On the Design of a Scalable Architecture for Crowd Simulation

Miguel Lozano, Pedro Morillo, Juan M. Orduña, Vicente Cavero

Resumen— **Crowd simulations require both rendering visually plausible images and managing autonomous behaviors. Therefore, scalability is crucial for these applications. Although several proposals have focused on the software architectures for these systems, no proposals have focused on the computer systems supporting them.**

In this paper, we analyze the computer architectures used in the literature to support virtual environments. Also, we propose a computer architecture scalable enough to support simulations of thousand of autonomous agents. This proposal consists of a cluster of computers in order to provide scalability, as well as a client-server software architecture that efficiently provides consistency. Performance evaluation results show that the trade-off between scalability and consistency allows to efficiently manage thousands of autonomous agents.

Palabras clave— **Crowd simulation, autonomous agents, distributed virtual environments**

I. Introduction

Crowd simulations have become an essential tool for many virtual environment applications. Extensive use of virtual crowds has been made in many commercial movies like AntZ [1] or The Lord of the Rings [2]. Also, high quality crowd simulations are crucial for many virtual environment applications in education, training, and entertainment [3], [4].

Crowd simulations can be considered as virtual environment applications with two different goals. On the one hand, crowd simulations must focus on rendering visually plausible images of the environment, requiring a high computational cost. On the other hand, complex agents must have autonomous behaviors, greatly increasing the computational cost as well. Thus, some proposals tackle crowd simulations as a particle system with different levels of details (eg:*impostors*) in order to reduce the computational cost [5], [6]. Although these proposals can handle crowd dynamics and display populated interactive scenes (10000 virtual humans), they are not able to produce complex autonomous behaviors for their actors. On the contrary, several proposals have been made to provide efficient and autonomous behaviors to crowd simulations [3], [4]. However, they are based on a centralized system architecture, and they can only control a few hundreds of autonomous agents with different skills (pedestrians with navigation and/or social behaviors for urban/evacuation contexts). Tacking into account that pedestrians represent the slowest human actors (in front of other kind of actors like drivers in cars, for example) these results show that scalability has still to be solved in crowd simulations.

Although some scalable, complex multi-agent systems have been proposed [7], these proposals are exclusively focused on the software architecture, forgetting the underlying computer architecture (the actual implementation of the computer and the application executing on it [8]). As a result, important features like inter-process communications, workload balancing or network latencies are not taken into account. In this paper, we analyze the computer architectures used in the literature to support crowded virtual environments. In order to manage the trade off between scalability, rich behaviors, and computational cost required by crowd simulations, we propose a hybrid computer architecture. This architecture consists of a networked-server Distributed Virtual Environment (DVE) [9]. Also, in order to efficiently supporting consistency and autonomous behaviors we propose a centralized software architecture. Performance evaluation results show that this architecture can efficiently manage thousands of autonomous agents.

The rest of the paper is organized as follows: section II analyzes the existing computer architectures proposed in the literature for supporting Distributed Virtual Environments (DVEs). As a result of such analysis, section III describes the proposed architecture for crowd simulations. Next, section IV shows the performance evaluation of the proposed system architecture. Finally, section V shows some conclusion remarks.

II. Computer architectures for DVEs

Different computer architectures have been proposed in order to efficiently support DVEs: centralized-server architectures [10], networked-server architectures [11] and peer-to-peer architectures [12]. Figure 1 shows an example of each one of these architectures. In this example, the virtual world is two-dimensional and avatars are represented as dots. The area of interest (AOI) of a given avatar is represented as a circumference.

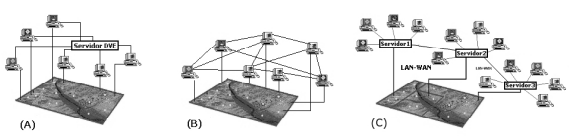

Fig. 1. Different architectures for DVE systems: a) Centralized b) Peer-to-peer c) Networked-server

Figure 1 a) shows an example of a centralized-server architecture. In this example, the virtual

This paper is supported by the Spanish MEC under Grant TIC2003-08154-C06-04

Departamento de Informática, Universidad de Valencia. e-mail: Juan.Orduna.@uv.es.

world is two-dimensional and avatars are represented as dots. In this architecture there is only a single server and all the client computers are connected to this server. Figure 1 c) shows an example of a networked-server architecture. In this scheme there are several servers and each client is exclusively connected to one of these servers. This scheme is more distributed than the client-server scheme, and since there are several servers, it considerably improves the scalability in regard to the client-server scheme. Finally, figure 1 b) shows an example of a peer-to-peer architecture. This scheme is the most distributed one, since each client computer is also a server.

