# Middleware P2P para la Sincronización de Eventos Discretos en una Simulación Distribuida de Sistemas que evolucionan en el tiempo

Lucas Guaycochea, Javier Luiso, y Federico Del Rio Garcia

Instituto de Investigaciones Científicas y Técnicas para la Defensa (CITEDEF),
San Juan Bautista de La Salle 4397, (B1603ALO) Villa Martelli, Argentina

{lguaycochea, jluiso, fdelrio}@citedef.gob.ar

**Abstract.** Se presenta un middleware para resolver la sincronización de eventos discretos distribuidos. El mismo es diseñado para ser utilizado en aplicaciones interactivas de simulaciones distribuidas de tiempo real. Este tipo de aplicaciones conllevan la simulación del avance o evolución del tiempo. Se utiliza un modelo de comunicación P2P sobre una LAN. La sincronización es resuelta a partir de un algoritmo de sincronización conservador basado en lo propuesto por Chandy y Misra. El middleware es exitosamente incorporado en una plataforma que permite el desarrollo de simuladores de entrenamientos distribuidos. Se muestran y analizan los resultados obtenidos experimentalmente en un escenario compartido de hasta 6 nodos. Los resultados muestran que el valor medio de las latencias de ejecución de los eventos originados en el sistema, se encuentra convenientemente acotado permitiendo su uso en simuladores distribuidos interactivos de tiempo real.

**Keywords.** Sistemas distribuidos, P2P, simulación distribuida, sincronización de eventos.

## 1 Introducción

Se llama sistema distribuido a un conjunto de computadoras independientes que se presentan a sus usuarios como un sistema único y coherente [1]. Un sistema distribuido permite compartir recursos y datos entre distintas computadoras. Entre las ventajas que presenta este tipo de sistemas se destacan: la disminución del costo de hardware, la optimización del rendimiento al permitir paralelismo, y la sencilla resolución de problemas de naturaleza distribuida. Las aplicaciones que adoptan este esquema de trabajo son denominadas aplicaciones distribuidas. Las computadoras que participan dentro del mismo sistema distribuido se encuentran interconectadas entre sí sobre una red que posibilita la comunicación entre ellas. Para facilitar el desarrollo de aplicaciones distribuidas es deseable contar con un módulo de software que permita resolver las comunicaciones abstrayendo al programador de la distribución del sistema. A este módulo de software lo llamamos middleware.

En este trabajo se presenta un middleware que permite, no sólo la comunicación, sino también la sincronización de eventos discretos para llevar a cabo una simulación en tiempo real. El tipo de simulación de interés es aquella donde se simula la evolución en el tiempo de un escenario virtual en el que participan usuarios en distintas computadoras. Las acciones de los usuarios originan eventos en el mismo escenario donde evoluciona la simulación. Además, las aplicaciones se caracterizan por ser interactivas, donde el usuario espera observar el resultado de sus acciones de manera inmediata. Como ejemplo de este tipo de aplicaciones interactivas podemos mencionar a los juegos multijugador y a los simuladores de entrenamiento distribuidos. Estas aplicaciones se caracterizan por representar un ambiente virtual tridimensional donde los usuarios interactúan de manera concurrente.

Las aplicaciones interactivas demandan entonces que la latencia de ejecución de un evento debe encontrarse acotada. La latencia debe ser tal que el usuario no perciba una demora entre su acción y la respuesta esperada. Como ejemplo, para un juego del tipo first-person shooter, que por características es uno de los más demandantes en este aspecto entre este tipo de aplicaciones, es necesaria una latencia menor a los 180 ms. [2]. Por ende, este es un requerimiento fuerte para un sistema distribuido. Además, los eventos deben sincronizarse para garantizar la consistencia de la realidad que se desarrolla en la simulación. En consecuencia, el tiempo insumido en la comunicación, sincronización y ejecución de un evento debe encontrarse acotado por debajo de 180 ms.

