The Perils of Using Simulations to Evaluate Massively Multiplayer Online Game Performance

Alexandre Denault
School of Computer Science, McGill University
Montreal, QC H3A 2A7, Canada
alexandre.denault@mail.mcgill.ca

Jörg Kienzle
School of Computer Science, McGill University
Montreal, QC H3A 2A7, Canada
Joerg.Kienzle@mcgill.ca

ABSTRACT

To test and benchmark Massively Multiplayer Online Games (MMOGs) requires hundreds, if not thousands of human players. Given this impracticality, many researchers substitute experimentation with simulation. However, when investigating performance and scalability issues, simulated experiments often yield results that heavily depend on the experimental setup. This paper critically reflects on the use of simulation to conduct experiments in MMOGs. Using Mammoth, a MMOGs research framework, performance measurements such as CPU usage, memory usage and used network bandwidth are collected while running the same game scenario using five different simulation setups. The results are analyzed, and the discovered differences are discussed. The paper concludes that experiments which are aimed at measuring the performance of a MMOGs using simulations must be designed very carefully, especially if they are run on a single machine.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*performance measures*; C.2.4 [**Computer Communication Networks**]: Distributed Systems, Distributed Applications

General Terms

Simulation, Performance, Massively Multiplayer

Keywords

Simulation of Games, Performance Measurements, Design and Implementation of Massively Multiplayer Online Games

1. INTRODUCTION

In the last decade, the video game industry has shown unparalleled growth, both in revenue and in development complexity. With the advent of the Internet, multiplayer and massively multiplayer games have become more and more popular. Compared to a traditional multiplayer game in which usually up to 16 players play a relatively short-lived game, Massively Multiplayer Online Games (MMOGs) offer the possibility for thousands of players to play together in a persistent world.

The biggest challenge in MMOGs is *scalability*: the aim is to allow as many players as possible to play together in the *same* virtual world. Presenting a consistent view of the virtual world to all players in real-time is difficult, because the machines of the players can be located anywhere on the Internet. As a result, the quality of the network connection to individual nodes varies: some connections exhibit a higher latency than others, meaning that it takes more time for a message to reach its destination. Bandwidth, i.e. the maximum throughput of data to and from a given node, is also limited, and varies depending on the quality of the connection. Finally, any one machine on the network has itself limited processing power and memory.

Given the nature of a MMOG, to accurately conduct experiments to measure the performance of an MMOG implementation, hundreds of human players would be required. This is, of course, unpractical, especially if repeated experiments are to be conducted. Thus, it is common to run experiments where the players are simulated [18, 21]. Player behaviour, also called *Artificial Intelligence* (AI) in computer games, is usually programmed using some sort of scripts [19], or even modelled using a graphical modelling notation [12, 15, 23].

Ideally, during an experiment, each of the AI-controlled players should be connecting to the game just like human players would, i.e. from a separate computer. Unfortunately this requires hundreds of computers, which, again, may be unpractical for several reasons. First of all, the researcher running the experiment must have access to such a significant number of machines. Second, the researcher must find a way to automate the starting of the game on each of the machines, since physically launching the game on each player machine for each experiment would be prohibitively time consuming.

Because of this, many researchers turn to simulations in order to conduct MMOG experiments. While this drastically simplifies the effort it takes to run the experiments, great care must be taken to ensure that the obtained measurements indeed reflect the "real-world", i.e. that they correctly reflect the behaviour of a real MMOG game instance. As for commercial MMOGs, very little information is available on how game companies test the scalability and performance of the game architecture, algorithms and techniques

DISIO 2010 March 15, Torremolinos, Malaga, Spain.

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8632
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8632

they use in their games. However, discussions with MMOG providers seem to indicate that performance is often evaluated experimentally by allowing a limited set of users to play a "beta" version of the game for free.

This paper critically reflects on the use of simulated experiments in the context of MMOGs, in particular when the experiments are aimed at evaluating game performance and server scalability. Section 2 presents related work that uses simulations in the context of MMOGs. Section 3 introduces Mammoth, the MMOGs research framework we used to conduct our experiments for this paper. Section 4 describes the different experimental settings we used to evaluate the performance of Mammoth, and presents the gathered measurements. Section 5 reflects on the obtained results and the last section draws some conclusions.

