Università della Svizzera italiana

Faculty of Informatics

Research Prospectus

July 7, 2011

A fault tolerant multi-server system for MMOGs

Carlos Eduardo B. Bezerra

Abstract

MMOGs – massively multiplayer online games – have become, in the last decade, an important genre of online entertainment, having a significant market share. In these games, many thousands of participants play simultaneously with one another in the same match. However, this kind of game is traditionally supported by a powerful (and expensive) central infrastructure with considerable cpu power and very fast and low latency Internet connections. This work proposes the use of a geographically distributed server system, formed by voluntary nodes which help serving the game to the players. Two very important features, therefore, must be provided: consistency (which may be week and only existing in the players' perspective) and fault tolerance, as some of the participating nodes may crash and become unavailable. For this work, we consider the crash-stop failure model.

Research Advisor
Prof. Fernando Pedone

Research Co-advisor
Prof. Cláudio Geyer

Academic Advisor
Prof. Fernando Pedone

Review Committee
Prof. Committee Member1, Prof. Committee Member2

Research Advisor's approval (Prof. Fernando Pedone): Date: ....

PhD Director's approval (Prof. Michele Lanza): Date: ....

1 Introduction

MMOGs emerged in the last decade as a new trend because of the decrease of the domestic Internet connection cost and the increase of its average bandwidth. This kind of games have become very popular because they allow the online interaction of a massive number of players at the same time. In the most succesful cases, such as World of Warcraft [5], Lineage II [19] and EVE Online [6], for example, tens of thousands of players are supported simultaneously [7]. In these games, the players can interact with one another in a virtual environment.

Generally, each player controls an entity called *avatar*, which is his representation in the virtual environment – a character who executes the orders given by the player, therefore interfering in the outcome of the game. To provide a consistent view among the different participants, every action performed by each avatar is processed by a server, which calculates the game state resulting from such action. This new state is then broadcast to the players to whom it might be relevant – again, the state itself does not need to be consistent, but only what the players see.

To cope with the receiving and processing of thousands of actions and the broadcast of their respective resulting states, the traditional approach is to use a large central server – usually a cluster with a high-speed, low latency Internet connection – which provides such cpu power and network capabilities [11]. Decentralized alternatives could be used, so that such expensive infrastructure would not be necessary. For example, a fully decentralized system could be used, where each player is a peer responsible for calculating a portion of the game state and for synchronizing it with other peers [21, 12, 10, 13, 14].

In such peer-to-peer approach, the players would need to agree upon the outcome of the actions, in order for their view to be consistent. To reduce the traffic between peers, the state updates could be sent only to whom they are relevant. To define what is relevant for each player, the virtual environment could be divided into regions [22], so that a small peer group would be formed by players who were inside each of these regions. Players whose avatars were in a given region could form a smaller peer-to-peer group – possibly reducing also the cost of each agreement – and their own computers would decide the outcome of the players' actions in that area of the virtual environment. Unfortunately, however, the fully decentralized approach would imply a heavy communication cost on the players, specially when running some agreement protocol. This might not only degrade the quality of the game, but also increase the requirements imposed on the players' computational resources, therefore likely reducing their number.

We propose here, then, a system composed of geographically distributed nodes which may act as game servers – these nodes could be provided, for example, by volunteers or by companies which might have some commercial interest with the game being hosted. Some works, such as [2, 20, 8, 17] also propose the use of a distributed server, but they consider that the nodes are connected through a high speed and low latency network (e.g. a cluster), what makes their solutions only partially applicable in a scenario with highly dynamic and volatile resources, such as wide-area networks formed with volunteer resources. Such networks have some inherent problems: nodes with low availability and dependability, low bandwidth and low processing power compared to current dedicated MMOG servers maintained by large game companies.

The rest of this document is organized as follows: Section 2 describes the basic features of an MMOG server; Section 3 introduces our proposed distributed server model and describes briefly some of its protocols; Section 4 presents the research directions to be followed next and Section 5 lists some of the accomplishments achieved by this research so far.

