to other peers in a broadcast fashion. This optimizes for consistency and latency, and doesn't create a bandwidth bottleneck at the super-peer. On top of that, we tried AI-based detection of unusual patters over streams of position updates coming from each peer. The main motivation was to detect blatant speed-cheating being performed by players while other players (without admin powers, nevertheless) quit the game in frustration. It was not our goal to prevent subtle position manipulation, since we wanted to avoid false positives more than we wanted to prevent any form of position cheating.

We have trained, using back-propagation, an Artificial Neural Network (ANN) over traces of gameplay generated from Hoverkill play sessions, which is our actual game prototype. We obtained some

positive initial results [Gaspareto et al. 2008], but we also found that a successful ANN detection engine is tied to many dynamic parameters, including specific game rules and maps, general behavior of game players (if it changes over time, the network must be re-trained), quality of the traces (they must cover all the detectable patterns and with enough variation), and others. The results put some merit on the general idea that using an automatic detection engine, instead of human attention, to discourage illegal position manipulation during gameplay could enable more simple and efficient but vulnerable protocols such as ours to still succeed in a real MMOG deployment scenario.

4.2.5 Security Module

The main purpose of the security module is to provide secure communications between server and clients, and between peer clients. Our security module is almost completely transparent to the application, both in terms of API and in network and CPU overhead. The module does many tasks behind the scenes, such as crypto key distribution and secure channel handshakes, tending to each application's individual needs for message confidentiality, integrity and authenticity.

The implementation works mainly by intercepting any engine UDP packets before they are written to the UDP socket for sending. In each packet, a small header is inserted with control information and, upon first contact between two parties (being them clients, servers or both), that message is delayed while another quick message exchange is performed for setting up the secure channel on-the-fly. The root Certificate Authority (CA) of the crypto system is the operator of the master server.

5 Simulations

Since one of the main objectives of the Proj48534 engine is to reduce the bandwidth utilization by the server, a numerical comparison between the load on a traditional server and the load on a P2PSE server was required. In order to do that, it was necessary to simulate two game servers, each one with players connected to it. One of them was a traditional game server, to which the clients kept connected and trading packets the whole time. The other one was a Proj48534 server, where the players needed to trade packets with the server only while they were in the social space. In the action space, the players formed a peer-to-peer group, communicating to one another directly.

The ns-2 simulator [McCanne et al. 1995] was used to create the simulated environment and perform the simulation itself. The servers and clients - both traditional and Proj48534 - were programmed extending the Application base class of the ns-2 API. The Proj48534 clients connect to the Proj48534 server, sending packets to it at a fixed rate while in the social space. The server, in turn, sends to each client in the social space an update containing information about all other clients in the same space. When a player leaves the social space and joins a peer-to-peer group of an action space, it stops exchanging packets with the server, due to its direct communication with the other players. The non-Proj48534 server and clients behave simply as follows: each client sends a packet to the server at a given fixed rate, while the server broadcasts an update to all clients, also at a fixed rate. Although this approach does not take into consideration possible optimizations on the server, such as interest management [Morse et al. 1996], this is not relevant, since the same optimizations could be used in the social space of the Proj48534 server, resulting in a similar bandwidth usage reduction percentage, when comparing the two server models.

We based our simulation on some works, such as [Breu 2007], [Park et al. 2005] and [Feng et al. 2005], which analyse the network traffic generated by action games, since this is the genre targeted by the Proj48534 engine. As in most first-person shooters, the transport layer protocol used was UDP. Each packet received by the server from the clients is 100 bytes long. The update sent to each client, containing information about all the players, is proportional to the number of players present in the game, for the traditional server. For the Proj48534 server, the update length is based only on the