El middleware que se presenta propone una comunicación P2P entre los participantes del contexto simulado. Se utiliza un método conservartivo de sincronización basado en lo propuesto por Chandy y Misra [3,4]. El uso de un método de sincronización optimista es descartado ya que el tipo de aplicaciones de interés no soportan la realización de un roll-back, mecanismo necesario para asegurar la consistencia en este tipo de métodos.

Por otro lado, el middleware resuelve internamente la generación a nivel local de un tipo de evento que implica el avance del tiempo en la simulación, llamado evento *tick*. Este tipo de evento se mantiene sincronizado con respecto a los eventos externos que se reciben. Además, el middleware necesita que le sea provisto un reloj donde consultar las marcas de tiempo o timestamp que se utilizan para la sincronización de los eventos. Por último, el middleware brinda una entrada para recibir una señal de reloj o *clock input signal* para la generación periódica de los eventos *tick*.

Tomando la configuración para el middleware descripta más adelante (Sección 5), la latencia en el ordenamiento está dominada por el tiempo de generación de los eventos *tick* por sobre cualquier otro factor (latencias de comunicación, de sincronismo, etc.).

## 2 Marco Teórico

### 2.1 Arquitecturas de comunicación

Entre las arquitecturas o modelos de comunicación utilizados a nivel de software, encontramos la arquitectura cliente-servidor, donde el sistema es pensado en términos de nodos clientes que requieren servicios de un nodo servidor [1]. Aplicado este es-

quema en el ámbito de simulaciones distribuidas, el estado de la simulación es mantenido únicamente en el servidor, cada evento producido por un participante (cliente, en los términos del modelo) es enviado al servidor para que este lo procese, evolucione el estado y luego comunique el nuevo estado a todos los clientes. El servidor es responsable de realizar todos los procesos de cálculos y el procesamiento de datos involucrados en los cambios de estado.

Otro modelo utilizado es la arquitectura P2P, que puede definirse como un sistema donde sus participantes o nodos son iguales y no tienen capacidades especiales que lo diferencien del resto. La aplicación de esta arquitectura implica que no hay un nodo central de procesamiento donde resida de manera única el estado de la simulación. Cada nodo tiene la responsabilidad de administrar la evolución y de mantener una réplica local del estado de la simulación. En consecuencia, es necesario mantener consistente dicho estado en cada nodo porque los usuarios deben percibir la misma realidad en todo momento.

### 2.2 Arquitectura P2P. La sincronización

La arquitectura P2P requiere de un mecanismo que asegure la consistencia entre las réplicas del estado de la simulación presentes en cada nodo, donde la evolución del estado depende tanto de las acciones realizadas por el usuario local como aquellas realizadas por los usuarios remotos. Cada acción da origen a un evento que se comunica a través de la red para notificar dicha acción a todos los nodos. Si en los nodos los eventos se ejecutan en el orden temporal en que se originaron entonces en todo momento se lleva a cabo la misma secuencia de acciones y por la tanto se garantiza la consistencia del estado de la simulación [5].

En un ambiente distribuido no puede garantizarse que los eventos que se generan en distintos nodos arriben a cada uno de ellos en el mismo orden temporal en que fueron originados, debido a los procesos de comunicación involucrados. Esta realidad impone que se necesite de un mecanismo que sea capaz de ordenar los eventos que llegan a un nodo.

Para lograr el ordenamiento se analiza la dependencia causal entre ellos. Si dos eventos ocurren en momentos distintos y uno de ellos depende causalmente del otro, el estado al que evolucionará la simulación depende del orden en que dichos eventos se ejecuten. Debe garantizarse el ordenamiento causal de los eventos en cada nodo a fin de que la simulación evolucione consistentemente en todos ellos. Básicamente sincronizar se trata de decidir cuando es consistente ejecutar un evento. Ese momento es cuando todos los eventos anteriores, con los que existe una dependencia causal, ya fueron ejecutados.

Los mecanismos de sincronización pueden clasificarse en conservadores u optimistas, según como manejen las inconsistencias [6,7]. Los primeros buscan prevenir la inconsistencia verificando que los eventos sean ejecutados sólo cuando sea seguro hacerlo. En cambio, los últimos, ejecutan los eventos al disponer de ellos. Cuando se detecta una inconsistencia se intenta corregirla deshaciendo las últimas operaciones realizadas, mecanismo conocido como roll-back.