2 MMOG server overview

We consider here persistent state real-time games with virtual environments where each player controls an avatar – a virtual object that represents him – by issuing commands to it. Through the avatar, the player interacts with the virtual world and with objects present in it, such as avatars from other players. The basic operation of the game network support system must be as follows: when a player connects to the server, it must send the current state of the player's avatar and of the surroundings of its current location; after that, this player keeps receiving periodic state updates for the objects which are relevant to it (usually based on whether those objects may be seen by the player [3, 1, 18]). When a player issues a command to the avatar (e.g. move avatar from one place to another, pick up an object, attack another player's avatar etc.), it is sent as a message to the server, which processes it and decides its outcome, probably changing the state of the game, which is then broadcast to all involved players.

2.1 Services to be provided

In order to give to the players the sense of a persistent virtual world through which they can wander and interact with each other, the MMOG server must provide, at least, the following services:

- **Simulation** of the game, that is, the processing of commands sent by players to their avatars, calculating the corresponding outcomes, which may result in changes to the game state;
- **Broadcasting** a portion of the current game state to the different players connected to it, so that their view is consistent with the changes that might have happened;
- **Storage** of the game state, which is the combination of the states of all objects in the virtual environment.

It is important to note that in real-time games (as opposed to turn-based games), due to the possibly fast-paced interaction of the players, it is critical to provide **timeliness** for the delivery of state update messages to them, so that they have the illusion of a shared environment. For the same reason, it is necessary to provide **consistency** for the game state between each pair of players interacting with one another, which means that the game state they perceive must be as similar as possible. Although it is very hard to guarantee that every player will share the exact same view, it is important for their commands to be processed as if there was a single, strongly consistent game state, which is the one stored at the server.

3 Distributed server system

The main purpose of the system is to reduce or, ideally, eliminate the necessity of a central server for the game. The approach we propose to achieve this is based on principles similar to those of volunteer computing, where a potentially large set of contributors make their computers available to perform tasks requested by some remote entity. With such a system, although each of these nodes is probably much less powerful than a traditional MMOG server, when working together they can sum up the cpu power and the network bandwidth necessary to host one of these games.

However, several questions arise when dealing with a potentially large, geographically distributed server system composed of voluntary nodes. First, as the nodes are volunteer, there is no guarantee that they will be available whenever they are needed, neither there is any guarantee regarding the amount of resources that each of these nodes makes available for the system – be it cpu power, network bandwidth or storage capability. Group communication may also be a problem, since primitives for that – such as network level multicast – are not widely available on the Internet.

Finally, considering the potentially large number of nodes in this system, it is very likely that some of them present failures, for what we are assuming the crash-stop failure model – therefore, we assume the possibility of nodes crashing and network links becoming unavailable. The services provided by the system must continue even in the presence of such failures. Not only must the distribution itself be transparent to the players, but also the existence of failures and their countermeasures must not be noticeable by them.

In Section 3.1, we describe how the game state will be distributed among the different server nodes and, in Section 3.2, a preliminary consistency management protocol, based on state machine replication, is presented. We also introduce, in Section 3.2.1, our group communication protocol on top of which our consistency management is performed.

3.1 State partitioning

In order to split a game server into several smaller units executed by many volunteer nodes, it is used the idea of dividing the virtual environment in regions, each of which being managed by a different server. Different approaches can be found in the literature, such as using fixed size cells and grouping them to form the regions [9], or using binary space partition trees [4]. To tolerate failures, each server has several replicas, forming a server group.

The state partitioning is performed based on which objects are located in each region. The state of an object is managed, then, by the server group assigned to the region where that object is located. This way, the locality of data is explored: objects close to each other are more likely to interact and, as they are located on the same server group, their interaction should be faster. Besides, by exploring the locality of the game data (state of the objects), the inter-group communication overhead is reduced.

By having multiple server replicas in each region, the cost of broadcasting state update messages to the players is also reduced. Since all the replicas in a group are supposed to have an identical state, each one of them may send state updates to a different subset of the players located in that region. As the number of players may be as large as many thousands, this would probably be the most costly operation for the system and distributing it would have a significant impact on the system scalability.

