Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

TESIS DE LICENCIATURA: INFORME CIENTÍFICO

Implementación de un entorno para estudios comparativos de simulación paralela y distribuida

Edgardo G. Szulsztein
L.U. 790/91
edgardo@dc.uba.ar

Director: Dr. Gabriel Wainer

Pabellón 1 - Planta Baja - Ciudad Universitaria
(1428) Buenos Aires
Argentina

http://www.dc.uba.ar

Marzo 2001

Contenido

Figuras

Algoritmos

Tablas

1. Introducción

1.1. Sistemas y Modelos

Para comenzar, daremos algunas definiciones. En primer lugar, llamaremos sistema a una entidad real o artificial. De hecho, no existe una definición de sistema que tenga una aceptación general. Se llama sistema a una parte de una realidad, restringida por un entorno. Está compuesto por entidades que experimentan efectos espacio-tiempo y relaciones mutuas. También se dice que un sistema es un conjunto ordenado de objetos lógicamente relacionados que atraviesan ciertas actividades, interactuando para cumplir ciertos objetivos.

Llamaremos modelo a una representación inteligible (abstracta y consistente) de un sistema. En muchos casos no se puede resolver un problema directamente sobre un sistema real, por ende razonamos sobre modelos. El proceso de pensar y razonar acerca de un sistema resaltando la reacción de un modelo se llama modelado de sistemas.

Para estudiar sistemas complejos, la idea es partir haciendo un modelo del sistema que se quiere estudiar, y se estudian problemas del sistema real estudiando el modelo.

1.2. Modelos y Simulación

Se define a la simulación como la reproducción del comportamiento dinámico de un sistema real en base a un sistema con el fin de llegar a conclusiones aplicables al mundo real [Gia96].

Luego, surgen preguntas tales como: ¿Porqué se hacen modelos de los sistemas? ¿Por qué realizamos simulación de los mismos? El motivo es que en muchos casos no se puede experimentar directamente sobre el sistema a estudiar, o se desea evitar costos, peligros, etc. En la actualidad existe una gran variedad de aplicaciones muy complejas en las que se usan modelos y/o simulación, que van desde manufactura

hasta diseño de circuitos para computadoras, pasando por aplicaciones bélicas y estudio de experimentos complejos. Las características comunes a estos sistemas son su complejidad y la falta de herramientas de evaluación de desempeño adecuadas.
El uso de simulación permite experimentación controlada, compresión de tiempo (una simulación se realiza en mucho menos tiempo que el sistema real que modela), y análisis de sensibilidad. Otra gran ventaja es que su uso no afecta al sistema real, ya que es independiente del mismo; incluso, el sistema podría no existir.

1.3. Paradigmas de modelado

Durante siglos el desarrollo de sistemas dinámicos estuvo basado en el estudio de modelos de ecuaciones diferenciales ordinarias y parciales. Estas permitieron modelar exitosamente los sistemas dinámicos encontrados en la naturaleza (de hecho, fueron exhaustivamente aplicados a través de los siglos en el campo de la física). Pero, por otro lado, la tecnología moderna ha permitido que el hombre cree sistemas dinámicos que no pueden ser descritos fácilmente por medio de ecuaciones diferenciales ordinarias o parciales. Como ejemplos de tales sistemas podemos mencionar líneas de producción o ensamblado, las redes de computadoras y comunicaciones, los sistemas de control de tráfico (de aire y tierra), los sistemas de control militar, etc. En estos sistemas, la evolución en el tiempo depende de interacciones complejas de varios eventos discretos y de su temporalidad, tales como la llegada o partida de un trabajo, y la iniciación o finalización de una tarea, etc. El "estado" de tales sistemas sólo cambia en instantes discretos de tiempo en lugar de hacerlo continuamente [Ho89]. La simulación aparece como una alternativa para estudiar el comportamiento de estos sistemas complejos.
Una de las primeras aplicaciones de simulación con computadoras fue el proyecto Manhattan, donde se estudió la difusión aleatoria de neutrones para el desarrollo de la bomba atómica, usando métodos de Montecarlo. El impacto de la tecnología de computadoras ha tenido gran influencia en el desarrollo de técnicas de simulación, y en

la actualidad existe hardware, interfaces con el usuario y herramientas de programación que influenciaron los métodos teóricos existentes.

La gran variedad de paradigmas de modelado puede clasificarse de acuerdo a distintos criterios:

- Con respecto a la base de tiempo, hay paradigmas a tiempo continuo, donde se supone que el tiempo evoluciona de forma continua (es un numero real), y a tiempo discreto, donde el tiempo avanza por saltos de un valor entero a otro (el tiempo es un entero).
- Con respecto a los conjuntos de valores de las variables descriptivas del modelo, hay paradigmas de estados o eventos discretos (las variables toman sus valores en un conjunto discreto), continuos (las variables son números reales), y mixtos (ambas posibilidades) [Gia96].

1.4. Simulación centralizada y distribuida

Es posible implementar la simulación de un sistema en dos arquitecturas bien opuestas: una centralizada, en donde se realiza la simulación en un único equipo procesador, o distribuida, en donde actúan equipos multiprocesadores (con la posibilidad de que tengan memoria compartida) o equipos heterogéneos de una red de área local y/o extendida.