Although the first DVEs were based on centralized architectures, during the last few years architectures based on networked servers have been the major standard for DVE systems [11]. However, each new avatar in a DVE system represents an increase not only in the computational requirements of the application but also in the amount of network traffic [13]. Due to this increase, networked-server architectures seem not to properly scale with the number of clients, particularly for the case of MMOGs [14], due to the high degree of interactivity shown by these applications. As a result, Peer-to-Peer architectures have been proposed for massively multi-player online games[12].

Nevertheless, P2P architectures must still efficiently solve the *awareness* problem. This problem consists of ensuring that each avatar is aware of all the avatars in its neighborhood [15]. Providing awareness to all the avatars is a necessary condition to provide time-space consistency (as defined in [16]). Awareness is crucial for DVEs, since otherwise abnormal situations could happen. For example, a game user provided with a non-coherent view of the virtual world could be shooting something that he can see although it is not actually there. Also, it could happen that an avatar not provided with a coherent view is killed by another avatar that it cannot see. In networked-server architectures, the awareness problem is easily solved by the existing servers, since they periodically synchronize their state and therefore they know the location of all avatars during all the time. Each avatar reports about its changes (by sending a message) to the server where it is assigned to, and the server can easily decide which avatars should be the destinations of that message (by using a criterion of distance). There is no need for a method to determine the neighborhood of avatars, since servers know that neighborhood every instant.

III. A New Architecture for Crowd Simulation

From the discussion above it seems that the more physical servers the DVE relies on, the more scalable it is. On the contrary, features like the awareness and/or consistency are more difficult to be provided as the underlying architecture is more distributed (peer-to-peer architecture). Therefore, we propose a networked-server scheme as the computer system architecture for crowd simulation. On the one hand, this distributed scheme allows to improve scalability (the maximum number of agents in the system) when compared to centralized (client-server) architectures. On the other hand, the small number of servers in networked-server architectures makes easy to provide awareness (and therefore time-space consistency) to the avatars moving in the virtual world.

On top of this networked-server architecture, a software architecture must be designed to manage a crowd of autonomous agents. In order to easily maintain the coherence of the virtual world, a centralized semantic information system is needed. In this sense, it seems very difficult to maintain the coherence of the semantic information system if it follows a peer-to-peer scheme, with lots of computers supporting each one a small number of actors and a copy of the semantic database. Therefore, on top of the networked-server computer system architecture, we propose the software architecture shown in figure 2. This architecture has been designed to distribute the agents of the crowd in different server computers (the networked-servers). We denote this computers as *the clients*, and each one of them can manage a variable number of autonomous agents.

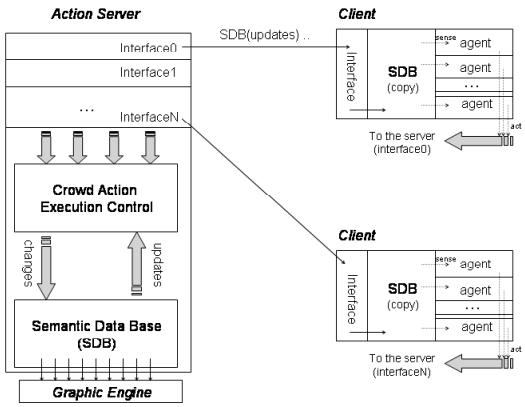

Fig. 2. The proposed software architecture

This centralized software architecture is composed of two elements: the *Action Server (AS)* and the *Client Processes (CP)*. In turn, the action server is composed of two different modules, Semantic Data Base (SDB) and the *Action Execution Module (AEM)*.

