MiMaze, a Multiuser Game on the Internet

Laurent Gautier, Christophe Diot

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

MiMaze, a Multiuser Game on the Internet

Laurent Gautier and Christophe Diot

N° 3248

Septembre 1997

THEME 1

Réseaux et Systèmes

ISSN 0249-6399

MiMaze, a Multiuser Game on the Internet

Laurent Gautier and Christophe Diot*

Theme 1 : Réseaux et Systèmes

Projet RODEO

Rapport de recherche n°3248- Septembre 1997

30 pages

Abstract: This report describes the design, the implementation, and the evaluation of MiMaze, a totally distributed game on the Internet. This work focusses on the mechanisms that have to be used to make it possible to play the game on the Internet. Our major contribution is to have designed MiMaze with a distributed architecture, without any server. MiMaze consequently uses a multicast communication system based on RTP/UDP/IP to which have been added mechanisms to guaranty the consistency of the game, whatever the network delay is. MiMaze has been designed with the DIS rules. This report provides a detailed evaluation of the game, as well as elements of monitoring to compare the "efficiency" of such applications. Missing mechanisms and future works are discussed at the end of the report.

Key-words: Distributed Architecture, Interactive Applications, Synchronization, Experimental System, Group Communication, Communication Control, Internet.

(Résumé : tsvp)

Version provisoire du 5 Septembre 1997.

* Email: [laurent.gautier | christophe.diot]@sophia.inria.fr

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE
Unité de recherche INRIA Sophia Antipolis : 2004, route de Lucioles - B.P. 93 - 06902 Sophia Antipolis cedex **(France)**
Téléphone : +33 4 93 65 77 77 - Télécopie : (33) 4 93 65 77 65 - Télex : 970 050 F

MiMaze,
un jeu multi-utilisateurs
sur Internet

Résumé : Ce rapport décrit la conception, l'implémentation, et l'évaluation de MiMaze, un jeu totalement distribué sur Internet. Ce travail cible plus particulièrement l'aspect communication de l'architecture du jeu. Il propose des mécanismes pour permettre à chaque joueur de participer quelque soit l'endroit où il se situe sur le réseau. La contribution majeure de ce travail est d'avoir défini les bases d'une application distribuée sans serveur, avec les mécanismes à mettre en oeuvre pour garantir la consistence du jeu quelque soit le délai réseau de chaque participant. MiMaze utilise par conséquent un support de communication qui s'appuie sur RTP/UDP/IP auquel ont été additionnés des mécanismes qui garantissent la consistence du jeu. MiMaze a été conçu en suivant les règles définies par la norme DIS. Ce rapport fournit une étude détaillée des performances du jeu, ainsi que des éléments qui permettent de comparer ces performances. Les mécanismes manquants et les travaux futurs sont discutés à la fin du rapport.

Mots-clé : Architectures distribuées, Applications interactives, synchronisation, systèmes expérimentaux, Communication de groupe, Contrôle de communication, Internet.

1.0 Introduction

This article describes the design and the evaluation of a multiuser (distributed) game over the Internet. MiMaze is a totally distributed game (no server) that uses an unreliable communication system (MiMaze transmission control is based on RTP [9] over UDP/IP multicast). A gaming application has been chosen because it represents a new type of application for networks that is representative of the new generation of interactive multimedia applications, including Distributed Interactive Simulations (DIS), digital battle fields and Air Traffic Control (ATC)[3][4].

Because of the distributed nature of this game, we had to design a synchronization mechanism to cope with different transmission delays among the participants. This synchronization mechanism (called bucket synchronization) will be analyzed later in this document. It is the minimum functionality required to play MiMaze on the Internet. But it is not sufficient to render the real-time properties of the game with network delays and large numbers of participants. Consequently, this article also proposes an preliminary analysis of new functionalities that have to be added to the application to make it “real” despite the networking environment.

For this work, we have chosen to follow the DIS [1][2] rules on transmission characteristics. In particular, any action issued in the game must reach (and be displayed) any participant before 100 ms. The number of participants can be very high (let say more than 1000). For this reason, part of the analysis (on group management) will be done by modeling and simulation using a group level simulation platform (that is under design).

Before we move to the technical content of this paper, we define the notion of an "object". The game is made of objects. E.g. any mobile is an object; the terrain is an object. When two objects have a common behavior (like a common trajectory, or belong to the terrain), they are considered as a single object. For example, a vehicle is an object; if this vehicle losses a wheel, the wheel becomes immediately a separate object. With the same definition, smoke or fog are independent objects.

This article is structured as follows. Section 2 describes MiMaze. We start with a description of the game and of its functional architecture. Then we give a detailed description of the bucket synchronization algorithm. In section 3, we describe and analyze early performance measurements realized on the Internet. The performance results gives us an opportunity to discuss, in section 4, what enhancements are necessary, to be make MiMaze more real-time, and more robust. We conclude with the next MiMaze design step, and with the extension of the game to a more realistic DIS environment.

2.0 Description of MiMaze

The characteristics of distributed games are very similar to those of DIS applications [1][2]. As explained earlier, the simplicity of the game does not influence the

characteristics of the communication architecture. Instead, choosing a more complex game would have made it difficult to analyze the game traffic. The characteristics of a DIS application (including games) are the following:

- Interaction delay: any action issued by any participant must reach other participants before 100 ms. This is of course independent of the transmission delay between participants (e.g. in some cases, information has to be delivered to the destination within 100 ms, when the network delay experienced is more than 100ms).
- Large number of participants. The targeted number of participants is 10.000. That means existing group management policies are not applicable. For MiMaze, we have decided to limit the number of participants in order to analyze scalability problems to smaller groups (let say 2 to 50 participants).
- Interactive data (actions) are short (few bytes) and frequent (but not periodic). They differ from other multimedia data (audio and video). Each game object can also transmit his local state which, in the worst case (many articulated parts, complex objects), represent a small amount of information (less than few hundred bytes).
- High level of dynamicity in group structure and topology. This can be amplified by the participation of mobile units.
- Information is "continuous". In most of the cases, action *X* at time *n+1* "contains" action *X* at time *n* (for example the displacement of a participant, or the trajectory of a bullet). The consequence is that the loss of action *Xn* has a limited impact on the transmission. A second consequence is that information *Xn* becomes obsolete as soon as a more recent value of action *X* (*Xn*+1) is received. This property proves that only a small part of the application data needs a reliable transmission.
- Video and audio type information might have to be transferred. In that case, they would have to be synchronized on the participant actions or displacement. In MiMaze, only non-synchronized audio is provided among participants (using FreePhone [5]).
- Adaptation techniques can be applied. Data encoding has been designed in such a way to enable hierarchical transmission of the game information.

2.1 MiMaze design characteristics

MiMaze is an evolution of the iMaze game [8]. iMaze is a bi-dimensional Pacman game with a 3D representation (see screen shot figure 1). Participants (Pacman) evolve in a labyrinth where they try to kill each other. Part of the labyrinth wall can be passed in one direction. Each participant has a 3D representation of his vision domain and a 2D view of the game (from the top).

2.1.1 MiMaze distributed architecture

To design MiMaze (which architecture is shown figure 3) we have started from a centralized unicast version of the game (iMaze, figure 2). MiMaze is to our knowledge the only game with *a fully distributed architecture using multipoint communication support.*

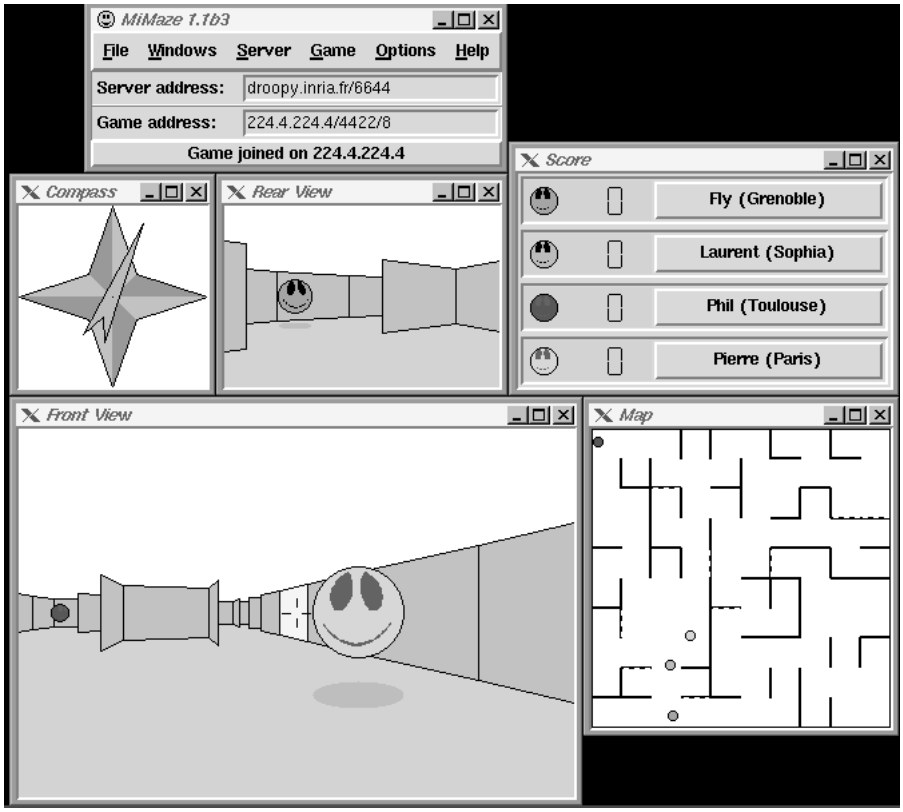

Figure 1. MiMaze screen shot

Fully distributed architectures have many advantages compared to server based architectures:

- Robustness. In a distributed architecture, the failure of any of the participants has no effect on other participants. State information are not lost and sessions can continue almost normally.
- Scalability. We have identified two different factors that limit the scalability of centralized architectures:

- All the data converge to the server. The server is consequently a natural bottleneck. Once its maximum CPU load is reached, the game state computation frequency will slow down in order to continue to serve all participants, and to collect all participants actions.