2. RELATED WORK

The idea of creating a distributed game infrastructure is decades old, with works such as [10] formally introducing the concept. This particular work introduces Mimaze, a distributed game that leverages multicasting for communication. The paper introduces many key issues, such as state inconsistencies and the possibility of cheating. However, Mimaze did not deal with scalability issues and was thus limited to 25 players.

Since then, there has been no shortage of work proposing distributed architectures for games on a massive scale [2, 3, 6, 7, 11, 14, 16] . Several of these ideas are based either on Pastry [22], which describes a communication infrastructure for large scale Peer-to-Peer (P2P) systems, or Mercury [4], which proposes a publish/subscribe infrastructure that could be used for large scale games.

Unfortunately, very few of these distributed architectures were tested on a large number of nodes. For example, [17] doesn't describe any environment setup, leading us to believe that the results are purely simulated on a single node. [2] also lacks an environment description, but cites the use of micro benchmarks to estimate the maximum capacity of the framework. On the other hand, [7] describes an experimental setup, but the experiments were run on only 6 machines.

As mentioned previously, several of these architecture are built on top of Pastry, more specifically, FreePastry [1]. Both [16] and [11] were benchmarked using the simulator provided with FreePastry, creating several thousand nodes on a single machine. [8] describes a similar scenario, creating traces for 6000 players and executing them on a single machine simulator. [24] describes how the realism of simulated benchmarks can be be improved by introducing artificial network delays (lag). For example, experimentation has shown that slowing down nodes by 10% in a P2P messaging system can increase the transmission latency from 300ms to as much as 800ms.

Performance measurement and evaluation can also be done at lower layers of architecture, such as the network layer. For example, the tools provided by OPNET [20] can be used to simulate and evaluate new network protocols. In fact, there is no shortage of network simulation tools currently available on the market. However, the quality of a network simulation relies mainly on the quality of the network trace used as input. This network trace can be generated by user experimentation, but is often generated using a higher-level game simulation.

Although simulations can be used to evaluate certain characteristics of a MMOG implementation, it does not replace testing and benchmarking using a large number of players. [21] proposes a solution to this by using computer-controlled (AI) players. This has the advantage of closely reproducing game conditions without the unpredictability of real human players. Possible behaviours for such AI players can be found in [18], where actions are decided using either a greedy approach, Markov chains or hierarchical plans. [3] illustrates a successful use of agent-based benchmarking, by using unmodified Quake II bots (computer controlled players) to generate load to evaluate the performance of their distributed Quake II servers.

Figure 1: Screenshot of Mammoth

3. MAMMOTH

Mammoth is a MMOGs research framework. It was created as a collaborative project between a group of McGill professors and students in early 2005, and has evolved considerably during the last 4 years. Its goal is to provide an implementation platform for academic research related to multiplayer and MMOGs in the fields of distributed systems, fault tolerance, databases, networking, concurrency.

In Mammoth players take control of a game character, also called an avatar. A game session consists of moving around in a virtual world and interacting with the environment by executing actions (see figure 1). Basic building blocks of such actions are, eġ; moving the avatar, picking up or dropping items, or communicating with other players.

In multiplayer and MMOGs, in order to provide a shared sense of space among players, each player must maintain a copy of the (relevant) game state on his computer. When one player performs an action that affects the world, the game state of all other players affected by that action must be updated. Different strategies for the distribution of the game state can have a profound impact on the scalability, consistency and performance of the game.

Since Mammoth is specifically designed for experimentation with different distribution approaches, Mammoth defines an intuitive object-based interface between the game layer and the framework components that handle distribution. All game state is encapsulated in *game objects*, which are objects that provide operations to manipulate the state in a consistent way. The developer then proceeds to map

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8632
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8632

game objects to *replicated objects*, an abstraction provided by the Mammoth replication engine. Whenever a player needs to access the game state encapsulated in a replicated object, a copy of the object – a *replica* – is created on the player's machine. *Read* operations that are called on the game object are executed on the state of the replica. Modifying operations, however, cannot be executed locally for consistency reasons. They are forwarded to a specially designated replica called the *master*, which executes the modifying operation and then broadcasts the new state to all the replicas.