3.2 Consistency protocol

As mentioned before, the players interact by issuing commands and having them processed by servers, which calculate the resulting game state and send it to the players. Every command can alter the state of the game, possibly based on a previous state. We assume, however, that the commands lead to a deterministic resulting game state. In other words, if the game starts in the state S, there is only one possible state S' to be reached after a sequence $C = \{c_1, c_2, ..., c_n\}$ of commands.

The game state could then be modeled as a state machine, which would be replicated across the different server nodes, as proposed in [15, 25, 16], and where each command would be responsible for a state transition. One way of implementing the distributed server system would be by creating such state machine and forwarding all commands to every server node, in the same (total) order. As the state transitions are deterministic, every server would be able to calculate the correct game state independently[1]. We would, then, have a fully replicated server system.

The problem with a fully replicated state machine approach for a distributed game server is that it would hardly scale well. As the number of servers increased, so would the cost of processing a command, as it would have to be totally ordered among other commands received by any of the other servers. Not only the number of control messages would grow and potentially saturate the servers' network bandwidth, but also the time needed for the total order of the commands to be achieved would possibly increase beyond a tolerable value.

We decided, therefore, to use a partial state machine replication scheme, so that only the server group responsible for a given state will have to process the commands that alter such state. To make this possible, each command must specify a *target set* – defined either by the player or by a server –, which contains the objects whose states are needed or altered by the command. As every object is assigned to a different server group, the protocol will then be able to infer which servers must process each command.

Instead of forwarding each command to every server, now only the servers that manage some object in the target set of a command would process it. The problem with such partial state machine replication is that, in order to execute some command, a group might need the previous state of an object managed by some other group. To solve this, a possibilty would be to have the groups requesting the states they need from other groups. Another solution, probably more latency-efficient, would be every group proactively sending the new states of the objects managed by it to other groups which might need them.

Sending new states to other groups proactively based on the possibility that they might need such states, however, may lead to all groups sending to each other the state changes of every object, degenerating to a fully replicated approach. Nevertheless, we can explore the fact that, in games, objects interact only with other objects situated nearby: we can define the regions of the virtual environment in such a way that only objects either in the same or in neighboring regions would be able to interact – two regions are neighbors if, and only if, they share a border. As commands are executed by avatars, which are simply objects controlled by players, the result will be that, for each pair of objects in the target set of any command, these two objects will be either located in the same or in neighbor regions. This way, only groups managing regions that share a border would need to forward commands to each other[2].

Figure 1 illustrates two avatars situated in regions managed by different server groups, but still close enough to interact. Each command issued by their players has to be forwarded to the groups that manage some object in its target set, which is done by multicast.

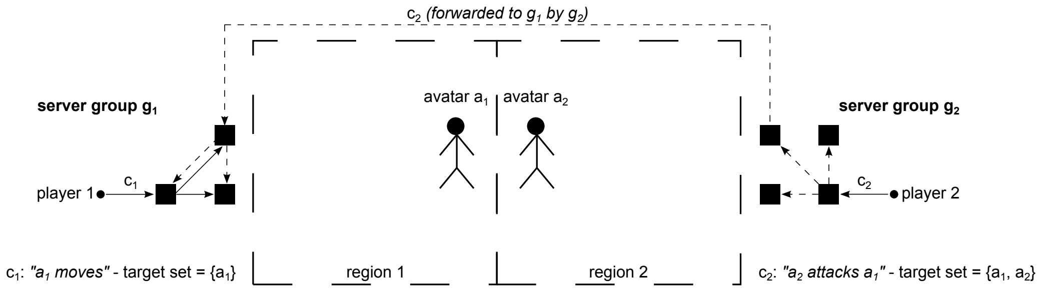
c2 (forwarded to g1 by g2)
server group g1
avatar a1
avatar a2
server group g2
player 1
c1
c2
player 2
c1: "a1 moves" - target set = {a1}
region 1
region 2
c2: "a2 attacks a1" - target set = {a1, a2}

Figure 1. Multicasting a command to a group which manages an object in that command's target set

[1]It might be the case that some servers could still be processing an older command, while others might have already processed newer inputs. Although this might lead to an old state, it would never lead to an incorrect state. To check which one is the newest, it would be enough to have a version number considering, for instance, the number of commands processed, since the commands arrive in the same order at the different servers.