- In a server architecture, the amount of data transferred on the network is bigger than with a distributed architecture, where the game state is never in transit on the network.

There is consequently no doubt that distributed architectures are more scalable than centralized architectures. But the second limiting factor also applies to distributed architectures; it is only differed (it applies for much larger groups)

- Minimum delays. In a centralized architecture, the interval of time before information can be delivered is double that of a distributed one. When an action leaves a participant location, it has to reach first the server where a game state is computed; then, the state is forwarded to all participants to be displayed. In the distributed architecture, any action has to cross the network only once to reach its destinations, where the global state will be computed and displayed.

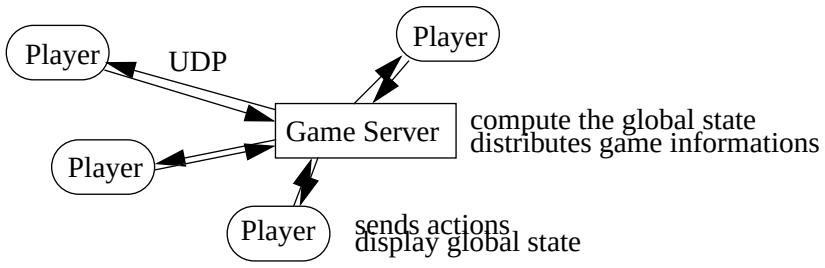

Figure 2. iMaze architecture (with server)

We are conscious of the fact that the centralized architectures have other advantages. One of them is that all the participants in the game have the same “image” of the game. Moreover, the server introduces a natural synchronization of the players that all display the same game at the same time (at the price of interactivity). We are also aware that the main advantage of centralized architectures is that it allows game companies to avoid piracy and make money.

But the main advantage of centralized architectures is that the presence of a server makes cheating difficult. In a totally distributed architecture, each entity is taking its own decisions, and there is no authority to identify potential cheaters. On the other hand, in a centralized architecture, all the information transit in the server that can authenticate the exactness of the global state. Consequently, the deployment of dis-

tributed architectures will require the use of specific distributed mechanisms to deal with the honesty of participants (see section 4.5 for more details)[12].

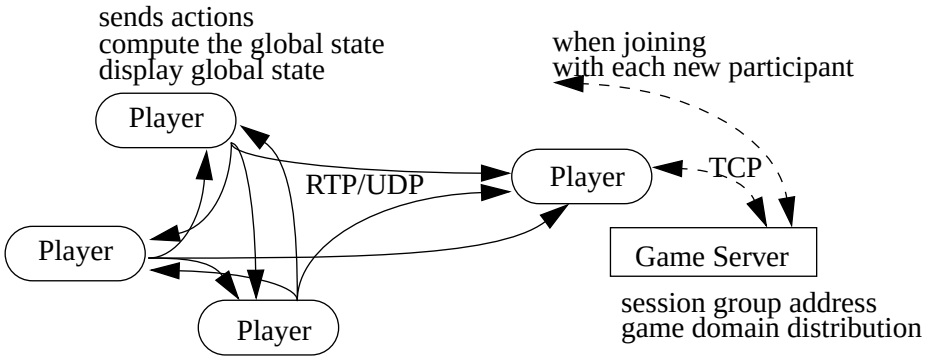

Figure 3. MiMaze architecture

2.1.2 Object characteristics

The DIS standard defines numerous types of information to be packetized and managed by the application. The DIS standard is an application level standard designed by DIS applications experts. It is not optimized for network transmission. Even worse, it is unrealistic to use this protocol over Internet as the DIS relies on low loss, low delay and high bandwidth networks [3][4].

Most of the DIS packets (called PDU for Protocol Data Unit) apply to distributed games. There are more than 25 PDUs. The most frequently used is the ES PDU (Entity State PDU) which carry state information describing the game objects. Then, a specific PDU type has been defined for each exception such as collisions, fire, detonation, and also for logistics, and control. Please refer to the official standard for details on these PDUs.

In MiMaze, packets (called ADU for Application data Unit) are slightly different from those defined by the DIS standard. For network optimization reasons, we have kept only one packet type (called the State ADU) which format is described figure 4. A State ADU is made of two parts: the header, and the payload. To make it more flexible for network transmission, and to make the application adaptive, each information field in MiMaze ADUs is encoded as a descriptor in the packet payload. Descriptor structure is [type of information / size of information / information].

The State ADU is equivalent to the DIS ES PDU. An object uses State ADUs to periodically advertise its local state. Exception information such as shooting and collisions are sent as part of the State ADU (it is not a problem with shots as bullets remain part of the Pacman of its origin, see below). State ADUs are transmitted with an unreliable protocol; the frequency of the transmission provides a natural redundancy.

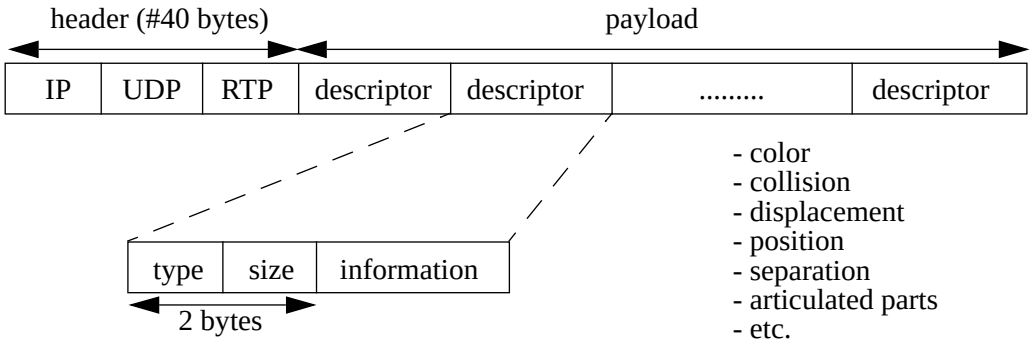

Figure 4. MiMaze ADU format

The main difference with DIS is that in MiMaze, the bullets remain part of the Pac-Man that shot them. This characteristic comes from the iMaze game; modifying it would have required the complete re-design of the game. But we consider this as an important issue for the next release of MiMaze.

2.2 The Bucket Synchronization Mechanism

Synchronization has been introduced to allow actions issued at the “same time” to be processed together by each participant (see Figure 5).

The synchronization mechanism makes sure that actions processed to evaluate the global state of the game at time *t* were all issued “close to” *t*, or in the same evaluation period. For this purpose, a bucket is associated to each evaluation period. This bucket receives data that will be used by the game to compute the game state when it will be time to compute this bucket.

For example, in figure 5, without synchronization, the information issued at t_3 would have been processed together with the one issued at t_1 at another location (but received at t_2, which is in the same processing interval than t_3). The synchronization allows action received at t_0 to be delayed in the bucket *d* (processed at t_d) in order to be synchronized with the action issued at t_1 by another participant.

Another aspect of synchronization is that each participant should display the same game at the same time. As the display frequency is high enough (equal or above 25 images per second), no specific mechanism is needed. But such mechanism could be needed for application having sharper time constraints than MiMaze (or application unable to process buckets frequently enough).