### 2.3 Aplicación de un algoritmo de sincronización

Chandy y Misra [3], y Misra [4], proponen un algoritmo que puede utilizarse para realizar la sincronización con un enfoque conservador. El mismo tiene aplicación en el tipo de problema que nos proponemos resolver.

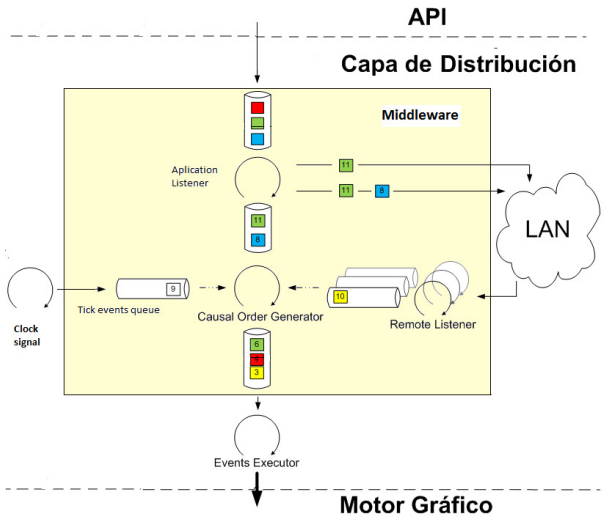


**Fig. 1.** Procesos y colas del middleware.

En los trabajos anteriormente citados, Chandy y Misra introducen un mecanismo para simular la evolución de un sistema físico de manera distribuida. Proponen que el sistema puede verse como un conjunto de procesos físicos relacionados que se comunican exclusivamente por mensajes y se encuentran distribuidos. Un proceso físico dado puede o no, recibir mensajes de otros, y a su vez, puede o no, enviar mensajes a sus pares.

En la simulación, cada proceso físico es representado por un proceso lógico. Cada proceso lógico puede avanzar su simulación un paso en el tiempo si en ese instante ya ha recibido los mensajes de los procesos que se comunican con él. Para esto, el instante de tiempo en el que un mensaje es generado es codificado junto con este o, dicho de otra manera, los mensajes tienen una marca de tiempo. De este modo, para que la simulación no se bloquee y pueda avanzar en todos los procesos lógicos, en cada paso de tiempo, cada uno de ellos siempre envía a sus destinatarios un mensaje: si recibe un mensaje producido en el proceso físico, lo comunica; sino, envía un mensaje vacío, o nulo, que sólo tiene la marca de tiempo. Estos mensajes nulos son los que permiten la evolución de la simulación del sistema físico.

Este algoritmo, entonces, permite la sincronización de eventos discretos distribuidos para la simulación de un sistema de manera asincrónica.

Recapitulando sobre las características de las aplicaciones interactivas de interés, pueden encontrarse analogías con lo planteado por Chandy y Misra. La situación a simular o el mundo virtual donde participan los usuarios es equivalente al sistema físico, donde cada participante representa un proceso físico. Como cada usuario desea percibir la totalidad del mundo virtual, las acciones de cada uno de ellos deben comunicarse a los demás. De este modo, al momento de avanzar la simulación en cada nodo deben encontrarse disponibles las acciones realizadas por todos los participantes hasta ese instante.

## 3 Diseño del Middleware

Desde el punto de vista de sistemas distribuidos el middleware genera un orden temporal para los eventos generados en el sistema. Además, se basa en una arquitectura P2P y utiliza el patrón *Productor-Consumidor*. La comunicación se resuelve utilizando colas de mensajes en forma local y canales TCP para los nodos remotos.

### 3.1 Procesos y Comunicaciones en el Middleware

Rosenfeld[8] utiliza el algoritmo de Chandy y Misra proponiendo tres procesos fundamentales; uno para detectar y difundir los eventos locales, otro para recibir eventos remotos y un último proceso que ordene todos los eventos en forma temporal a partir de la marca de tiempo de cada uno de ellos.