Estudiamos mediante un ejemplo las posibilidades de cada tipo de simulación.

Consideremos un sistema de producción en serie, como puede ser el de una fábrica, en donde ingresa el material a un puesto de trabajo A, se trabaja sobre el mismo, realizando soldaduras y ensamblando distintas plaquetas, y se lo envía hacia el puesto de trabajo B, donde se actúa sobre el producto resultante de la estación anterior, y así sucesivamente, hasta que se llega a la última estación, en donde se almacena el producto obtenido en un repositorio. Tomemos como ejemplo una fábrica de productos electrónicos, y la manera en que elabora minicomponentes. Para tal fin tiene dos líneas de "producción en serie": una para producir el equipo en sí, y otra para la elaboración de los parlantes. Ambas líneas depositan el producto terminado en un mismo

repositorio, ya que se debe empaquetar cada unidad con el cuerpo del equipo y un juego de parlantes. Se muestra un esquema de este sistema en la Figura 1.1, en donde los procesos B y C se encargan de la fabricación del cuerpo del equipo, y el proceso E de los parlantes; los procesos A y D abastecen de material a ambas líneas de producción. El proceso R representa al repositorio en donde se depositan los productos terminados, para su inmediato empaquetamiento.

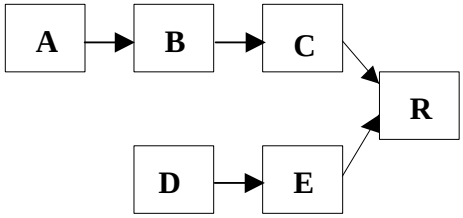


Figura 1.1 - Esquema de un sistema de fabricación de minicomponentes.

Supongamos que el proceso B tarda 5 unidades de tiempo, el C 2 unidades y el E 6 unidades para realizar sus respectivas tareas. Si los procesos A y D abastecen de material a sus siguientes procesos en los tiempos 0, 6 y 12, debido a que se desea fabricar tres equipos, se producirá la secuencia de eventos mostrada en la Tabla 1-1 en la producción de dichos equipos.

Número Evento	Tiempo	Emisor	Receptor	Pieza
1	0	A	B	Cuerpo1
2	0	D	E	Parlante 1
3	5	B	C	Cuerpo1
4	6	E	R	Parlante 1
5	6	A	B	Cuerpo2
6	6	D	E	Parlante 2
7	7	C	R	Cuerpo1
8	11	B	C	Cuerpo2

9	12	E	R	Parlante 2
10	12	A	B	Cuerpo3
11	12	D	E	Parlante 3
12	13	C	R	Cuerpo2
13	17	B	C	Cuerpo3
14	18	E	R	Parlante 3
15	19	C	R	Cuerpo3

Tabla 1-1 - Secuencia de eventos obtenidos para la producción de 3 minicomponentes para la fábrica de la Figura 1.1.

Podemos deducir de esta tabla que hay eventos dependientes de otros eventos. Por ejemplo, el evento 3 depende del evento 1, ya que para que el proceso C comience a trabajar, deberá esperar a que el proceso B finalice y le entregue el producto parcial. Se muestra en la siguiente figura la dependencia de eventos para este sistema:

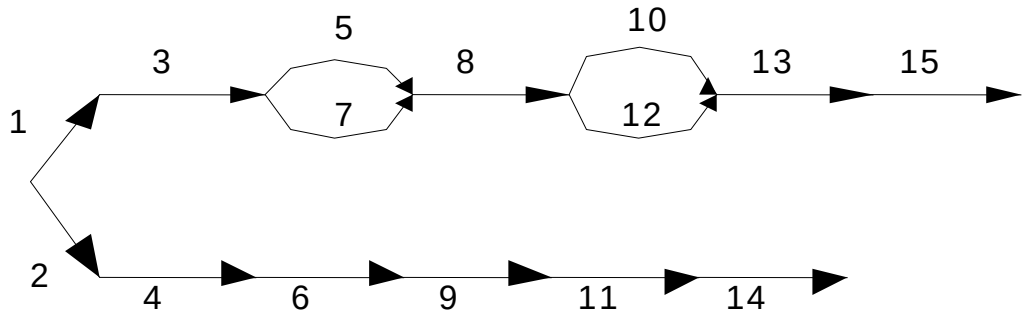


Figura 1.2 - Dependencia de eventos para la Tabla 1-1

En general, si un evento *e2* depende de un evento *e1*, la simulación de *e1* deberá realizarse antes que la de *e2*. Por otra parte, si los eventos *e1* y *e2* son independientes, es decir, no hay relación de dependencia entre ellos, entonces será posible simular los mismos en forma concurrente o en un orden arbitrario.