A. The Action Server

The AS can be viewed as the world manager, since it controls and properly modifies all the information the crowd can perceive. The Action Server is fully dedicated to verify and execute the actions required by the agents, since they are the main source of changes in the virtual environment. For scalability purposes, the AS must be placed on the computer with the highest computational power. This computer should exclusively used for this purpose. Since the AS is unique, consistency is easily provided. In this context, consistency involves the information that the agents

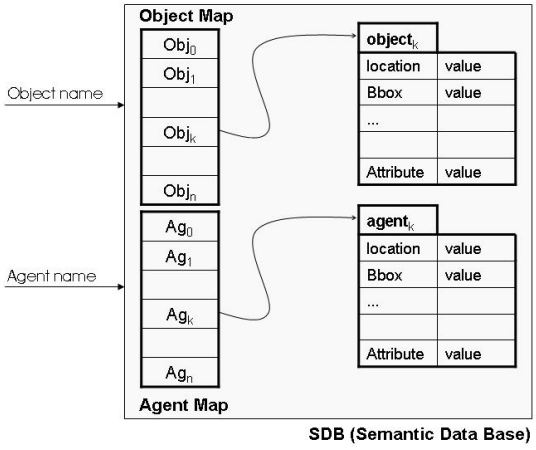

Fig. 3. The Semantic Data Base

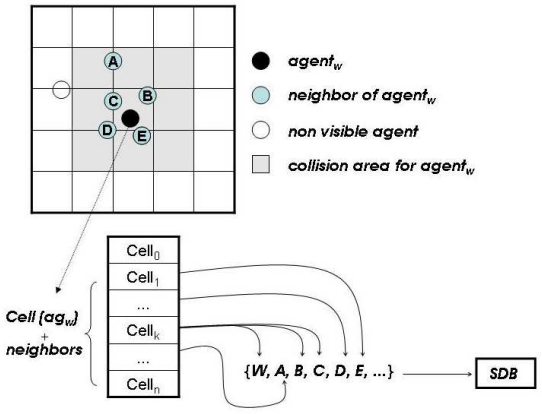

Fig. 4. A collision example.

should know to animate consistent behaviors. Additionally, another important parameter for interactive crowd simulations is the *server main frequency*. This parameter represents how fast the world can change. Ideally, in a fully reactive system all the agents send their action requests to the server, which processes them in a single cycle. In order to provide realistic effects, the server cycle must not be greater than the threshold used to provide quality of service to users in DVEs [17]. Therefore, we have set the maximum server cycle to 250 ms..

The server can be viewed as the world manager, since it controls and properly modifies all the information the crowd can perceive. It consists of two modules: the Semantic Data Base and the Action Execution Module. The SDB represents the global knowledge about the interactive world that the agents should be able to manage, and it contains the necessary functionalities to handle interactions between agents and objects. In our case, we manage two maps (objects, agents) which let us to efficiently control a set of (attribute, value) pairs associated to each object/agent during the simulation. Since these attributes are centered into objects or agents, we use their names to index the correspondent map. The semantic information managed can be symbolic (eg: $object_i$ free true, $object_i$ on $object_k$, ...) and numeric (eg: $object_i$ position, $object_i$ bounding volume, ..), since it has been designed to be useful for different types of agents. Figure 3 illustrates this data base scheme.

We have decided to avoid complex spatial maps (such as cuad/oct-trees) to control the SDB, since these structures can be too expensive to handle when the number of insertions and deletions grows (agents can be continually changing their location). Instead, we use STL maps, which allows us to access to any object/agent with a logarithmic cost. Also, in order to face the collision detection in a scalable way, we reduce the visible area for each agent. Concretely, we use a vector of cells (2D-grid) to manage the list of the elements contained in each collision area. Figure 4 shows an example of such cells. Once the server frequency has been set (in our case to 250 milliseconds), the available time to process each agent action results from dividing the server frequency by the number of agents in the system. If the number of agents increases in such a way that the server is not be able to process all the actions in a single cycle, the pending actions are simply left to be processed in the next cycle. In the experiments shown in section IV we measure the action latency, so we could estimate the degree of reactivity achieved by these scheme.