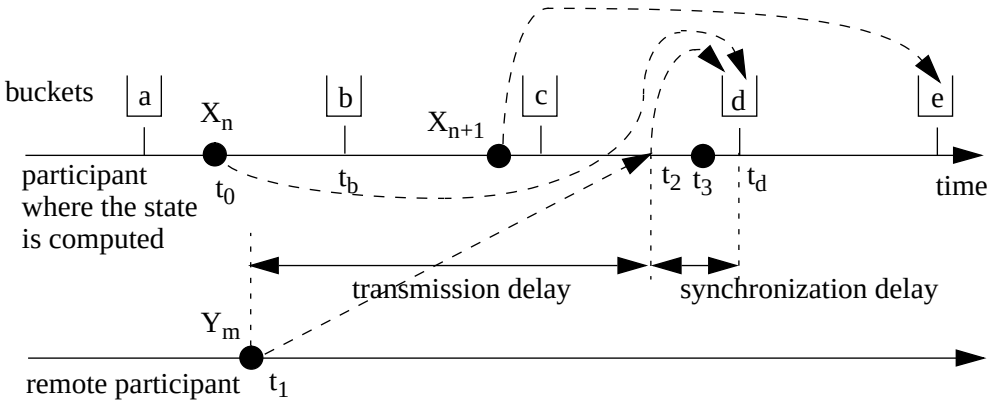

Figure 5. The bucket synchronization mechanism

2.2.1 Synchronization delay

The synchronization delay is computed by a receiver to determine in which bucket the incoming information should be stored. The sum of the synchronization delay with the network delay should be less than 100 ms, even if the network delay is more than 100ms (in case a network delay is larger than 100ms, a mechanism called Dead-Reckoning- described section 4.1- must be used to re-synchronize efficiently the participants). So, the bucket mechanism is very close to a playout buffer mechanism [10] used to reduce network jitter effects on audio information.

2.2.2 Bucket frequency

The bucket frequency defines the period with which a new game state is computed. The evaluation period (or interval between two buckets processing) has to be defined carefully. The human vision is continuous for a frequency of 25 images per second. Consequently, bucket frequency should be at least 25 times per second, or every 40 ms. From the previous discussion on inter participants synchronization, 25 times per second signifies that the same global state of the game will be seen by all participants within a period of 40 ms. Performance evaluation will tell us if it is acceptable or not.

2.2.3 Global clock mechanism

The bucket synchronization mechanism uses a global clock mechanism to evaluate the delay with which an information reached him. The clock has to be accurate (10ms precision) and continuous. In our implementation, we use NTP [6]. In case NTP is not available, we use a NTP-like algorithm based on the evaluation of the round trip time [6]. There are two problems with NTP:

- First, all implementations of NTP are not homogeneous. There are 3 levels of NTP servers and it is very difficult to maintain good synchronization among participants when level 3 servers are involved. Lower stratum mechanisms (e.g. ntpdate) are not sufficient.
- Second, NTP encodes clock information on 64 bits, when RTP uses 32 bit clock. This was a problem because MiMaze manipulates both clock informations.

At this time, in order to increase the precision of NTP under stratum 2, we use both NTP and our NTP-like mechanisms to compute clock offsets.

3.0 Performance evaluation

This section is organized in five subparts. We start with the description of the experimental setting, and the monitoring tool designed specifically to re-synchronize the distributed logs. We then define what are the distributed game metrics we would like to analyze. The evaluation part begins with an informal analysis and continues with the analysis of the behavior of the network during the experiment (studying delay distribution and losses). We conclude with the analysis of the game parameters.

3.1 Experimental environment

We have performed a performance evaluation on the Mbone [11] with up to 25 participants located on the French territory. The architecture of the experiment multicast tree is given figure 6.

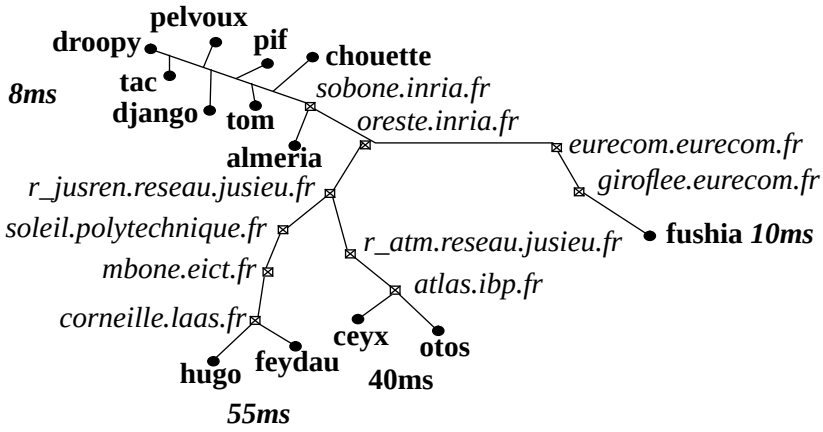

Figure 6. Mbone architecture during the evaluation

Delays given are average measured end-to-end from droopy. Computers connected where heterogeneous including SUN (SPARC 10, 20, ULTRA), DEC Alpha, PCs).

The reference computers we are using in our benchmarks are droopy (number 23 on benchmarks), tom (number 10), and hugo (number 43).

3.2 Monitoring

The problem with monitoring is that each participants collects dump files in which all the game information are collected. A dump is composed of:

- Application level data: time-stamped states of local and remote participants, time-stamped actions of the local participant, time-stamped collisions information. Those data allow us to reconstruct the local view of the game for a participant.
- Synchronization data: round trip times and clock offsets between local and remote participants. Those data are used for the dumps synchronization described below, and for game synchronization analyzes.
- Network level data: for each received packet we collect the senders' identity, the emission time-stamp, the reception time-stamp, and the sequence number.
- Local resource data: CPU load for several computations. The CPU load has never been a bottleneck yet but the increasing number of participants can produce side effects we need to analyze.

The main difficulty is not in benchmark computation, but in the dumps re-synchronization. As a result of the system being distributed, there is no absolute clock in the game session, and each participant computes offset for any other participant in the session. To re-synchronize logs, we apply the least square algorithm to the clock offset measured to compute an approximation of the real offset. Then, this offset approximation is used as a reference value to resynchronized dumps with the measured (or collected) offsets.

This re-synchronization method is only valid when measured offsets are not too far from the reference offset. Consequently, when the NTP synchronization was very loose, we were not able to resynchronize dumps. That explains that the following benchmarks have been obtained on limited time intervals where the clock synchronization was efficient.

Once the dumps are synchronized, we could analyze most of the games parameters, and in particular we could compute a reference state of the game that could be compared to the local state of each participant.

3.3 Distributed game metrics

We had to define objective criteria for multiuser games evaluation. These criteria have to visualize how far the distributed game behavior is from the behavior it would have if all the participants would have no delay with their game representation*. There are three important functions to evaluate the game behavior:

* the representation of a participant on a game is called avatar in the virtual game terminology.

- Scalability *S*. The algorithms designed must be scalable. The number of participants can be high, as well as the dynamic. Scalability is an important parameter from a group management viewpoint; and algorithms which have costs that are directly dependent on the number of participants should be avoided. In certain cases, two different algorithms might have to be used for the same function depending on the size of the group.
- Interaction *I*. The control algorithms provided with a distributed game must hide the network delay that separate a participant from other participants. Maintaining interaction among participants means that the view of the game at each location is not dependent on the network delay.

  A solution to maintain this interaction, in case of large delays, could be to slow down the game speed.
- Consistency *C* of the game. The previous parameter has a direct influence on the game consistency. In a distributed application, each user has his own view of the application's global state. This view is consistent if it is the same for each participant.

The parameters that influence the previous functions are of two different kinds: network parameters and game parameters.

3.3.1 Network parameters

Network parameters that will help us to understand the behavior of our application are primary:

- Loss (*L*). L is a function of the number of participants *N* and the network.
- Network delay (*D*). *D* is the delay between any couple of participants (measured application to application, not user to user). The most important value is *Dmax*, which is the same for all participants. The synchronization delay D_s is a direct function of *Dmax*. *D* is influenced by the group topology and by the multicast routing algorithm.

3.3.2 Game parameters

To be able to analyze the previously defined functions, we need to express them using game parameters. We have identified the following game parameters:

- Game state *St* is the view of the game by a participant. If the game works properly, each participant should see the same game state at each instant. This state would be called the global state. We consequently define the reference state as the state obtained with all participants playing at their avatar location (no delay between the participant and his avatar).
- Number of participants (*N*). *N* is a dynamic parameter that we will use to evaluate the scalability of the game (or $S = f(N, C)$).

- Frequency with which each participant sends its actions (*Fa*). It is defined by each participant (eventually) independently, depending on its local performance.
- Bucket frequency *Fb.* It is the frequency with which the global state of the game is computed.
- Synchronization delay D_s. As defined earlier, the synchronization delay is the time that should be added to a received action to delay it to the appropriate bucket. D_s is a function of *D* and *Dmax*. As explained earlier, $D_s + D < 100$ *ms.*
- Game speed (*Sp*). *Sp* is the frequency with which the game is displayed. The speed of the game might have to be slowed down (locally) to maintain a consistent view of the game to all participants. The usual game speed is 25 times per second (and is correlated to the bucket frequency). It can be reduced to 16 times per second with "tolerable" effect on the game (Japanese cartoon quality).