[2]Although unlikely, it is possible to have a command sequence $C = \{c_1, c_2, \ldots, c_n\}$ where (1) every c_i depends on the state resulting from c_{i-1}, (2) not every c_i is processed by the same group and (3) not all of these groups manage neighbor regions. Nevertheless, the servers can detect such dependencies and forward the object states to the servers which need them. In the worst case, every group but one – there is no dependency cycle – would be waiting for another group, so the time needed to process a command would be proportional to the number of groups.

3.2.1 Quasi-genuine multicast

As the servers are organized in groups, and every command is sent to one or more of such groups, we can use a multicast protocol. Also, as commands seen by different servers must be handled in the same order, we need total order, so that the state machine replication approach can be used. Besides, the commands sent by each player should be executed in the same order in which this player issued them; therefore, the multicast protocol should provide FIFO ordering.

However, the inter-group communication overhead should be minimized, so that the distributed server would be more scalable. Non-genuine multicast primitives [24] require that control messages keep being exchanged between groups, even when there is no application level message to be sent. On the other hand, although genuine multicast algorithms minimize the number of messages exchanged, they impose a greater latecy on their delivery [23].

We devised, then, a multicast protocol that, although not genuine, allows the system to increase its scale without getting flooded with control messages. We call it *quasi-genuine*, because every group keeps sending messages to (and receiving from) other groups, even if there is no application level message to be exchanged, but only the ones with which it *is possible to* exchange messages. The information concerning which other groups a given group can communicate with is given by the application.

The advantage of a quasi-genuine multicast protocol is that it would be a middle ground between genuine and non-genuine protocols, inheriting some advantages from both – namely, improved scalability and lower message delivery time –, at the cost of restricting which groups can communicate with each other. Although the system can be reconfigured to change the group pairs that are able to exchange messages, such reconfiguration will have some cost, so this multicast protocol is more suitable for applications where this would not be frequent.

This can be used by our consistency management algorithm because every server group only has to forward commands to a specific set of other groups, which are the ones managing neighbor regions, and this is known as soon as the partitioning has been performed. Changes to the partioning should be performed rarely, as it would incur in some players migrating between servers. Therefore, the set of group pairs that can communicate with each other will not change frequently.

To further decrease the message delivery time, an optimistic delivery will also be employed. By assigning timestamps based on the local clocks – which are supposed to be more or less synchronized – and by every server waiting long enough for messages to arrive from another servers before performing the delivery, the final total message order will most likely match the timestamp order, so that the messages could be delivered in only one communication step, although that delivery would have to be confirmed by some conservative delivery.

4 Future work

We are currently workings towards the validation of both the optimistic multicast

5 Activities and publications

Some related work that we have published so far include:

References

[1] D. T. Ahmed and S. Shirmohammadi. A Dynamic Area of Interest Management and Collaboration Model for P2P MMOGs. *Proceedings of the 12th IEEE International Symposium on Distributed Simulation and Real Time Applications, 2008, Vancouver, BC, Canada*, pages 27–34, 2008.

[2] M. Assiotis and V. Tzanov. A distributed architecture for MMORPG. *Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*, 2006.

[3] C. E. B. Bezerra, F. R. Cecin, and C. F. R. Geyer. A3: a novel interest management algorithm for distributed simulations of MMOGs. *Proceedings of the 12th IEEE International Symposium on Distributed Simulation and Real Time Applications, 2008, Vancouver, BC, Canada*, pages 35–42, 2008.