The Action Execution Module manages the action's flow of the simulation. In order to allow the maximum flexibility, the AS can process 4 types of actions:

Motion actions : Location changes where collisions can occur, although agents were (potentially) able to navigate without colliding. If an agent wants to move to the location currently occupied by another object/agent, the environment should simply not allow it. Figure 4 illustrates an example of a collision situation for $agent_w$. Since the server knows the location of the agent, it can access to its cell through a simple function (similar to the one used by hash tables), and perform an object-object collision test with the neighbors of $agent_w$.

Motor actions : We use simple key-framing tables to animate the actors in walking, running, and other motions. Since no constraints are allowed, the value received is simply accepted in the SDB as an internal change in an agent attribute. We consider the agent motor system as the responsible to continually read its *animation state* from the SDB. The graphic engine, which contains all the *actor's skeletons* of the crowd, will perform this task according to its frame rate.

Agents interactions : Corresponds to a normal agent-agent communication scheme, which can be easily obtained from the system through the server. Messages can be managed as other agent attributes, so the SDB will simply route them into the correspondent slots.

STRIPS actions : STRIP is the action language used by our planning agents. A STRIPS ac-

tion scheme [18] can be represented through the *Preconditions*, *Add* and *Delete* lists associated to each agent action. Before executing an action (eg: pick up an object), the server verifies its preconditions using the SDB maps (eg: is object-k free?). When a STRIPS action is accepted, the Add and Delete lists contain the new information the SDB needs to update its state.

When an action is positively checked, it should be executed and (possibly) the SDB updated. In order to do this, this module puts all the action effects in a new vector *(vUpdate)* which reflects the local changes produced by each actuation (eg: an agent changes its position). Finally, when the server cycle has finished, this vector is sent to both, the clients and the SDB, which will update their correspondent environmental states.

For example, when an agent tries to perform a motor action (a movement) then a collision can occur. The agent should request the server to validate that movement by sending a message. Since the server knows the location of the agent, it accesses to its cell through a simple function (similar to the one used by hash tables), and perform an object-object collision test with the neighbors of $agent_w$. If no collision caused by that movement is detected, then the server should send an acknowledgment message to the agent.

B. The clients

Each process in a client computer (the rest of the networked servers not used as the AS) manages an independent group of autonomous agents (a subset of the crowd), and it is executed in a single computer as a single process. This process has an *interface* for receiving and updating the information from the server, and a finite number of threads (each thread for an agent). Using this interface, a client initially connects to the Action Server and downloads a complete copy of the SDB. From that instant, agents can think locally and in parallel with the server, so they can asynchronously send their actions to the server, which will process them as efficiently as possible (since each agent is a process thread, it can separately access to the socket connected to the server). When a server cycle finishes, the accepted changes are submitted to all the clients interfaces, that will update their SDB copies.

The proposed multi-threading approach is independent of the agent architecture (the AI formalism driving the agent behavior), that is out of the scope of this paper. However, the proposed action scheme guarantees the awareness for all agents [15], since all the environmental changes are checked in a central server and then broadcasted to the agents. Although time-space inconsistencies can appear due to agent asynchronies and network latencies, all these inconsistencies are kept below the limit of the the server period.

As an example, figure 5 shows a snapshot of a real system with the architecture described in this section. For the sake of clearness, this snapshot has been taken for a simulation of only 1000 agents. This figure shows an overall view where agents are represented as black dots. In this example we have simulated a square bi-dimensional world, all agents were initially located in the world following a random distribution inside one half of the world, and they moved trying to arriving to the opposite side of the world.

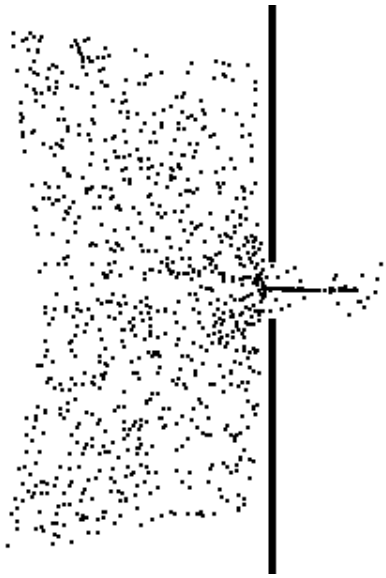

Fig. 5. Snapshot of a simulation of 1000 agents

IV. Performance Evaluation

This section shows the performance evaluation of the architecture described in the previous section. In order to achieve such goal, we have performed measurements on a real system with this architecture. Concretely, we have performed simulations where each agent keeps moving during 5 or 6 minutes. As cited in the previous section, the AS cycle (the maximum period of time an interactive actor can wait for it action response) has been set to 250ms., and every 2.5 seg. statistics are computed, resulting in 30 different samples. Each point of the plot and each value of the tables in this section has been computed as the average value of the 30 samples. In order to make a performance evaluation we have used wandering agents, since this type of agents is the one that generate the highest workload to the AS (since they simply move, they require the server to validate their movements in each server cycle).