3.4 Performance analysis

3.4.1 Informal analysis

As it was our first experience with a distributed game over the Internet, it is useful to describe how players "perceived" the game*:

- We found that neither the network delay (note that network delays were less than 100 ms) nor the number of participants had a negative impact on the "quality" of the game. Previous experiments with the original iMaze game were limited to less than 10 participants. The number of participants reached 25 during the experiment with no more trouble other than with 2 or 3. The "speed" of the game also did not suffer as a result of network delay and all participants had a comparable interactivity. The displacement of Pacman in the labyrinth was smooth and with a regular speed at any participant.
- The behavior of each participant is independent. Due to the nature of the game, and to the multiple participants (having each a different computer and network connection), few unexpected behavior occurred during the evaluation session, such as participant disconnection (CPU load, network failure, dump memory saturation, etc.) or synchronization loss (NTP re-synchronization or failure). Such situation had strictly no effect on the other participants because of the totally distributed architecture of the game. The game is designed to remove immediately a participant from the display when a failure is detected). Other players can consequently continue to play without inconvenience.

* It is important to remind here that the game architecture is totally distributed (no server), that it uses multicast (RTP/UDP/IP), and that transmission is totally unreliable.

- The game is very sensitive to the host computer activity. MiMaze is installed in the user space of the computer. It is consequently often de-scheduled to run higher priority process. If the waiting time is too long (more than 200 ms), the synchronization is lost, as well as game information (the synchronization algorithms discards packets that cannot be re-synchronized). We had this problem on Almeria (SPARC 10) where a kernel process was running every minute for one second, thus de-synchronizing the local player with apparently no real visible effect for the player.

- The bucket algorithm was efficient, except in the case of potential collision, e.g. when two participants were “too” close to each other (we will evaluate in further experiments what “too” close means). The visible effect of a collision situation to the session participant was that his pacman was unable to kill the other “too close” pacman. Because the distance between two Pacmen is too short, the game control was totally unable to deliver actions to these two participants “on-time”. These actions were consequently discarded.

- Each participant had different scoring information. The score being computed locally, it cannot be incremented when the participant is not-synchronized.

3.4.2 Network parameters

Delay distribution and losses are the first parameters we investigated. The reason is these parameters are network dependent, and unavoidable.

Delay distribution and clock

To evaluate relative delays (with each participant), each computer uses a global clock signal. To maintain a synchronization between these clocks, we use NTP.

- MiMaze uses the starting time and the arrival time of S ADUs to compute the delay. Consequently, it maintains a clock offset information for each other participant.

- When the delay measured is variable, the clock offset is interpolated to minimize synchronization loss. Interpolation is used until new "stable" values can be computed from the NTP information.

Hence, the accuracy of the delays used by MiMaze (most of the synchronization efficiency of the synchronization algorithm lays on the accuracy of the delay) is strongly correlated to the accuracy of the clock. Figure 7 gives a relative view of the participant clocks during 1400 second of the experiment. For this experiment we were using NTP server level 3 on most of the computers, and NTP servers level 2 on some of them. The 0 position on the vertical axis is the reference clock of the computer from which we made this measurement (Droopy is a SPARC Ultra I creator at 167 MHz).

As long as the relative value of the clocks is the same, MiMaze has no synchronization problems. This is the case between 1000 and 1400 seconds. Around 700 s, a NTP re-synchronization occurs on the reference computer (close to 3 seconds). The consequences on MiMaze are that the delays computed by the bucket synchronization algorithm are wrong and corresponding packets can not be resynchronized. Another interesting observation is the drift speed of the clocks. While around 600 seconds the clocks remain parallel, They drift very quickly after 1400 seconds. This creates an imprecision on the evaluation of the delays in MiMaze.

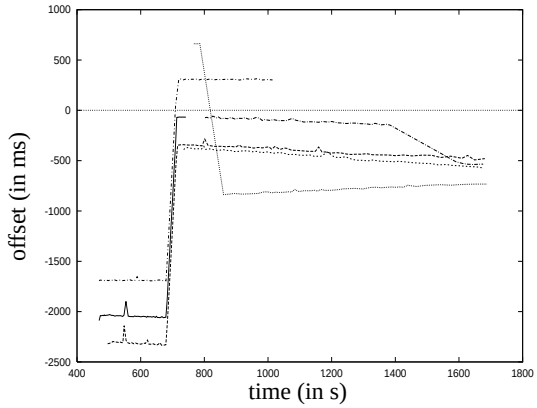

Figure 7. Clock synchronization using NTP (level 2 and 3)

Figure 8 now shows the delay measured locally (between two computers on the same Ethernet) and on the MBone during the experiment. We have chosen to present the distribution curves only as the delay vs. time figure would be less easy to understand and to analyze.

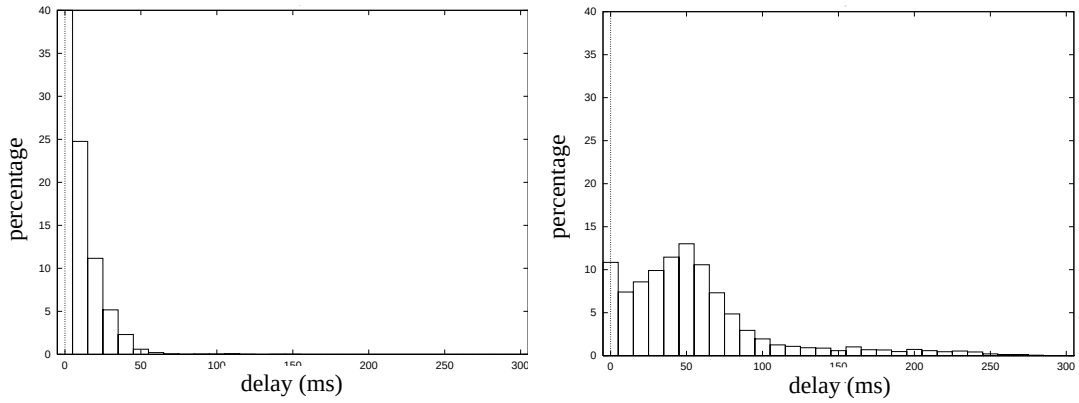

Figure 8. Network delay distribution observed by MiMaze locally (left) and through the Mbone (right)

We observe in figure 8 that the average delay computed by MiMaze is very close to the average delay observed on the Mbone during the experiment. Locally, the delay was 8ms; on the Mbone to the most distant participant (Hugo, 43) was at 55 ms. The delay distribution is also very similar in both cases. For local transmission, 40% of the delay information was centered on the average network delay. For Mbone transmission, only 15% are centered on the average delay. But a 10ms imprecision is more acceptable on a 55 ms delay than on a 8ms delay. In the case of local participants, approximately 40% of the delays are 10% accurate (the granularity of figure 8 does not allow us to look at it in more details); on the Mbone, it is also approximately 40% of the delay that are within a 10% range of accuracy.

The calculation of the mean (μ) and of the standard deviation (σ) gives more accurate information on the measure. For figure 8 left, σ=ms (with μ=8.73ms), when for figure 8 right, σ is 50.44ms (with μ=55.47ms). The mean calculated is very close to the average delay measured during the experiments. The interesting result is that the standard deviation is not very different in both cases. But it is too large. We hope that, using NTP level 1 or 2 servers synchronized by GPS (or any other global clock) will reduce consequently this standard deviation [7].

Such a reduction on the delay standard deviation will have a direct influence on the coherency of the game, and on the average number of un-synchronized actions. To illustrate this, let take a mean delay of 50ms and a standard deviation of 50ms. In that case, approximately 70% of the delays will be less than 100 ms, and 15% of the delays should be higher than 100ms (and consequently un-synchronized). For a mean delay equal to 100ms, that is 50% of the actions that will be discarded because they are too late.

So, it is essential to add to MiMaze both a better global clock mechanism and dead-reckoning to maximize the efficiency of the game for any range of delay.

Losses

We only analyze here losses due to the network. Data that are discarded because they cannot be re-synchronized can also be considered as lost. But we will analyze it in the following section.

Figures 9 left and 10 left show the percentage of action lost* on a 200s period of time (locally and on the Mbone respectively). Figures 9 right and 10 right give the distribution of loss for the same period locally and on the Mbone respectively). The computation of the mean loss rate gives 2% locally and 7% on the Mbone (nationwide). A comparison of the participant to participant loss rate shows that the 2%

* The maximum should be 100%. Due to delay evaluation inaccuracy, it sometimes goes up to 140%, which can be explained by more than one action received from a participant in the same bucket. This phenomenon has no effect on the game as MiMaze keep only the most recent information to compute the global state.

losses observed locally are in fact ADUs coming through the Mbone (from other external participants).

If the average loss rate is much higher on the Mbone than on a local Ethernet (this result is not surprising), the distribution of loss is approximately the same locally and over the Mbone: figures 9 and 10 right show that approximately there is no loss in 80% of the time in both cases.