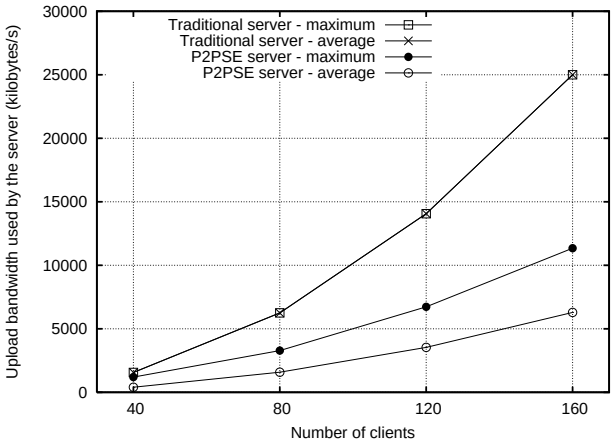

Figure 4: *Upload bandwidth utilization by the server*

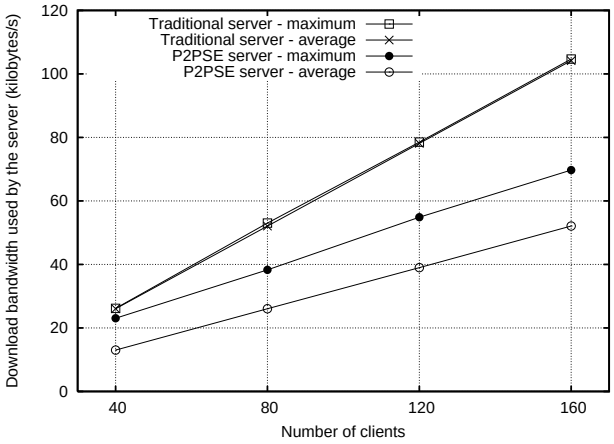

Figure 5: *Download bandwidth utilization by the server*

number of players in the social space. In both cases, the length of this update is 100 bytes multiplied by the number of players who will be updated. The interval between two consecutive packets received from each client is 150 ms, while the interval between two consecutive updates from the server to each client is 100 ms. Each player keeps alternating between the social space and the action space. The time he stays in each one of them is chosen randomly, ranging from 0 to 20 minutes, for each space change. The whole game session lasts for 1 hour.

The results were collected as follows: to measure the average upload bandwidth usage by the server, the sizes of all packets sent during the whole session were added up and then divided by the session time; to determine the maximum usage, it was measured how many bytes had been sent each second and the highest value was selected. In Figure 4, it is shown the maximum and average upload bandwidth utilization results found for a traditional server and a Proj48534 server. In Figure 5, the download bandwidth usage is also depicted. Table 1 and Table 2 show the numerical values of the average bandwidth utilization found in the simulation.

As we can see, the average download bandwidth utilization is decreased by, approximately, one half, and the average upload bandwidth utilization is reduced to one quarter. This happens because the periods during which a client stays on the social and action spaces have approximately the same duration. In consequence of that, there are roughly half the clients exchanging packets with the server for each instant, in the average. As each client sends fixed size packets periodically to the server, the bandwidth occupation grows linearly, and so, as the number of players halves, so does the server's download bandwidth utilization. However, as the update sent by the server to each client is proportional to the number of clients being updated, the upload bandwidth utilization by the server grows quadratically. Therefore, when the number of players

is divided by two, the upload bandwidth utilization of the server is divided by the square of two, that is, four.

It is important to notice, however, that the time a player usually spends playing action games, such as first-person shooters, is much longer than the time they keep talking in the game chat room. Therefore, the average time spent by each player in the social space should be much shorter than the time spent on the action spaces in the simulation. Also, the social space will most likely have much less need for frequent updates, allowing further decrease of the bandwidth utilization by the Proj48534 server. Anyway, those parameters were chosen considering a pessimistic scenario, where the social space is as interactive as the action space, and the players spend more time socializing than it usually occurs in this kind of games.

6 Conclusion

We proposed a different approach to MMOGs, that uses P2P groups and transfers the simulation processing to them. In order to provide security and scalability, the game model was restrict to the instanced game model. The existence of a server-side guarantees that, if deem necessary, the server can act as final arbiter. The performed simulations demonstrate that, even in a pessimistic scenario, the reduction of the average bandwidth usage by the game server can be significant. The download bandwidth utilization was reduced by one half, while the upload bandwidth utilization was the most benefited one, decreasing by 75%, allowing to reduce the maintenance cost of such kind of server.