Like in other distributed systems, the most important performance measurements in DVE systems are latency and throughput. First, we have focused on system throughput (the number of agents that the system can efficiently support), that is limited in our architecture by the AS throughput. Concretely, we have measured the AS throughput when it is fully dedicated to collision detection tasks. The rationale of this test is to evaluate the number of actions that the server is able to carry out in a single cycle, since this could be a plausible bottleneck.

When an action is requested by an agent, the server basically must access to its cell and then it must compute a set of simple distance checking against the agents which are sharing the same cell, as figure 4 shows. If no collisions are produced, then this process continues until the 8 neighbor cells pass the same test.

Figure 6 shows the results obtained in a collision detection test performed in an isolated server while handling 10.000 agents demanding a random position as soon as they can, in order to saturate the server. The purpose is to know how fast the server can run, that is, the average of actions that it is able to process in a single cycle. As figure 6 shows, this value highly depends on the density of the crowd. Nevertheless, we have represented this parameter as the percentage of finally executed actions (ACK's), since it is more informative for our purposes. Thus, an ACK percentage of 0% occurs when no motion is allowed because the crowd is completely full and no one can move. On the other hand, when all the agents pass a full collision test, all the actions are allowed (100% of ACK's) and all the agents finally move. In these experiments, this case (94% of ACK's) represents the worst case because the server should access to 8 + 1 cells and compute a variable number of distance checks for each action.

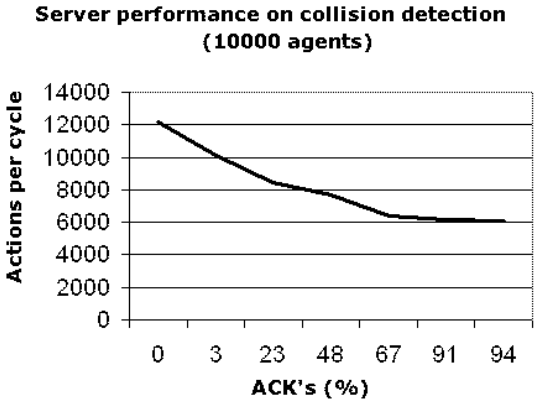

Fig. 6. Server performance on collision tests

Figure 6 shows that in the worst case (94% of ACK's) the server is able to process around 6000 actions in a single cycle (250 ms). However, when the density of the crowd increases the percentage of ACK's decreases because the probability of collision increases in very dense worlds. This will produce that the server can finish the cell checking without visiting all the neighbors cells. As a consequence, the server can process a higher number of actions requests per cycle (12000 actions for a percentage of 0% ACKS). It is also worth mention that for a medium case (48% ACK's), the system can manage around 8000 agents.

Additionally, we have evaluated the throughput and the latency of the entire system in a configuration composed of 4 clients and 1 server. For both cases, agents continuously demand a random position to the server. The purpose of these evaluations is to check if the proposed architecture can improve the latency and throughput provided by other general purpose DVEs [13]. In these experiments, the existing agents are distributed among the existing client computers, in such a way that all the client computers manage a similar crowd subset (group of agents). We have increased the number of agents until the whole system has entered saturation (the system response time greatly increases and the CPU utilization reaches 100% in any computer [13]).

TABLA I
System throughput for a fixed agent cycle of 250ms.