This remote method invocation is completely transparent for the game developer, which is not only convenient, but also allows Mammoth to migrate a master from one node to another node for load balancing, fault tolerance, or cheat prevention reasons. The interested reader is referred to [9] for further details on the replicated objects technology in Mammoth, such as how it addresses interest management and dead reckoning.

3.1 Mammoth's Flexible Architecture

In order to allow researchers to easily conduct experiments, the Mammoth framework has been designed as a collection of collaborating components that each provide a distinct set of services. The components interact with each other through two types of well-defined interfaces, engines and managers. Engines are core components that can be completely replaced to experiment with alternative implementations. Managers are components designed to manage multiple implementations of a given algorithm or strategy.

For example, there are currently 4 different network engine implementations that can be used for communication within Mammoth. The *SternNetworkEngine* is an engine that implements TCP/IP based communication using a star topology, where one node is designated as a hub which takes care of forwarding and broadcasting messages. *ToileNetworkEngine* implements a fully connected network topology, and *Postina* implements a dynamic Peer-2-Peer network topology using Pastry [22]. *FakeNetworkEngine* is a network engine implementing communication within a single process using a shared memory structure. The use of the *Factory* design pattern [13] makes it possible to switch engines without modifying any code.

An example of a manager is the non-player character (NPC) manager, which can be used to simulate player behaviour. The NPC manager allows game designers to write code, also named an *agent*, that takes control of an avatar and executes actions just like a human player would. Many different control algorithms have been implemented, such as *RandomWanderer*, *ItemGatherer*, *Follower*, *WaypointVisitor*, etc. Through a configuration file, the developer can specify which control algorithm should control which avatar.

Finally, the core of a Mammoth process is the *Node* object. When a Mammoth process starts up, it always creates a *Node* instance. Subsequently, different *tasks* are assigned to the node. For example, a graphical Mammoth client that humans can use to play the game is created by assigning the *JMonkeyGraphicalClient* task to a node. A simple server component can be created by assigning to a node the task to *manage all the master objects* in the game world. To create an computer-controlled player, the *NPC Controller* task is assigned to a node object. The *Webmonitor* task runs a web server on the node that makes it possible to remotely look at the Mammoth game data structures at run-time using a standard web browser.

Of course, it is also possible to assign several tasks to a node. For instance, to observe the behaviour of a newly created NPC control algorithm, a *NPC Controller* task and a *JMonkeyGraphicalClient* task are assigned to a node. Likewise, it is possible to assign several *NPC Controller* tasks to a single node object to control several avatars from within the same process.

4. EXPERIMENTS

The experiments presented in this paper are aimed at demonstrating the fragility of performance measurements obtained in a simulated MMOG environment. The experimental environment we used is detailed in subsection 4.1. The 5 different simulation setups we evaluated are presented in subsection 4.2 and the results are summarized in subsection 4.3.

4.1 Experimental Environment

All experiments presented in this paper were conducted using the Mammoth framework. For all the experiments, the performance measurements were taken on a machine with Quad-Xeon 2.6 GHz processors and 8 gigabytes of main memory. In addition to this machine, the distributed experiment also used 60 additional computers from the McGill School of Computer Science computer labs. Each of these machines is equipped with a processor that runs at a minimum speed of 2 GHz and with at least 2 GB of memory.

All experiments were run on the same virtual game world, which included multiple obstacles, such as buildings and trees, and hundreds of game objects for players to pickup. The *RandomWanderer* NPC control algorithm was used to control the behaviour of the avatars. It instructs the controlled game character to wander randomly in strait lines, changing direction approximatively every second[1]. Each change of direction requires to execute a modifying operation on the replicated object that represents the avatar. This results in a remote method invocation from the node on which the NPC controller is running to the node that hosts the master object of the avatar.