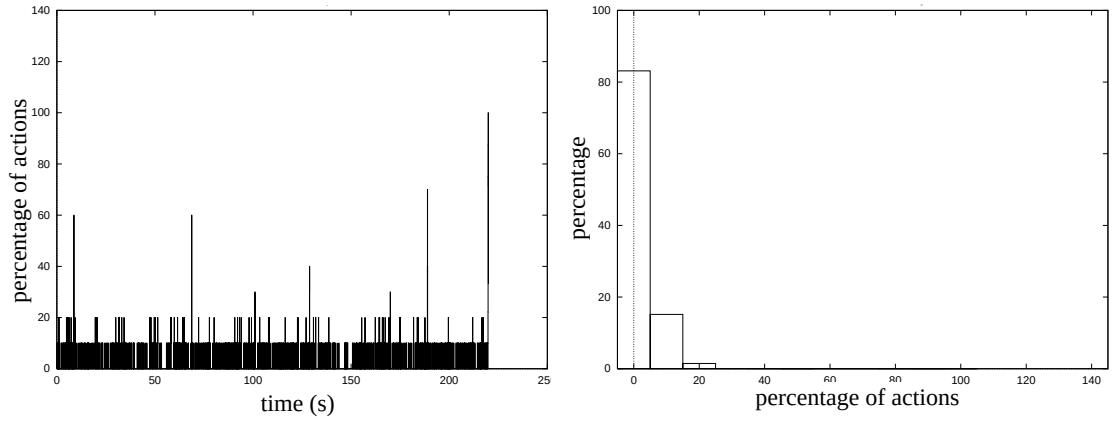

Figure 9. Percentage (left) and distribution (right) of lost actions by a local computer (Droopy, 23).

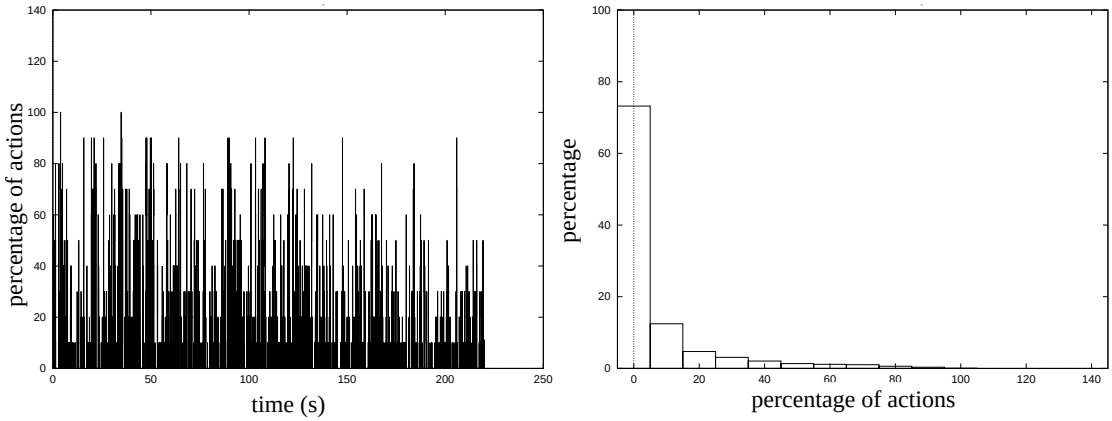

Figure 10. Percentage (left) and distribution (right) of lost actions by a Mbone computer (Hugo, 43).

This result is not very useful to define the frequency with which a participant should transmits a State ADU should be sized accordingly. In fact, we would need to evaluate also the probability to have consecutive losses of the same action to define the

level of redundancy. With the current game settings, the State ADU frequency is 25 per second. It is a lot more than the minimum required with the observed loss rates. But it could be too slow with higher loss rates, longer network delays, and more consecutive loss (a higher frequency will increase the load, and consequently increase the chance that one has to have consecutive loss).

Note that the frequency of the State ADU transmission is independent of the frequency of global game state computation (that should be 25 times per second). The frequency of State ADU transmission should be chosen to guarantee that each bucket will contain at least one ADU from each participant. We do not investigate this problem further because the addition of dead-reckoning should influence the behavior of the game in such a way that it should be possible to reduce the ADU transmission frequency even for higher loss rates.

3.4.3 Scalability

We have not studied scalability in depth for this first evaluation of MiMaze. We already have explained that the group problems will be a concern for the next generations of MiMaze. But with the current benchmarks, we were not able to notice a change in the game performance due to the number of participants.

The reasons might be that the number of participants remained small (less than 25) and that we had problems to re-synchronize dumps for any number of participants. The following curves have been realized for 3 or 4 different number of participants (from 5 to 20), and the differences are not significant enough to draw any conclusions on scalability. That explains why we have decided to present the following results independently of the number of participants (percentage of actions).

3.4.4 Bucket synchronization efficiency

To evaluate the efficiency of our synchronization mechanism, the metric we are using is the number of packets that the receiving entity did not re-synchronize because they were too late. Remember that from the network delay analysis (section 3.4.2), we expect the average un-synchronized action rate to be less that 15% (including un-synchronized actions due to losses).

Figures 11 and 12 give (respectively on a LAN and on the Mbone) the percentage of actions that are not re-synchronized by the bucket algorithm because they reached their destination (resp. droopy and hugo) after more than 100 ms. Action lost or corrupted do not appear on these figure (they have been analyzed in the previous section). Here again, the same type of observation can be made. First the rate of un-synchronized actions is much higher for hugo (Mbone) than for droopy (local). This is expected as hugo receives most of his actions with an average delay of 55ms. This is more difficult to understand for droopy that mostly receives local information. As for the loss rate, a comparison of un-synchronized participant to participant actions reveals that the un-synchronized actions at droopy are those coming from the remote computers (hugo, feydau, ceyx, etc.).

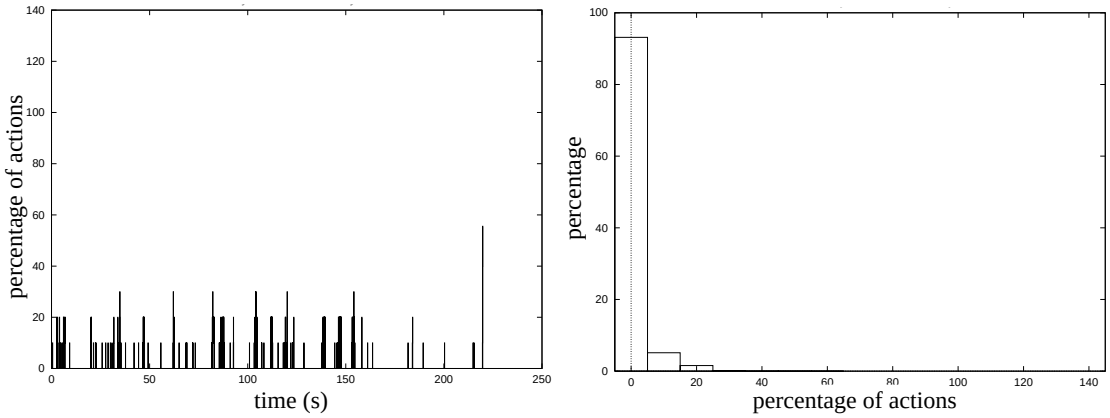

Figure 11. Percentage (left) and distribution (right) of un-synchronized actions by a local computer (Droopy, 23).

In figure 10, the mean un-synchronized action rate is less than 1%; and 6% over the Mbone (Figure 11). The addition of the ADUs lost (7%) and un-synchronized (6%) on the Mbone gives a total of 13% of ADUs missing for the Mbone participants, which is close to the 15% expected from the delay analysis.

The analysis of the distribution figures (10 right and 11 right) show that un-synchronized actions occur in less than 10% of the cases, so that in more than 90% of the time, there is no un-synchronized action (both locally and on the Mbone).

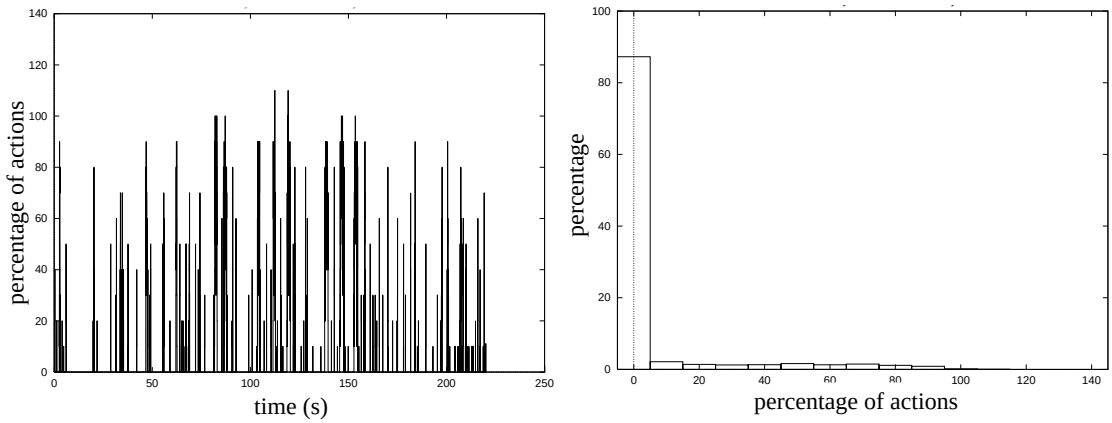

Figure 12. Percentage (left) and distribution (right) of un-synchronized actions by a remote computer (Hugo, 43).