El middleware debe soportar las siguientes funcionalidades: recibir los eventos generados por la aplicación usuaria local, transmitir los eventos a los peers que pertenecen al contexto, asignar la marca de tiempo a cada evento local, recibir los eventos de los demás peers y ordenarlos según su marca de tiempo. Para hacer esto el middleware tiene una serie de procesos o tareas y colas de mensajes que son recursos compartidos (Fig.1). Estas colas de mensajes se utilizan de manera sincrónica y bloquean a las tareas que las acceden hasta tanto tengan garantizado el acceso exclusivo a las mismas. La primera tarea es el *AplicationListener*; mediante una cola de mensajes de entrada recibe los eventos locales, les coloca la marca de tiempo, los transmite utilizando un canal TCP a cada uno de los peer asociados al contexto y luego los encola en la cola de mensajes de salida. Esta tarea espera eventos de la capa superior con un timeout, si el mismo se vence genera un evento nulo y le pone una marca de tiempo. Luego, al igual que a los eventos recibidos de la aplicación usuaria, lo transmite a los peers utilizando los canales TCP y lo encola en su salida. Este evento nulo, generado ante la ausencia de un evento producido por la aplicación usuario, es el que permite la evolución del proceso físico que menciona Chandy y Misra.

En este punto el algoritmo se diferencia del propuesto por Chandy y Misra, ya que para optimizar el uso de la red de comunicaciones y tener un esquema más asincrónico, el evento nulo no es generado en cada avance discreto del tiempo. Este evento es generado después de una determinada cantidad de tiempo de inactividad, para acotar la latencia máxima.

Por cada peer asociado al contexto, en cada nodo el middleware tiene una tarea *RemoteListener* (es decir que si un contexto tiene asociados n peers, el i-ésimo nodo posee n tareas *RemoteListener*); esta tarea recibe mensajes del canal TCP asociado a un peer y los encola en una cola también asociada a dicho peer.

Para permitir el avance del tiempo en la simulación se necesita que el middleware genere eventos tick, para esto posee una cola de mensajes donde se encolan estos eventos. Un evento tick se interpreta como que el tiempo ha avanzado un periodo determinado y conocido, por lo tanto esto implica una evolución de todo el modelo acorde al tiempo transcurrido. Los eventos tick se generan a partir de la funcionalidad de la API que permite la entrada de una señal de reloj.

Por último, la tarea *CausalOrderGenerator* es la que ordena temporalmente los eventos generados en el sistema. Utiliza una cola de mensajes para los eventos locales (salida de *AplicationListener*), una cola de mensajes por cada peer que pertenece al contexto (salida de cada tarea *RemoteListener*), la cola de mensajes de eventos tick y una cola de eventos ordenados (salida de eventos ordenados). En cada ciclo de ejecución consulta los mensajes en el frente de cada una de las colas, desencola el de menor marca de tiempo y lo encola en la salida de eventos ordenados. Dado el enfoque conservador del algoritmo, si una cola de mensajes se hallase vacía el proceso queda bloqueado hasta que llegue un evento a dicha cola. Los únicos eventos que no se encolan en la salida de eventos ordenados son los eventos nulos generados por el mismo middleware.

### 3.2 Sincronización de Relojes. Evolución del tiempo

En toda simulación la evolución del tiempo se simula a partir de sucesivos incrementos discretos de un contador o reloj, que representa un avance del tiempo en un paso definido. Como el paso del tiempo puede implicar una modificación en el estado de los elementos que conforman el modelo de la simulación, este incremento también debe estar sincronizado con los demás eventos.

En cada nodo hay un proceso que actúa como reloj local para simular el avance del tiempo. Este reloj es consultado por el middleware cuando éste asigna la marca de tiempo a cada evento local. Además, es este proceso quien genera la señal de reloj que es utilizada por el middleware para generar los eventos *tick*.