As mentioned earlier, the Mammoth game architecture is scalable. Master objects can be located on any node, and as more players join the game, master objects can be migrated from machine to machine to balance the load among all connected nodes. Although such flexibility is required in order to support thousands of players, the experiments we conducted for this paper always used a single node to host all the master objects. This node is subsequently called the *server*. Having a single server makes sense in our case, since we are not trying to evaluate the scalability of Mammoth, but rather evaluate the effects different simulation setups can have on performance measurements. To simulate worst-case load on the server, a "complete view" interest management algorithm was used. As a result, every game state modification is broadcast to all other nodes, and therefore all players can see the movement of all other players. Again, Mammoth supports more elaborate interest manage-

[1]Although the choice of direction is also random, previous experiments with approx. 40 students [5] have shown that this type of computer-controlled players generate similar work load as real human players.

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8632
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8632

ment algorithms that avoid sending unnecessary state updates to players, but this would just add unnecessary, player position-dependent noise to our performance measurements.

The course of the individual experiments was always the same. NPC controllers were started one at a time in 30 seconds intervals. This gave the server node plenty of time to setup a particular NPC player before the next one attempted to connect. Three metrics were measured during each experiment: the CPU utilization of the server process, the memory allocated to the server process and the network throughput on the process' network interface. Readings were taken when the server achieved a load of 5,10,15,20,25,40, 50 and 60 clients. The cap of 60 players was chosen because that was the highest number of players that all configurations were able to support.

4.2 Evaluated Simulation Setups

This subsection describes the various simulation configurations used to run the experiments. All the variations were easily implemented in Mammoth by switching the network engine (or removing it completely) and by running the NPC agents in the same or in different nodes, or even in different processes. A summary of the simulation setups is presented in Table 1.

4.2.1 No (Network) Engine

- Measured node contains:
  - Server task
  - All NPC controller tasks running on 1 thread
  - No replication engine
  - No network engine

In the simplest simulation setup, both the server component and the NPC controllers are executed using a single thread within the same process. At every clock iteration, the agents are queried in a round-robin fashion for new instructions. Agents interact directly with the objects found in the server component, thus no network engine is needed. No objects are ever replicated and consequently there is no need for interest management either.

It should be noted that since the NPC controllers run in the same process as the server component, the measurements for this scenario include the load of both the server component and the NPCs.

4.2.2 Fake (Network) Engine

- Measured node contains:
  - Server task
  - All NPC controller tasks, 1 threads per NPC controller
  - Replication engine with "complete view" interest management
  - Fake network engine

In this simulation setup, each NPC has its own environment and runs in its separate thread. This means, when a new NPC connects to the server component, it receives a copy of every object it is interested in. It can then interact with the world by executing actions on those objects.

Communication between NPCs and the server component is achieved using the *FakeNetworkEngine*, which communicates using a series of data structures. Messages are transmitted instantly, as they are written to and read from main memory. The *FakeNetworkEngine* for Mammoth was first developed to simplify unit testing, but has since then proven to be a valuable simulation tool when working on a single node.

It should be noted, again, that since the NPC controllers run in the same process as the server component, the measurements for this scenario include the load of both the server component and the NPCs.

4.2.3 Local Engine

- Measured node contains:
  - Server task
  - All NPC controller tasks, 2 threads per NPC controller
  - Replication engine with "complete view" interest management
  - Stern network engine

This simulation setup is similar to the Fake Engine setup, but this time a real socket-based communication engine is used. Based on the *SternNetworkEngine*, this simulation configuration uses a central communication hub that all nodes connect to it. For the purpose of this paper, the hub is always spawned by the server, allowing all players to connect to a centralized location.

As with the previous setup, both server processes and NPCs are located in the same process. However, each NPC has to spawn 2 threads, as one additional thread is required to monitor the sockets for incoming traffic. In addition, all communication is routed through the localhost network interface.

Like in both previous scenarios, since the NPC controllers run in the same process as the server component, the benchmark for this scenario includes the load of both the server component and the NPCs.

4.2.4 Local Engine Separate NPC

- Measured node contains:
  - Server task
  - Replication engine with "complete view" interest management
  - Stern network engine
- Other node, running on the same machine, contains:
  - All NPC controller tasks

This simulation setup is identical the previous one except for one key difference, which is that the NPC agents are spawned in a separate process. This means that although the computer used to run the experiments is subject to the same load, only the load on the process that runs the server component is measured. All communications between the server component and the NPC agents is again routed thought the localhost interface, using the Stern Network Engine.