	S0	C1	C2	C3	C4
Agents	%	%(cy.)	%(cy.)	%(cy.)	%(cy.)
1000	11	1(1)	1(1)	1(1)	1(1)
2000	22	4(1)	4(1)	4(1)	4(1)
3000	32	10(1)	10(1)	11(1)	10(1)
4000	43	18(1)	18(1)	18(1)	17(1)
5000	54	24(1)	27(1)	25(1)	23(1)
6000	64	31(1)	32(1)	30(1)	32(1)
7000	79	39(1)	40(1)	37(1)	36(1)
8000	94	45,0(1)	48(1)	45(1)	46(1)
9000	97	50(1,5)	49(1,3)	50(1,4)	49(1,3)
10000	98	48(2,0)	48(1,6)	48(1,7)	48(1,6)

Table I shows the results for different simulations with the same AS cycle (250 ms.) but with different numbers of agents. Each column in this table shows two different values, except for the server. The column labeled with S0 shows the percentage of CPU utilization reached in the server. The columns labeled with Cx show the percentage of CPU utilization reached in that client and the average response time (measured in AS cycles) for the agents supported by that client. The left column show the number of agents used in each simulation. Each row in this table shows the results for a simulation with a different number of agents, ranging from 1000 to 10,000 agents.

Table I shows that, as it could be expected, the system bottleneck is the AS, since it shows the highest percentage of CPU utilization in all the rows. This table also shows that the system provides acceptable response times with up to 8000 agents. Although the CPU utilization in the AS is close to saturation (94%), the response times in all the clients are kept below the AS cycle. That is, the AS is able to process all the requests in a single cycle. However, when the system is supporting 9000 or 1000 then the AS can only serve part of such requests, increasing the average response time up to 2,0 cycles (client C1 for ten thousand agents). It is worth mention that none of the clients reaches a CPU utilization of more than 50%, showing that there is a single bottleneck (the AS).

Table II also shows the results for different simulations performed with different numbers of agents. However, in these cases we have studied the minimum response times that can be achieved. In order to achieve such goal, for each number of agents we have adjusted the AS cycle until either it has reached

TABLA II
RESPONSE TIMES OBTAINED AT MAXIMUM THROUGHPUT WHEN SUPPORTING DIFFERENT NUMBERS OF AGENTS

	S0	C1	C2	C3	C4
Agents	%	%(s.)	%(s.)	%(s.)	%(s.)
1000	28	2(0,1)	3(0,1)	4(0,08)	3(0,11)
2000	59	12(0,1)	13(0,1)	13(0,09)	16(0,1)
3000	90	36(0,1)	35(0,1)	43(0,08)	37(0,1)
4000	93	49(0,11)	48(0,11)	48(0,13)	48(0,13)
5000	96	50(0,15)	50(0,15)	50(0,16)	50(0,15)
6000	94	51(0,19)	51(0,22)	52(0,18)	52(0,18)
7000	95	52(0,23)	52(0,21)	51(0,23)	53(0,21)
8000	97	47(0,25)	48(0,25)	46(0,25)	46(0,25)
9000	97	45(0,29)	47(0,29)	46(0,31)	46(0,3)
10000	97	43(0,36)	43(0,34)	43(0,35)	42(0,34)

a CPU utilization close to saturation (90-97%) or until the CPU utilization did not increase (that number of agents was not enough to saturate the AS). Therefore, the column S0 in table II shows values equal or higher than 90% for the last eight rows, and the average response times for the agents supported by each client (the values in parenthesis) are expressed in seconds.

Table II shows that for 3000 or less agents in the system the average response times in all the clients is below 0.1 s, and these average response times increase as more agents are in the system. When comparing tables I and II and figure 6, it can be seen that they provide coherent results. Effectively, the results in table I are obtained with an AS cycle of 250 ms., and that table shows that the system can support up to 8000 agents while providing average response time of 1 cycle in all the clients. Table II shows that for 8000 agents the average response times in all the clients are 250ms.. That is, the system can manage up to 8000 autonomous agents if the AS cycle is 250ms.. These results are obtained with a 50% of positive server acknowledgment, and therefore they agree with the ones in figure 6, where the server process around 8000 actions (one per agent and cycle) for a moderately dense world (48% of ACKs). Effectively, since the agents considered for performance evaluation exclusively move following a random pattern, they generate the highest number of requests as possible to the AS server, that is the system bottleneck. Therefore, more complex agents will require more time between successive requests to the AS, thus allowing the system to support a higher number of agents. Therefore, these results validate the proposed scheme (a computer architecture based on networked servers and a client-server software architecture) as a trade-off between scalability and consistency.