Cada nodo, debido a sus características de hardware y la carga con la que se encuentre trabajando, presentará desvíos en la evolución de su reloj respecto de los demás. Por este motivo es necesario que uno de estos relojes se tome como referencia (reloj global) para sincronizar, cada cierto tiempo, cada reloj local. En cada nodo, cuando corresponda, se consulta el estado del reloj global y como resultado de la comparación con el reloj local se determina si se el mismo se ha adelantado o atrasado. La corrección para volver al estado de sincronización con el tiempo global, se realizará de manera gradual. Para ello el reloj local modifica la velocidad de avance de su contador interno. Por último, el reloj de referencia o reloj global se sincroniza con el reloj interno de la computadora donde se encuentra, con el mismo mecanismo mencionado anteriormente.

# 4 Caso de Aplicación

## 4.1 Caso de Uso en Plataforma de Simulación Distribuida

El middleware presentado es usado como parte de una Plataforma de Simulación Distribuida, que consta de 3 capas de software (Fig. 2) [9]. La capa inferior se corresponde al motor gráfico. La capa intermedia o Capa de Distribución es la responsable del manejo de la comunicación y la distribución. Por último, la capa superior o API de la plataforma reproduce las funcionalidades brindadas por el motor gráfico agregando la posibilidad de crear contextos de simulaciones distribuidas.

El motor gráfico es utilizado para la representación de escenarios virtuales para el desarrollo de videojuegos y simuladores de entrenamiento, llevando a cabo tanto su evolución en el tiempo como su visualización o renderizado.

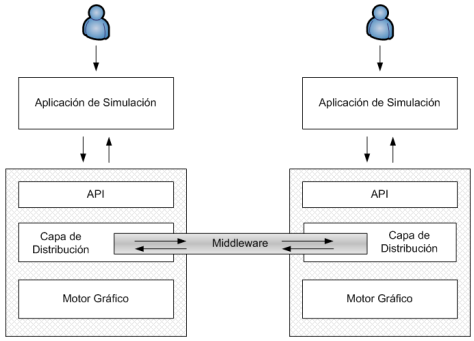


**Fig. 2.** Arquitectura de la Plataforma de Simulación Distribuida.

Esta plataforma es usada exitosamente para el desarrollo de simuladores de entrenamiento distribuidos. Un simulador distribuido requiere brindar a distintos usuarios en distintas computadoras la posibilidad de interactuar en la misma realidad simulada. La Capa de Distribución se encarga de resolver la comunicación entre las distintas computadoras para lograr ese objetivo. Para esto, utiliza el middleware presentado en la sección anterior para abstraer los aspectos concernientes a la distribución, encapsulando las comunicaciones y la sincronización. Por un lado, se ingresan en el middleware los eventos provenientes del usuario local que deben ser distribuidos y, por otro lado, se obtienen el total de los eventos temporalmente ordenados para ejecutar sobre el motor gráfico. La incorporación del middleware a la plataforma de simulación permite ahora que las aplicaciones presentes en cada computadora trabajen sobre el mismo escenario virtual sin tener que preocuparse por la distribución de las acciones que ocurre en cada una de ellas.

Por último, cabe mencionar a modo de ejemplo, que la plataforma ha permitido el desarrollo de dos simuladores distribuidos diferentes. El primero, un simulador de vuelo que ofrece un campo de visualización amplio (180º) utilizando tres computadoras. El segundo, un simulador de batalla virtual de vehículos de combate, con la parti-

cipación de hasta 3 vehículos y un instructor en la misma situación simulado, cada uno en su respectiva PC.

## 5 Resultados

Para la evaluación del middleware presentado utilizamos una configuración de prueba emulando el ambiente donde la Plataforma de Simulación es utilizada. Es decir, PCs conectadas en una LAN participando de un mismo contexto. Por ende, se utilizaron 6 nodos (PC Intel Core i7, 4 GB de memoria RAM, sobre el sistema operativo Windows 7 Ultimate) sobre una red LAN de 100 Mbps.

El middleware se configuró con un período de generación de eventos tick de 50 ms. y un timeout para la generación de eventos nulos de 75 ms. Se utiliza una aplicación cliente generando eventos a razón de 10 eventos por segundo, tomando los primeros 1500 eventos generados a partir de que todos los participantes están activos. Las pruebas se realizaron utilizando entre uno y seis peers.