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8632
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8632

Table 1: Overview of Different Simulation Setups

	No Engine	Fake Engine	Local Engine	LE Separate NPC	Distributed Engine
NPC included in CPU Load	Yes	Yes	Yes	No	No
NPC included in Memory Usage	Yes	Yes	Yes	No	No
Bandwidth Measurable	No	No	Yes	Yes	Yes
Interest Management Enabled	No	Yes	Yes	Yes	Yes
Remote Method Invocations	No	Yes	Yes	Yes	Yes
Number of Node Objects	1	n+1	n+1	n+1	n+1
Number of Processes	1	1	1	2	1+n
Number of Threads	1	n+1	2n+2	2n+2	2n+2
Number of Computers Used	1	1	1	1	n+1

4.2.5 Distributed (Network) Engine

- Measured node contains:
  - Server task
  - Replication engine with "complete view" interest management
  - Stern network engine
- Other nodes, running on different machines, contain:
  - 1 NPC controller task

This simulation setup moves the NPC agents to different computers, alleviating the workload of the computer hosting the server component. This setup is the most realistic, as it simulates different players using different computers connecting to a centralized server.

As with the previous scenario, only the load from the server component is measured.

4.3 Results

The CPU utilization, the memory allocated, and the data throughput on the network interface of the process running the server task are shown in Figures 2, 3 and 4.

The measurements were gathered until a total of 60 players were connected to the server. The Local Engine configuration was not able to scale beyond that point because of a bandwidth and CPU overload. This is not surprising, as this configuration has the highest work load, since a single machine must manage the server task and the NPC controllers in the same process, including their socket read and write activities. It is interesting to compare these numbers with the ones obtained by the Local Engine Separate NPC engine. The comparison reveals that half of the CPU power and an increasing amount of memory is spent on running the NPCs.

If we compare CPU utilization between the different configurations (see figure 2), we notice that Fake Engine, Distributed Engine and Local Engine Separate NPC have similar CPU load. This is not surprising for Distributed Engine and Local Engine Seperate NPC, as their workload is similar (no NPCs, but using socket communication). However, the similarity with Fake is unexpected, as there is no correlation between the workload generated by NPC using direct memory access to communicate and the workload of the server which communicates using sockets. The low CPU load of the No Engine configuration is not surprising, since elements that create load in the system, such as interest management, replication, remote method invocation and socket communication are not present.

The memory allocation needs of the various configurations (see figure 3) also reveals some interesting patterns. Local Engine and Fake Engine have memory requirements that grow linearly as more clients are added to the system. This is easily explained by the memory required by the NPC controllers, which are stored in the same process as the server component. Local Engine has an even higher memory requirement, as it must provide memory to the network sockets, which are not found in the Fake Engine. Even though the No Engine configuration also stores its NPC controllers in the same process, no replication engine is used, and hence all NPCs can share the same game objects.

The memory requirements of the Local Engine Seperate NPC and Distributed Engine are interesting, as they both stabilize once 25 clients have connected to the system. This can be explained by the fact that although additional clients generate additional CPU load in the server, once the initial data structures (hash tables, etc.) that help to keep track of replicas and interest management have been allocated, connecting additional players requires only very little memory.

The bandwidth curves (see figure 4) are clearly exponential. Given the "complete view" interest management algorithm used in our experiments this is not surprising. The n players generate 1 remote method invocation message every second, which results in n^2 updates that have to be propagated to all player nodes per second. Throughput for the Local Engine is much higher, as it includes the traffic generated by both the NPCs and the server component. Since monitoring is done at the network interface, it is impossible to differentiate traffic from the NPC and server component. The bandwidth requirements for the Local Engine Separate NPC setup was identical to the Local Engine one, and therefore omitted.

5. DISCUSSION

Typically, a simulation is designed to reproduce "real-world" conditions as accurately as possible. In the context of MMOGs, the parameters that are to be considered include the number of simulated players and the algorithms that control the behaviour of the players. The experimental results presented in subsection 4.3 demonstrate that the simulated performance of a server component also varies depending on the type of communication used, the location of the NPC controllers, and whether they execute in the same process or even in the same thread.