We are currently integrating the Proj48534 architecture with our game prototype, Hoverkill. This game is a client-server capture-the-flag tank game, and its network system is being replaced by our architecture. The final goal is to be able to test the game in client-server and peer-to-peer modes, in order to base our conclusions in real results rather than simulations..

Acknowledgements

This work was supported by the Funding for Studies and Projects (FINEP), through the Proj48534 project (Proj48534-5849-1), and by the National Research Council (CNPq).

References

BLIZZARD ENTERTAINMENT, 2004. World of warcraft. http://www.worldofwarcraft.com/.

BREU, L., 2007. Online-Games: Traffic Analysis of Popular Game Servers (Counter Strike: Source).

CECIN, F. R., GEYER, C. F. R., RABELLO, S., AND BARBOSA, J. L. V. 2006. A peer-to-peer simulation technique for instanced massively multiplayer games. *Proceedings of the Tenth IEEE International Symposium on Distributed Simulation and Real-Time Applications (DS-RT'06)-Volume 00*, 43–50.

CECIN, F. R., 2007. Zig game engine. http://zige.sourceforge.net/.

CHAN, L., YONG, J., BAI, J., LEONG, B., AND TAN, R. 2007. Hydra: a massively-multiplayer peer-to-peer architecture for the game developer. *Proceedings of the 6th ACM SIGCOMM workshop on Network and system support for games*, 37–42.

FAN, L., TAYLOR, H., AND TRINDER, P. 2007. Mediator: a design framework for P2P MMOGs. *Proceedings of the 6th ACM SIGCOMM workshop on Network and system support for games*, 43–48.

FENG, W., CHANG, F., FENG, W., AND WALPOLE, J. 2005. A traffic characterization of popular on-line games. *Networking, IEEE/ACM Transactions on 13*, 3, 488–500.

FIEDLER, S., WALLNER, M., AND WEBER, M. 2002. A communication architecture for massive multiplayer games. *Proceedings of the 1st workshop on Network and system support for games*, 14–22.

GARAGE GAMES, 2004. Open tnl - torque network library. http://www.opentnl.org/.

GASPARETO, O., BARONE, D., AND SCHNEIDER, A. 2008. Neural Networks Applied to Speed Cheating Detection in Online Computer Games. *To be published in the 4 th International Conference on Natural Computation (ICNC'08), Shandong University, Jinan, China*.

ID SOFTWARE, 1996. Quake. http://www.idsoftware.com/games/quake/.

MCCANNE, S., FLOYD, S., ET AL. 1995. Network simulator ns-2. *The Vint project, available for download at http://www.isi.edu/nsnam/ns*.

MORSE, K., ET AL. 1996. Interest management in large-scale distributed simulations. *Technical Report ICS-TR-96-27, University of California, Irvine*.

PARK, H., KIM, T., AND KIM, S. 2005. Network Traffic Analysis and Modeling for Games. *LECTURE NOTES IN COMPUTER SCIENCE 3828*, 1056.

PELLEGRINO, J., AND DOVROLIS, C. 2003. Bandwidth requirement and state consistency in three multiplayer game architectures. *Proceedings of the 2nd workshop on Network and system support for games*, 52–59.

SCHIELE, G., SUSELBECK, R., WACKER, A., HAHNER, J., BECKER, C., AND WEIS, T. 2007. Requirements of Peer-to-Peer-based Massively Multiplayer Online Gaming. *Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, 773–782.

SINGULAR STUDIOS, 2006. Hoverkill. http://www.singularstudios.com/sitehoverkill/.

SONY ENTERTAINMENT, 1999. Everquest. http://www.everquest.com/.

SONY ENTERTAINMENT, 2003. Planetside. http://www.everquest.com/.

STEWART, R., XIE, Q., MORNEAULT, K., SHARP, C., SCHWARZBAUER, H., TAYLOR, T., RYTINA, I., KALLA, M., ZHANG, L., AND PAXSON, V. 2000. Stream Control Transmission Protocol.

VALVE, 2000. Counter-strike. http://www.counter-strike.net/.