[4] C. E. B. Bezerra, J. L. D. Comba, and C. F. R. Geyer. A fine granularity load balancing technique for MMOG servers using a kd-tree to partition the space. *VIII Brazilian Symposium on Computer Games and Digital Entertainment - Computing Track, 2009, Rio de Janeiro, RJ, Brazil*, 2009.

[5] Blizzard. World of Warcraft, 2004. http://www.worldofwarcraft.com/.

[6] CCP. EVE online, 2003. http://www.eve-online.com/.

[7] A. Chen and R. Muntz. Peer clustering: a hybrid approach to distributed virtual environments. *Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*, 2006.

[8] R. Chertov and S. Fahmy. Optimistic Load Balancing in a Distributed Virtual Environment. *Proceedings of the 16th ACM International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, 2006.

[9] B. De Vleeschauwer, B. Van Den Bossche, T. Verdickt, F. De Turck, B. Dhoedt, and P. Demeester. Dynamic microcell assignment for massively multiplayer online gaming. *Proceedings of 4th ACM SIGCOMM workshop on Network and system support for games*, pages 1–7, 2005.

[10] A. El Rhalibi and M. Merabti. Agents-based modeling for a peer-to-peer MMOG architecture. *Computers in Entertainment (CIE)*, 3(2):3–3, 2005.

[11] W. Feng. What's Next for Networked Games? *Sixth annual Workshop on Network and Systems Support for Games (NetGames)*, 2007.

[12] T. Hampel, T. Bopp, and R. Hinn. A peer-to-peer architecture for massive multiplayer online games. *Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*, 2006.

[13] T. Iimura, H. Hazeyama, and Y. Kadobayashi. Zoned federation of game servers: a peer-to-peer approach to scalable multi-player online games. *Proceedings of ACM SIGCOMM 2004 workshops on NetGames' 04: Network and system support for games*, pages 116–120, 2004.

[14] B. Knutsson, H. Lu, W. Xu, and B. Hopkins. Peer-to-peer support for massively multiplayer games. *IEEE Infocom*, 2004.

[15] L. Lamport. The implementation of reliable distributed multiprocess systems* 1. *Computer Networks (1976)*, 2(2):95–114, 1978.

[16] B. Lampson. How to build a highly available system using consensus. *Distributed Algorithms*, pages 1–17, 1996.

[17] K. Lee and D. Lee. A scalable dynamic load distribution scheme for multi-server distributed virtual environment systems with highly-skewed user distribution. *Proceedings of the ACM symposium on Virtual reality software and technology*, pages 160–168, 2003.

[18] R. Minson and G. Theodoropoulos. An adaptive interest management scheme for distributed virtual environments. *Proc. of PADS '05*, pages 273–281, 2005.

[19] NCsoft. Lineage II, 2003. http://www.lineage2.com/.

[20] B. Ng, A. Si, R. Lau, and F. Li. A multi-server architecture for distributed virtual walkthrough. *Proceedings of the ACM symposium on Virtual reality software and technology*, pages 163–170, 2002.

[21] S. Rieche, M. Fouquet, H. Niedermayer, L. Petrak, K. Wehrle, and G. Carle. Peer-to-Peer-based Infrastructure Support for Massively Multiplayer Online Games. *Consumer Communications and Networking Conference, 2007. CCNC 2007. 2007 4th IEEE*, pages 763–767, 2007.

[22] G. Schiele, R. Suselbeck, A. Wacker, J. Hahner, C. Becker, and T. Weis. Requirements of Peer-to-Peer-based Massively Multiplayer Online Gaming. *Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, pages 773–782, 2007.

[23] N. Schiper and F. Pedone. On the inherent cost of atomic broadcast and multicast in wide area networks. *Distributed Computing and Networking*, pages 147–157, 2008.

[24] N. Schiper, P. Sutra, and F. Pedone. Genuine versus non-genuine atomic multicast protocols for wide area networks: An empirical study. In *Reliable Distributed Systems, 2009. SRDS'09. 28th IEEE International Symposium on*, pages 166–175. IEEE, 2009.

[25] F. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.