For example, the No Engine scenario is the easiest configuration to set up, as it includes no communication and sharing of data between threads. Its results demonstrate incredible scalability, as the CPU load and the allocated

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8632
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8632

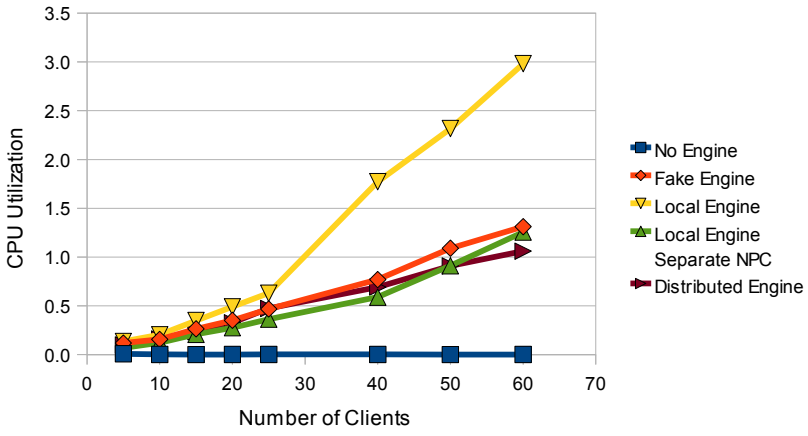

Figure 2: CPU Utilization

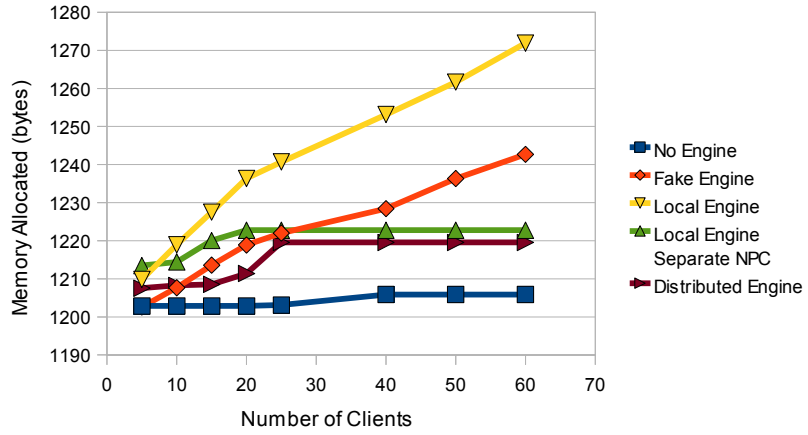

Figure 3: Memory Utilization

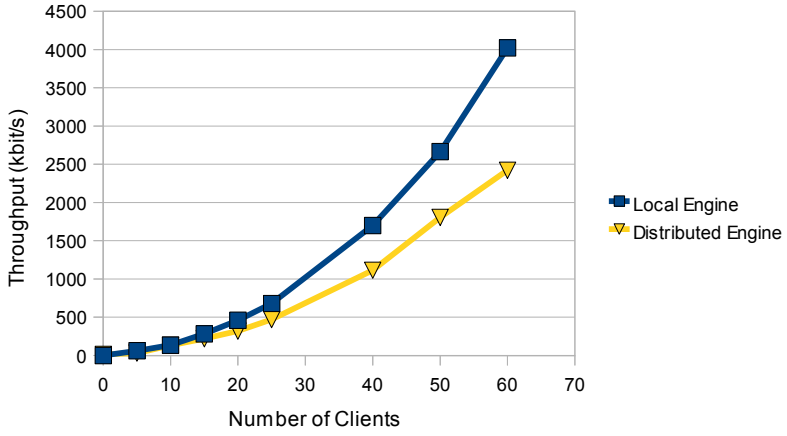

Figure 4: Bandwidth Utilization

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8632
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8632

memory do not increase significantly as clients are added to the experiment. However, this simulation does not provide accurate performance information, as all CPU intensive operations (interest management, serialization, network I/O) are omitted.