It should be noticed that these results have been obtained with network delays that are much smaller than 100ms. Without dead-reckoning, and with the current standard deviations on network delays, the rate of unsynchronized actions would be

much larger for longer delays. But using such high delays would not help us to evaluate the efficiency of the bucket synchronization mechanism.

3.4.5 Consistency

We have chosen two different metrics to analyze the consistency of the game. First, we have measured the number of actions that are used by each participant to compute his local evaluation of the global state. With the current game setting, there should be one action per participant available in a bucket at the time this bucket is processed.

Figure 13 and Figure 14 show that the global game state is most of the time computed with 70% of the actions. This global state is computed with 100% of action in only 10% of the cases. Results are very similar on the Mbone and locally (the mean computed is 76% locally and 66% on the Mbone). The main difference comes from the standard deviation which is much higher for the remote participants (σ=27.05ms, for only 15.54ms locally).

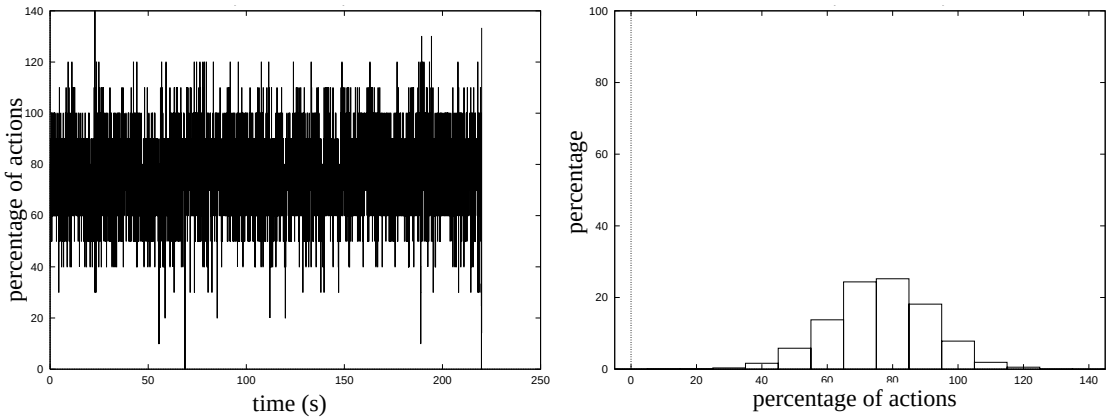

Figure 13. Percentage (left) and distribution (right) of re-synchronized actions by a local computer (Droopy, 23).

As we have no dead-reckoning in this implementation of MiMaze, we had to use the most recent action received when the current action was missing in the computed bucket. Because of the high bucket frequency, this had no visible effect on the game participants.

Before we move to the second metric, it should be noted that approximately 5% of the buckets are computed with no updated actions (Figure 14 right).

The second metric is to compare the locally computed global state with the reference state computed by the monitoring tool (and visualized by the 0 line on the vertical axis). In a perfect scenario (or with a server based architecture), all states should be on the reference line. This is obviously not the case for MiMaze!

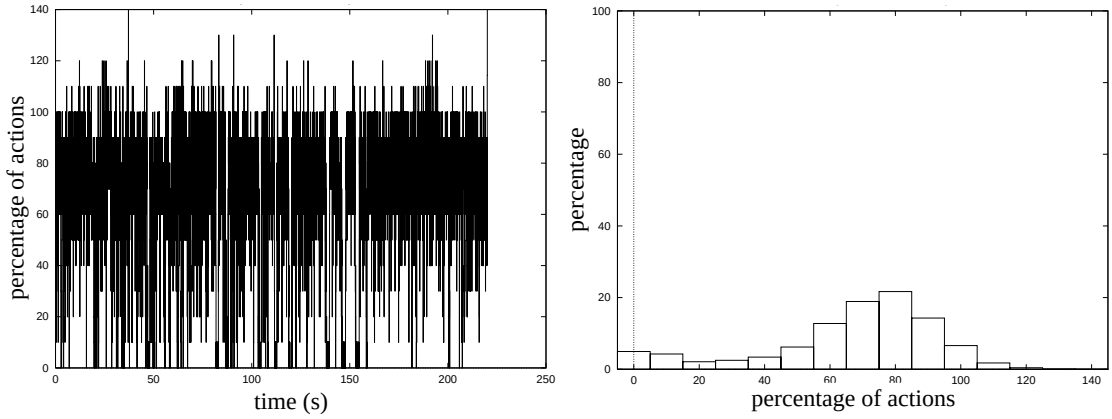

Figure 14. Percentage (left) and distribution (right) of re-synchronized actions by a local computer (Hugo, 43).

Figures 15 and 16 show the position of the global state as computed by droopy and hugo (respectively) to the reference state (which is the same for all participants). The vertical axis corresponds to a "drifting function" which is based on the number of actions used to compute the local state, and on the difference between the delay used to synchronize an action and the average delay (30 units on the vertical axis corresponds approximately to one action missing). This function has been chosen to render the fact that even if all actions are present, they do not necessarily belong to the same period of the game.

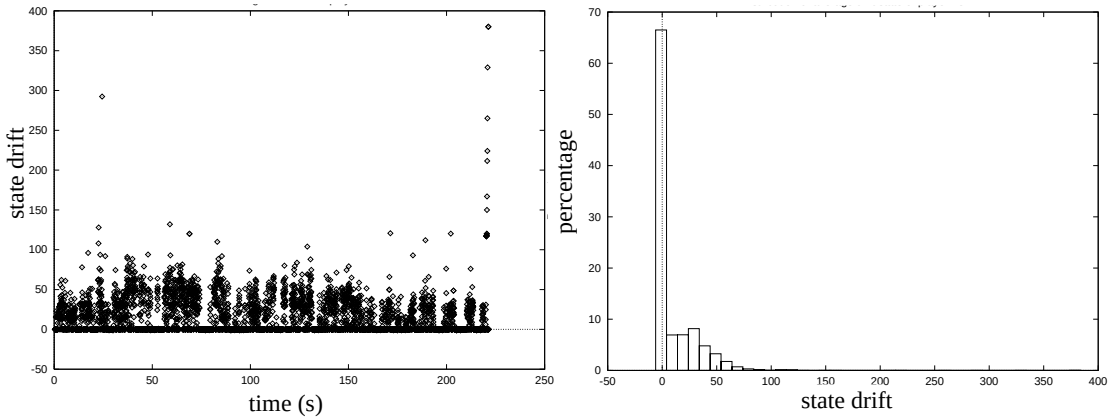

Figure 15. Comparison of the reference state to the global state as observed by Droopy (23)

Despite the results observed with the previous metric, we can see here that droopy and hugo have computed the reference state in more than 65% of the time. This is a

surprisingly good result that can certainly be explained by the nature of the application analyzed (PacMan in MiMaze do not move too fast, and 25 buckets per second is really high compared to the participant reaction time).

This is an interesting result that shows that depending on the nature of the application, reliability, as well as very costly synchronization mechanisms are not compulsory to render the real time capabilities of an interactive application.

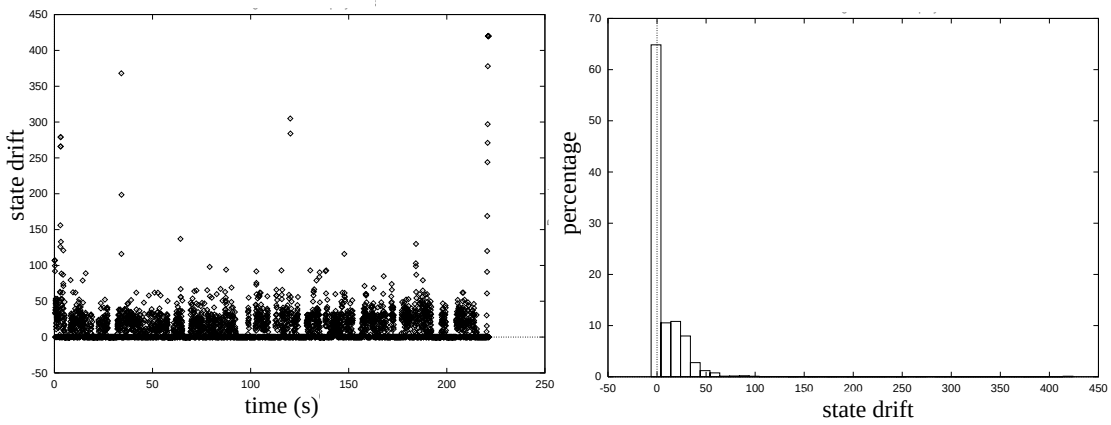

Figure 16. Comparison of the reference state to the global state as observed by Hugo, 43

The distribution figures (15 right and 16 right) are also very similar in both cases. It confirms that, because of the nature of the game, the network characteristics do not have a significant influence on a game session. For any participant, the state drift is most of the time limited to 50 units.