Es de interés medir la latencia de comunicación y sincronización, es decir el tiempo transcurrido entre la entrega de un evento al middleware y el momento que este lo devuelve ordenado; esto se mide sólo para los eventos generados localmente. Se utilizaron dos esquemas de pruebas: la primera sin contemplar el tiempo que le lleva a la aplicación cliente el procesamiento de los eventos, y la segunda , incluyendo la simulación del tiempo necesario para ejecutar cada evento.

Los tiempos requeridos para ejecutar los eventos fueron medidos experimentalmente en los casos de aplicación exitosos mencionados en la sección 4, utilizando la plataforma descripta. Se eligieron los tiempos de procesamiento a simular usando una cota superior al peor caso experimental obtenido. Los tiempos son de 10 ms. para el avance del tiempo en la simulación a partir de los eventos tick, y entre 1 ms. y 5 ms. para los eventos producidos por la aplicación usuaria.

Las Figuras 3 y 4 muestran los resultados obtenidos. Para todos los casos del primer esquema de pruebas, las latencias introducidas por la comunicación y sincronización se encuentran por debajo de los 82 ms. para el 95% de los eventos (Fig 3 – Izq.). Cuando se contempla el procesamiento de los eventos, el 95% termina su ejecución antes de los 110 ms. (Fig. 3 – Der.). Por último en la Tabla 1 se muestran las latencias promedios y máximas obtenidas.

Si bien las latencia de comunicación y sincronización aumentan a medida que aumenta la cantidad de peers en el contexto, esta nunca excede el máximo fijado en 180 ms. y las latencias medias están muy por debajo de este límite, así como las correspondientes al 95% de los eventos.

Recapitulando, en el mecanismo de sincronización utilizado el período de eventos tick impacta en el proceso que ordena temporalmente los eventos, ya que si no dispone de un evento tick en la cola de mensajes no podrá procesar ningún tipo de evento (local o remoto) hasta recibir uno. O dicho de otra manera, cuando se bloquea esperando un evento tick se acumulan en las colas tanto los eventos locales como los eventos remotos de los demás peers. Al recibir el evento tick, procesa todos los eventos acumulados con una marca de tiempo menor a la del tick.

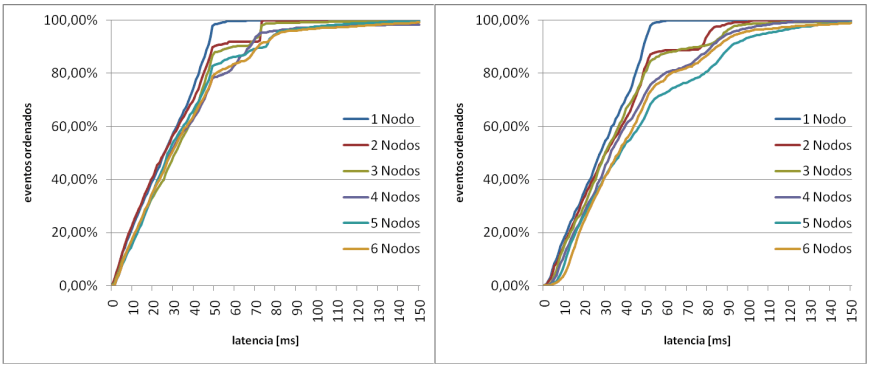


**Fig. 3.** Los gráficos muestran el porcentaje de eventos ordenados para una determinada latencia. (Izq.) Sin contemplar el procesamiento de los mismos. (Der.) Contemplando el procesamiento de los mismos.

Dado el mecanismo de sincronización utilizado, es de esperar que la latencia media de comunicación y sincronización de un contexto con un sólo peer sea aproximadamente igual a la mitad del período de generación de eventos tick. Esto se cumple experimentalmente como puede observarse en los resultados (Tabla 1).

En consecuencia, en un contexto de más de un peer, se puede estimar la latencia media originada sólo por la comunicación como la diferencia entre el valor obtenido para este contexto y la latencia media del período de generación de eventos tick.