Comparing Fake Engine and Local Engine provides a better understanding of the load generated by using socket-based communication. Both setup configurations are identical, except that Fake communicates using local memory while Local Engine uses network sockets connected to *localhost*. The difference between both curves indicates the non negligible load created by the sockets. A careful inspection using a profiler reveals that in addition to spending time reading from and writing to sockets, a non-negligible time is also spent on queueing data until it is ready to be transmitted. This transmission overhead is often overlooked in simulators.

The difference between the numbers obtained from the Local Engine and Local Engine Separate NPC experiment are interesting, since the difference between both curves provides insight into the load generated by the NPC controllers. Again, surprisingly, this load is considerable. We conclude from this that in order to run accurate performance benchmarks in MMOGs, the control for simulated agents should always be located in a separate process.

One interesting and encouraging statistic is the similarity between the Local Engine Separate NPC and Distributed Engine measurements. This similarity suggests that given the proper conditions, it is possible to create a simulation on a single machine that provides performance measurements that closely reflect those obtained in a real distributed setting. Of course, if measurements are to be taken that span execution paths across several nodes, eg from an NPC controller node to the server and back again, then latency effects encountered in a real distributed setting can not be ignored. Hence, Local Engine Separate NPC can only be used if a network latency simulator is inserted between the node running the NPC agents and the node running the server. However, the scalability of the Local Engine Separate NPC simulation is still limited by the load generated by the NPC controllers in the second process, since it runs on the same machine[2].

6. CONCLUSION

This paper critically reflects on the use of simulated experiments in the context of MMOGs, in particular when the experiments are aimed at evaluating game performance and server scalability. Five simulation setups were defined within the Mammoth MMOG framework, 4 using a single computer and 1 in a distributed setting. Each setup ran the simulated players in a different way (using a single thread, separate threads or separate processes) and used different ways of communication between the players and the server.

An analysis of the gathered measurements – CPU load, memory and bandwidth usage – showed non-negligible performance variations, demonstrating the important influence of the simulation setup on the performance results. We conclude that performance simulations for MMOGs must be designed very carefully. It could even be argued that there is no substitute to testing a MMOG in live conditions, using a large number of computers. We believe that running at least one experiment in a real distributed environment is probably unavoidable in order to discover and/or understand the performance costs associated with the used communication technology irrespective of the lag introduced by the network. Only then can a centralized simulation – one that is more convenient to setup and run – produce measurements that can be interpreted correctly.

Some performance changing factors are easily overlooked when designing a simulation. For instance, as shown in our experiments, the way the players are simulated – using a single thread, using separate threads or using separate processes – has a considerable impact on the measurements. Likewise, the overhead of socket communication, such as the flattening of data structures and the queuing performed by the operating system, affects the performance and memory usage significantly. If these factors are not taken into account, the accuracy of the measurements gathered during the simulation is difficult to judge.

Of course, simulations can easily be used to measure other experimental data. For instance, the number of messages sent among the players and the server was identical in all our simulation setups, provided that the simulation was run with a network engine.

[2]A different set of experiments that is not reported here in detail has shown that the Distributed Engine can support over 100 clients, whereas Local Engine Separate NPC caps out at 75.

Acknowledgment

This work has been partially supported by NSERC, the National Sciences and Engineering Research Council of Canada.

7. REFERENCES

[1] Freepastry, December 2009. `http://www.freepastry.org/`.

[2] R. Balan, M. Ebling, P. Castro, and A. Misra. Matrix: Adaptive middleware for distributed multiplayer games. *Lecture notes in computer science*, 3790:390, 2005.

[3] A. Bharambe, J. Pang, and S. Seshan. Colyseus: A distributed architecture for online multiplayer games. 2006.

[4] A. Bharambe, S. Rao, and S. Seshan. Mercury: a scalable publish-subscribe system for internet games. In *Proceedings of the 1st workshop on Network and system support for games*, pages 3–9. ACM New York, NY, USA, 2002.

[5] J. Boulanger, J. Kienzle, and C. Verbrugge. Comparing interest management algorithms for massively multiplayer games. In *Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*. ACM New York, NY, USA, 2006.