We did not present these figures for all participants because all participants on the same figure would not be readable. But we have verified that other participants follow the same behavior.

3.5 Discussion

The most important problem is to render the real-timeliness and the consistency of the game for all participants, whatever their network delay with the other participant is. With the current game and network setting, we have shown that for a human controlled application, there is no need for reliable transmission, and that the game is "consistent" in more than 65% of the time. Moreover, these results have been obtained with a pretty poor global clock mechanism. We expect to improve significantly the consistency of the game just by using a global clock network such as GPS that offers a 1 micro-second accurate clock everywhere in the world.

We have also shown in this experimental analysis that the most important parameter will be the frequency with which actions (State ADUs) are sent by each participant.

If this frequency is too slow, there could be not enough actions in each bucket to compute a realistic state; but the network will be less loaded and each participant will have more time to compute the buckets. On the other hand, a high frequency will load the network and each participants' CPU. Consecutive losses on the same action will become more frequent and participant might have to slow down the bucket frequency to avoid CPU congestion.

The bucket frequency is mostly independent of the action transmission frequency, and it should be chosen considering local parameters only such as the global state computing time (one bucket is processed anytime a new game state is displayed) and the velocity properties of the application. But slowing down the bucket frequency will not help to maintain the real-time properties of the game as long as the global state computing time is less than the interval of time between two buckets.

Last, the current game and network setting have been chosen to minimize the ratio of un-synchronized action. We have said in the introduction that the DIS standard as defined 100ms to be the synchronization delay in this type of application. A 150ms delay is also standardized for "loose" synchronization. The current implementation of MiMaze does not allow us to analyze the behavior of the game with average network delays higher than 70 to 80ms.

Consequently, our contribution of this work is to show that with a simple synchronization mechanism (the bucket mechanism), a fully distributed interactive application can work correctly on the Internet. Moreover, we believe that using an efficient global clock network will solve most of the synchronization problems (including total ordering, and inter-participant synchronization).

Our observations show that Dead-Reckoning is mandatory in a distributed interactive application such as a game or a DIS. Dead-reckoning will improve the efficiency of the game in various aspects:

- Re-synchronization in case of network delays higher than 100ms. Actions that will be received late (after 100ms) will be dead-reckoned to fill the next bucket to be processed. In the current implementation, these actions are simply discarded.

- Reduce action transmission frequency. We have explained earlier that DR can help to maintain an acceptable network and CPU load, depending on the application characteristics and on the network situation.

The next section is dedicated to the mechanisms that will be implemented in the next version of MiMaze. These mechanisms are expected to fix the current Mimaze deficiencies, and also to improve MiMaze, making it more robust to unpredictable network conditions.

4.0 Mechanisms for the Distributed Control

In this section, we identify the problems that will have to be solved to maximize the game consistency, interaction, and scalability.

4.1 Dead-Reckoning

We have shown before that Dead Reckoning (DR) is an essential mechanism to maintain the consistency of the application in a fully distributed environment. The transformation of the bucket synchronization mechanism with dead-reckoning is shown figure 17.

Basically, Dead reckoning is used when an action reaches a participant in more than 100ms. In that case, the action is too late to be processed in the corresponding bucket. It is consequently dead reckoned and the dead reckoned action goes to the next bucket to compute.

Dead-reckoning consequently guaranties that there will always be an action to compute for each participant of the session in any bucket. Dead-reckoned action are an estimation of what should be the new «state» of the avatar obtained from its most recently received action.

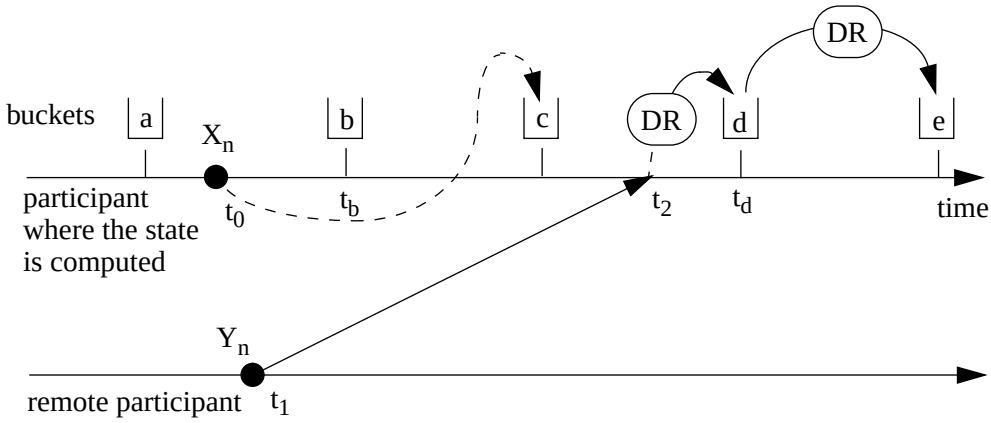

Figure 17. Bucket synchronization with dead-reckoning

For example, action *Yn* should be computed together with action *Xn*, e.g. in bucket *c*. *Xn* does not need to be dead-reckoned and is stored in bucket *c*. On the other hand, action *Yn* reaches its destination after that bucket *c* has been processed (or in more than 100ms). So *Yn* will be dead-reckoned and stored in bucket *d*, which is the next to be processed. If an action is missing in a bucket, this action will be dead-reckoned from the information found in the previous bucket (see the example between buckets *d* and *e* on Figure 17). Finally, if action *Yn+1* reaches its destination before bucket *d* is processed, *Yn+1* will replace *Yn* as it is more recent than *Yn*.

4.2 Control of actions and states transmission

As shown before, most of the information managed by a DIS application or by a distributed game does not require reliable transmission. Moreover, it is redundant by nature. Consequently, we have preferred an approach based on unreliable protocols. We believe that the nature of distributed games actions provides enough continuity to avoid any retransmission. So, there is no necessity for ARQ or for FEC mechanisms to be added.

The case of exception treatment is different. A collision information or a detonation are not continuous informations and, if a participant do not receive it, they will simply ignore it (that is the problem we had with scoring in MiMaze). This is not acceptable. Consequently, we have modified the structure of the ADUs to add redundancy to the exceptional actions.

4.2.1 Exception ADU

The Exception ADU is dedicated to all exception treatments (mostly collision and shots in the case of MiMaze). E ADU content just differs from S ADU in that it carries information on exceptional events only. E ADU has been separated from S ADU in order to use it with a different protocol, or on a different group. Such a need can happen with collisions for example that mostly affect two participants. We have observed during these experiments that the network delay to reach a point on the Mbone can be much higher that on a point-to-point transmission. In case of collision, the two avatars involved have to know it as fast as possible in order to compute the collision consequences before the collision really happens (see section 4.3 for more details). Sending E ADU point-to-point would in that case be more efficient than sending it to the whole group, on the Mbone. Anyway, E ADU is sent on the multicast group in order to advertise third-party avatars (and to avoid cheating).

4.2.2 ADU hierarchical encoding

Another way to make sure that exception information will reach all participants is to duplicate them in the State ADUs.

The optimal size of IP packets is 576 bytes. Most of the game actions (or State ADUs) are shorter than this minimum size. We consequently propose to use the available space to transmit duplicate of exception information, and if there is still room available, any other information of general interest (like scoring). The State ADU format described Figure 4 needs to be modified. The payload is now divided in two fields:

- The local field where a description of the sending avatar is given. This local part can also contain exception information if the avatar is involved in a collision for example.
- The global field where informations of general interest are given. This can be the local score, the detection of a collision between to other avatars, etc.

This structures is also very efficient from a network point of view as a receiver is not forced to read the general part but can simply ignore or discard it. Moreover, because the global field only uses the remaining bytes up to 576 bytes, adding this field does not increase the network load.

4.3 How to manage collisions

Collisions need a specific treatment. The reason is that when two elements collide, it is too late to compute the consequences of the collision. The consequences of a collision are numerous:

- disappearance of elements (a boat sinks for example),
- creation of new elements (smoke, parts, etc.),
- transformation of elements (a bullet can slow down its speed but keep a modified trajectory after the impact with a first element). The state of the other elements also changes in the collision.

These changes need to be computed before the collision in order to be applied immediately after (or even at the time of) the collision.

We have shown in the MiMaze experimental evaluation that when game avatars are "too close" to each other, it is impossible to re-synchronize them. The problem is not only to evaluate fast enough the consequences of a collision, but to anticipate the collision in order to avoid re-synchronization. We propose to implement two mechanisms in order to resolve the collision and avoid side-effects.