V. CONCLUSIONS

In this paper, we have proposed a scalable hybrid architecture for crowd simulations. On the one hand, this architecture consists of a computer system based on networked server, in order to achieve scalability while providing awareness and time-space consistency. On the other hand, it consists of a software architecture based on centralized semantic information system that can easily maintain the coherence of the virtual world (there is a single copy of the semantic database). Performance evaluation results show that this architecture can efficiently manage thousands of autonomous agents. Concretely, for the case of an AS cycle of 250ms., this scheme can handle at least 8000 autonomous agents.

As a future work to be done, we plan to characterize the requirements of different kinds of autonomous agents. In order to improve the scalability of the proposed scheme, the idea is to use each client for supporting one (or more) kind of agents, according to the computational power of the client and the requirements of the agents. Thus, by properly balancing the existing load among the clients we expect to improve the system throughput.

REFERENCIAS

[1] ," AntZ: http://www.pdi.com/feature/antz.htm.
[2] ," Lord of the rings: http://www.lordoftherings.net.
[3] Paul A Kruszewski, "A game-based cots system for simulating intelligent 3d agents," in *BRIMS '05: Proceedings of the 2005 Behavior Representation in Modelling and Simulation Conference*, 2005.
[4] Mankyu Sung, Michael Gleicher, and Stephen Chenney, "Scalable behaviors for crowd simulations," in *Proceedings of the 2004 ACM SIGGRAPH/Eurographics symposium on Computer animation*. 2004, pp. 519–528, ACM Press.
[5] Simon Dobbyn, John Hamill, Keith O'Conor, and Carol O'Sullivan, "Geopostors: a real-time geometry/impostor crowd rendering system," *ACM Trans. Graph.*, vol. 24, no. 3, pp. 933–933, 2005.
[6] Franco Tecchia, Celine Loscos, and Yiorgos Chrysathou, "Visualizing crowds in real time," *Computer Graphics Forum*, vol. 21, 2002.
[7] Huaglory Tianfield, Jiang Tian, and Xin Yao, "On the architectures of complex multi-agent systems," in *Proc. of the Workshop on "Knowledge Grid and Grid Intelligence", IEEE/WIC International Conference on Web Intelligence / Intelligent Agent Technology,*. 2003, pp. 195–206, IEEE Press.
[8] Dezso Sima, Terence Fountain, and Peter Karsuk, *Advanced Computer Architectures : A Design Space Approach*, Addison Wesley, 1997.
[9] S. Singhal and M. Zyda, *Networked Virtual Environments*, ACM Press, 1999.
[10] ," Quake: http://www.idsoftware.com/games/quake/quake/.
[11] John C.S. Lui and M.F. Chan, "An efficient partitioning algorithm for distributed virtual environment systems," *IEEE Trans. Parallel and Distributed Systems*, vol. 13, 2002.
[12] S. Mooney and B. Games, *Battlezone: Official Strategy Guide*, BradyGame Publisher, 1998.
[13] P. Morillo, J. M. Orduña, M. Fernández, and J. Duato, "Improving the performance of distributed virtual environment systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 16, no. 7, pp. 637–649, 2005.
[14] T. Alexander, *Massively Multiplayer Game Development II*, Charles River Media, 2005.
[15] Randall B. Smith, Ronald Hixon, and Bernard Horan, *Collaborative Virtual Environments*, chapter Supporting Flexible Roles in a Shared Space, Springer-Verlag, 2001.
[16] S. Zhou, W. Cai, B. Lee, and S. J. Turner, "Time-space consistency in large-scale distributed virtual environments," *ACM Transactions on Modeling and Computer Simulation*, vol. 14, no. 1, pp. 31–47, 2004.
[17] T. Henderson and S. Bhatti, "Networked games: a qos-sensitive application for qos-insensitive users?," in *Proceedings of the ACM SIGCOMM 2003*. 2003, pp. 141–147, ACM Press / ACM SIGCOMM.
[18] Richard Fikes and Nils Nilsson, "Strips: a new approach to the application of theorem proving to problem solving," *Artificial Intelligence*, vol. 5, no. 2, pp. 189–208, 1971.