[6] W. Cai, P. Xavier, S. Turner, and B. Lee. A scalable architecture for supporting interactive games on the internet. In *Proceedings of the sixteenth workshop on Parallel and distributed simulation*, pages 60–67. IEEE Computer Society Washington, DC, USA, 2002.

[7] L. Chan, J. Yong, J. Bai, B. Leong, and R. Tan. Hydra: a massively-multiplayer peer-to-peer architecture for the game developer. In *Proceedings of the 6th ACM SIGCOMM workshop on Network and system support for games*, pages 37–42. ACM New York, NY, USA, 2007.

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8632
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8632

[8] J. Chen, B. Wu, M. Delap, B. Knutsson, H. Lu, and C. Amza. Locality aware dynamic load management for massively multiplayer games. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 289–300. ACM New York, NY, USA, 2005.

[9] A. Denault, J. Kienzle, C. Dionne, and C. Verbrugge. Object-oriented Network Middleware for Massively Multiplayer Online Games. Technical Report SOCS-TR-2008.5, McGill University, Montreal, Canada, December 2008.

[10] C. Diot and L. Gautier. A distributed architecture for multiplayer interactive applicationson the Internet. *IEEE network*, 13(4):6–15, 1999.

[11] L. Fan, H. Taylor, and P. Trinder. Mediator: a design framework for P2P MMOGs. In *Proceedings of the 6th ACM SIGCOMM workshop on Network and system support for games*, pages 43–48. ACM New York, NY, USA, 2007.

[12] D. Fu and R. T. Houlette. Putting AI in entertainment: An AI authoring tool for simulation and games. *IEEE Intelligent Systems*, 17(4):81–84, 2002.

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: Elements of reusable object-oriented design, 1995.

[14] T. Iimura, H. Hazeyama, and Y. Kadobayashi. Zoned federation of game servers: a peer-to-peer approach to scalable multi-player online games. In *Proceedings of 3rd ACM SIGCOMM workshop on Network and system support for games*, pages 116–120. ACM New York, NY, USA, 2004.

[15] J. Kienzle, A. Denault, and H. Vangheluwe. Model-based Design of Computer-Controlled Game Character Behavior. In *10th International Conference on Model Driven Engineering Languages and Systems - MoDELS 2007, Nashville, TN, USA, Oct. 1-5, 2007*, number 4735, pages 650 – 665, October 2007.

[16] B. Knutsson, H. Lu, W. Xu, and B. Hopkins. Peer-to-peer support for massively multiplayer games. In *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, volume 1, 2004.

[17] H.-H. Lee and C.-H. Sun. Load-balancing for peer-to-peer networked virtual environment. In *NetGames '06: Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*, page 14, New York, NY, USA, 2006. ACM.

[18] M. Matskin. Scalable agent-based simulation of players in massively multiplayer online games. In *Eighth Scandinavian Conference on Artificial Intelligence: SCAI'03*, page 153. IOS Press, 2003.

[19] C. Onuczko, M. Cutumisu, D. Szafron, J. Schaeffer, M. McNaughton, T. Roy, K. Waugh, M. Carbonaro, and J. Siegel. A Pattern Catalog For Computer Role Playing Games. In *Game-On-NA 2005 - 1st International North American Conference on Intelligent Games and Simulation*, pages 33 – 38. Eurosis, August 2005.

[20] OPNET. ACE Analyst and ACE Live. http://www.opnet.com/, 2010.

[21] A. Raja and M. Katchabaw. Using Synthetic Players to Generate Workloads for Networked Multiplayer Games. In *3rd International North American Conference on Intelligent Games and Simulation*, 2007.

[22] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, pages 329–350, 2001.

[23] Unreal Technology. The Unreal Engine 3. http://www.unrealtechnology.com/html/technology/ue30.shtml, 2007.

[24] Z. Zhou, H. Wang, J. Zhou, L. Tang, K. Li, W. Zheng, and M. Fang. Pigeon: a framework for testing peer-to-peer massively multiplayer online games over heterogeneous network. In *3rd IEEE Consumer Communications and Networking Conference, 2006. CCNC 2006*, volume 2, 2006.

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8632
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8632