4.3.1 Source anticipation

The first problem with collisions is to detect them. Detect a collision at the time they occur is too late. The main problem with collisions is that before they occur, any evaluation on the effect of the collision is a supposition. Considering it take δt seconds to compute the effect of the collision, it is not possible to accept a change in the impact parameters within the last δt before this impact. This observation motivates us to anticipate the collision using a mechanism we call “source anticipation”. When a participant issues an action at time t, instead of synchronizing it and sending it to the group of participants, we apply dead-reckoning to the action to “see” the avatar at $t + \delta t$. If we observe a collision within δt, we consider this collision happens and we simultaneously send on the network the dead-reckoned action (State ADU) and the collision information (Exception ADU). The source anticipation mechanism has two important side-effects:

- It allows us to re-synchronize up to 100ms + δt network delays.
- Without the exception ADU carrying the collision information, the collision would have to be “detected” by everyone. Because of this separate transmission, all participants know without doubt that the collision happened.

4.3.2 Management of a collision domain

The second problem is the evaluation of the collision effect. We propose to associate to each game element a collision domain (or cell) that is a sphere centered on the element and which size is proportional to the speed of the element (Figure 18). Elements have no control on their collision domain. Collision detection is a function that is run when another element enters a collision domain. The collision function steps are:

1. contact the element that enters your cell, and exchange a set of collision data (resistance, speed, position, size, shape)
2. compute an estimated impact point (which can be no impact).
3. evaluate the consequences of the impact and prepare the after-impact situation.

This protocol must be processed until the collision occurs, or until the element leaves the collision cell. The current investigation lead to a point-to-point protocol. The main reason justifying a point-to-point protocol is that there is a real little chance that more than two elements enter in a collision at the same impact point. But collision can interact in term of consequences, and this should be resolved locally.

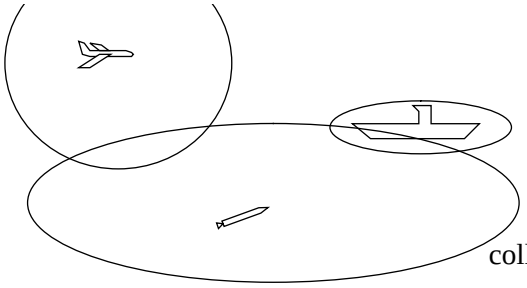

Figure 18. Collision detection

If an avatar enters a collision domain first, then the two avatars will compute the collision. When the two avatars will see each other, they will certainly try to either avoid or provoke the collision. But this is totally independent of the collision protocol that keeps computing the collision consequences with updated information.

4.4 Participant management

We will not develop in detail the problem of participant management. The goal here is simply to identify and highlight the problem that will be related to the management of large groups of participants with the current group infrastructure on the Internet. There will be mostly two types of problems: how to manage groups, and how to dynamically join or leave a group. We shortly address each of these problems.

4.4.1 Group management

Managing groups of 100,000 participants is not credible. There is no algorithm that can scale up to this size. The natural solution is consequently to put the session participants in sub-groups that can be organized on various criteria.

- Subdivision in logical groups. In such an application, there are natural sub-groups that can be made based on relationship among people, or center of interest. For example, if two armies participate to a battle, they will naturally belong to two different groups.
- Terrain based management. This is the approach proposed by the DIS. This division of the terrain in zones can be useful for example to reduce 3D display complexity, or to increase the scalability of the game.
- Entity based management. Each avatar can define a group (Figure 19) with all the avatars that are in his vision domain.The shape of the vision domain depends on the type of sensor used by the element.

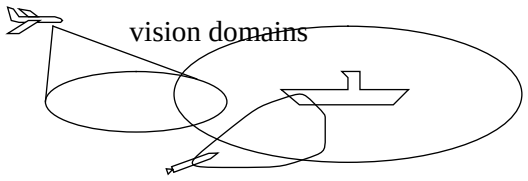

Figure 19. Game elements with their vision domain

- Subdivision in functional groups. Avatars could be grouped based on their nature (all planes would be in the same group). A finer classification would be to group all avatars with a radar, or with eyes.

It is not clear today which approach is the best. It is also not certain that only one approach will be used in a session. It is more likely that the session will be managed with a mixing of these four approaches, depending on the nature of the application, and on the dynamic of the session.

But it is clear that the multiplication of groups is dangerous and could create congestions on the network.

4.4.2 Join and leave latency

The join and leave cost is very important as it introduces latency in the processing of events. There are two different cost to be considered:

- the network level join latency with the associated IGMP traffic, and

- the application latency that is made of scenery or history information that have to be received by the participant before he can enter a group.

Specific join and leave mechanisms will have to be used to allow a participant that join a group to be active very quickly. Participants could join groups by anticipation to be ready to be active when they will really join the group. The only problems with this approach is that participants could have access to information that they should not know, and that they could join too many groups to be sure to have the good ones.

4.5 Distributed control of cheating

In a totally distributed architecture, there is no "referee" to guarantee that there is no participant cheating. On the other hand, in a classic client-server architecture, the server has a unique and global view of the session and he can guarantee that nobody is cheating.

Consequently, in a distributed architecture, a distributed mechanisms must be provided to eliminate wrong or corrupted actions. For example, there must be a way to identify and discard actions sent by a participants with erroneous time-stamps. But cheating in a distributed architecture is much complex than a simple time stamp authentication and all the difficulty will be to provide a distributed mechanism to control corruption (voluntary or un-voluntary).

5.0 Conclusion

The main contribution of this work is to show that with a simple synchronization mechanism (the bucket mechanism), a fully distributed interactive application can work correctly on the Internet. Moreover, we believe that using an efficient global clock network will solve most of the synchronization problems (including total ordering, and inter-participant synchronization).

To our knowledge, MiMaze is the first multiuser game to be designed with a totally distributed architecture, e.g. without server. Another major contribution of this work is to open this new research direction. Today, the most popular architecture in distributed systems is client-server. Totally distributed architectures scale better, and we have shown that it provides a good level of performance with potential scalability and better real-time properties.

A new release of Mimaze is being designed. The main features of this new implementation are:

- provide Dead-reckoning,
- resolve collision by source anticipation, and
- implement a new ADU type (called Exception ADU) and a new ADU structure for an optimized network transmission and control.

Other improvement such as group management and collision domain management are kept for further studies. We also keep for future work the extension of MiMaze with 3D VRML object and with video scenes (MPEG4).

6.0 References

[1] IEEE Standard for Distributed Interactive Simulation -- Application Protocols (IEEE Std 1278.1-1995). IEEE Computer Society. 1995.

[2] IEEE Standard for Distributed Interactive Simulation -- Communication Services and Profiles (IEEE Std 1278.2-1995). IEEE Computer Society. 1995.

[3] Steve Seidensticker and W. Garth Smith and Michael Myjak. "Scenarios and Appropriate Protocols for Distributed Interactive Simulation". Working Internet Draft <draft-ietf-lsma-scenarios-01.txt>. March 1997.

[4] J. M. Pullen and M. Myjak and C. Bouwens. "Limitations of Internet Protocol Suite for Distributed Simulation in the Large Multicast Environment". Working Internet Draft <draft-ietf-lsma-limitations-01.txt>, March 1997.

[5] J-C. Bolot, A. Vega Garcia, "Control mechanisms for packet audio in the Internet", Proceedings of IEEE Infocom '96, San Fransisco, pp. 232-239, April 1996

[6] D.L. Mills, "Network Time Protocol (Version 3) Specification, Implementation and Analysis", RFC-1305, March 1992.

[7] A. Cox, E. Luiijf, R. van Kampen, R. Ripley. "Time Synchronization Experiments". Proceedings of the 14th DIS workshop (dis-96-14-175). 1996.

[8] J. Czeranski, H-U. Kiel. "Softwarepraktikum Netzwerkprogrammierung unter Unix am Beispiel des Spiels", 1993/94, http://www.tu-clausthal.de/student/iMaze/.

[9] H. Schulzrinne, S. Casner, R. Frederick, V. Jacobson. "RTP: A Transport Protocol for Real-Time Applications", RFC-1889, January 1996.

[10] R. Ramjee, J. Kurose, D. Towsley, H. Schulzrinne, "Adaptive playout mechanisms for packetized audio applications in wide-area networks", Proceedings of Infocom '94, Toronto, Canada, pp. 680-688, April 1994.

[11] .H. Eriksson. "MBONE: The Multicast Backbone". Communication of the ACM. Vol. 37. pp. 54-60. August 1994.

[12] A. Goscinsky. Distributed Operating System, The Logical Design. Addision-Wesley publishing company. 1991.

Unité de recherche INRIA Lorraine, technopôle de Nancy-Brabois, 615 rue du jardin botanique, BP 101, 54600 VILLERS-LÈS-NANCY
Unité de recherche INRIA Rennes, IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, domaine de Voluceau, Rocquencourt, BP 105, LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur

Inria, Domaine de Voluceau, Rocquencourt, BP 105 LE CHESNAY Cedex (France)

ISSN 0249-6399