Por último, notamos que el período de generación de eventos tick es el factor que tiene mayor incidencia en las latencias medias obtenidas, siendo despreciables en la configuración actual los tiempos insumidos en la comunicación y la sincronización.

**Table 1.** Latencias medias y máximas obtenidas en las pruebas: (a) sin contemplar el procesamiento de eventos; (b) contemplando el procesamiento de eventos.

| Cantidad de peers | Media s/ejec. | Máx. s/ejec. | Media c/ejec. | Máx. c/ejec. |
|---|---|---|---|---|
| 1 peer | 27,4 ms | 73,8 ms | 29,3 ms | 66,9 ms |
| 2 peer | 29,6 ms | 76,4 ms | 34,9 ms | 110,0 ms |
| 3 peer | 33,7 ms | 174,2 ms | 36,4 ms | 178,0 ms |
| 4 peer | 36,1 ms | 170,2 ms | 40,6 ms | 161,0 ms |
| 5 peer | 34,9 ms | 177,4 ms | 46,8 ms | 173,0 ms |
| 6 peer | 36,7 ms | 179,2 ms | 44,3 ms | 177,0 ms |

## 6 Conclusiones

En este artículo se ha presentado un middleware para aplicaciones distribuidas que resuelve la comunicación y sincronización de eventos discretos. El mismo posibilita el ordenamiento temporal de los eventos para resolver consistentemente simulaciones distribuidas de tiempo real donde además esta involucrado en la simulación el avance

del tiempo. El middleware implementa un mecanismo de sincronización conservador basado en lo propuesto por Chandy y Misra.

Por otro lado, el middleware esta orientado a ser utilizado para el desarrollo de aplicaciones interactivas donde la latencia de ejecución de cada evento debe encontrarse por debajo de 180 ms. A partir de los resultados puede observarse que la latencia debida a la comunicación y sincronización que introduce el middleware se mantiene muy por debajo de este valor umbral. Este resultado es el deseado ya que deja tiempo libre para la ejecución de cada evento. Los resultados se obtuvieron realizando pruebas con hasta 6 computadoras trabajando sobre un mismo contexto.

Por último, la plataforma de simulación que hace uso del middleware ha sido exitosamente utilizada en el desarrollo de distintos simuladores distribuidos como se ha ejemplificado en la Sección 4.1.

## Referencias


1. Tanenbaum, A., V. Steen, M.: Distributed Systems, Principles and Paradigms. Prentice Hall, (2002)
2. B. Knutsson, H. Lu, W. Xu, and B. Hopkins. Peer-to-Peer Support for Massively Multiplayer Games. In INFOCOM'04, Hong Kong, China, March 2004
3. Chandy KM, Misra J: Distributed Simulation: A Case Study in Design and Verfication of Distributed Programs. IEEE Transactions on Software Engineering, Vol. SE-5, No. 5, September 1979.
4. Misra J.: Distributed Discrete Event Simulation. ACM Computing Surveys (CSUR), 1986.
5. S. Ferretti. Interactivity maintenance for event synchronnization in massive multiplayer online games (ph.d. thesis). Technical Report UBLCS-2005-05, 2005.
6. Eric Cronin, Anthony R. Kurc, Burton Filstrup, and Sugih Jamin. An efficient synchronization mechanism for mirrored game architectures. Multimedia Tools Appl., 23(1):7–30, 2004
7. Kyung-Seob Moon, Vallipuram Muthukkumarasamy, and Anne T. Nguyen. Reducing network latency on consistency maintenance algorithms in distributed network games. In Proceedings of the IADIS International Conference: Applied Computing 2006, 2006.
8. Rosenfeld, M.: Un framework para comunicación Peer-to-peer en juegos multiusuarios. Tesis de Grado. Instituto de Informática, Universidad de La Plata, 2009.
9. Luiso, J., Guaycochea, L., Abbate, H.: Simuladores de entrenamiento distribuidos: Plataforma de desarrollo para ocultar los aspectos de la distribución. 1er Workshop Argentino sobre VideoJuegos, WAVI 2010, 2010.