Una de las razones por la cual el dueño de la fábrica podría querer llevar a cabo esta simulación, es para saber qué capacidad de producción tiene la misma por hora de

trabajo. Una alternativa de ejecución de esta simulación es que simule los eventos en forma secuencial, en un único procesador. En este caso, el simulador obtendría la secuencia de eventos de la Tabla 1-1. Sin embargo, se deduce de la Figura 1.2 que la rama de eventos 2, 4, 6, 9, 11 y 14 es independiente de la que comienza con el evento 1. Esto significa que se puede analizar la fabricación de equipos y parlantes por separado y en forma paralela. Además, también hay eventos paralelos dentro de la fabricación del cuerpo del equipo, como por ejemplo los eventos 5 y 7, ó 10 y 12; los envíos de A hacia B y de C hacia R se pueden realizar en forma simultánea. Por lo tanto, si el simulador pudiera simular estas ramas de eventos en forma concurrente, mediante el uso de más de un procesador, se lograría una reducción considerable del tiempo total de simulación, por lo que se obtendría el resultado de número total de equipos fabricados por hora en un menor tiempo total de cómputo.

La práctica demuestra que las simulaciones secuenciales pueden ser inadecuadas para problemas de medianas o grandes magnitudes. Por ejemplo, un conmutador telefónico genera alrededor de cien mensajes internos para completar una llamada local. Los conmutadores telefónicos grandes pueden manejar cien o más llamadas por segundo. Por ende, una simulación de 15 minutos de tiempo real de un conmutador telefónico requerirá alrededor de 10 millones de mensajes; simularlos en un equipo monoprocesador muy rápido llevaría varias horas [Mis86].

En este punto ya se puede comprender, aunque sea en forma intuitiva, el porqué nos concentraremos a partir de aquí en el estudio y optimización de técnicas de simulación paralelas y distribuidas.

1.5. Simulación Paralela

Tal como comenzamos a esbozar en el punto anterior, se desea no sólo simular un sistema real, si no que éste se pueda ejecutar en el menor tiempo posible. Un medio obvio de obtener una simulación más rápida es dedicar mas recursos. En particular,

podemos acelerar una simulación usando un sistema multiprocesador en vez de un solo procesador. Como la mayoría de las simulaciones son de sistemas que tienen componentes independientes que operan en paralelo, parece razonable suponer que la simulación también explote el paralelismo inherente del sistema. El usar múltiples procesadores parece ser una aproximación promisoria para mejorar la velocidad, ya que se podrían asociar a cada componente del sistema, logrando así explotar el paralelismo por la independencia de eventos.

Se define aceleración al tiempo que le toma a un solo procesador hacer una simulación dividido el tiempo que toma al sistema multiprocesador hacer la misma simulación. La aceleración puede pensarse como el número efectivo de procesadores usados (la aceleración ideal con *N* procesadores es de: x/(x/N) = N, siendo *x* el tiempo total que le toma a un único procesador realizar la simulación). El problema con esta definición es especificar qué significa que un solo procesador haga la simulación. Presumiblemente queremos un procesador con las mismas capacidades que los del sistema multiprocesador, pero estos pueden no tener la suficiente memoria propia para manejar toda la simulación. Luego, el monoprocesador debe tener la misma capacidad que los procesadores en el sistema multiprocesador pero con la suficiente memoria para correr toda la simulación.

1.5.1. Sincronización en modelos distribuidos

Una aproximación para simular un modelo en forma distribuida es descomponerlo en componentes débilmente acoplados. Mientras que el sistema simulado no requiera mucha información y control global, esta aproximación parece ser promisoria, ya que tiene la habilidad de tomar ventaja del paralelismo inherente en el modelo. Sin embargo, requiere cuidado en la sincronización. Los procesos se comunican vía pasaje de mensajes, que incluyen timestamps que representan el tiempo simulado de un evento. El sistema se suele modelar como un grafo dirigido en el que los nodos representan procesos, y los enlaces representan las posibles interacciones o caminos de los mensajes.

Al distribuir los componentes del modelo, se deben proveer esquemas de sincronización. Las formas para atacar estas cuestiones dependen de si la simulación es dirigida por tiempo o por eventos, y de si la simulación es sincrónica (hay un reloj global y todos los procesos deben tener el mismo tiempo simulado) o asincrónica (cada proceso tiene su reloj local propio y el tiempo simulado para distintos procesos puede ser distinto). Cuando la simulación es manejada por tiempo, el reloj avanza de a un tick y todos los eventos planificados se simulan; cuando la simulación es manejada por eventos, el reloj avanza a la hora de procesar un mensaje simulado. Se detallan a continuación estas alternativas:

a) Simulación dirigida por tiempo

En una simulación dirigida por tiempo, el tiempo simulado avanza en incrementos fijos (ticks), y cada proceso simula su componente sobre cada tick. Los ticks deben ser cortos para garantizar precisión, lo que implica mayor duración de la simulación. La simulación puede ser sincrónica o asincrónica. Si es sincrónica, todos los procesos deben terminar de simular un tick antes que cualquiera pueda comenzar a simular al siguiente. Por ende, luego de simular hay una fase de actualización de estado y comunicación. Cuando la simulación es asincrónica, un proceso puede empezar a simular el siguiente tick tan pronto como sus predecesores hayan terminado de simular el último tick, y se sincronizan enviando mensajes a sus sucesores. Para implementar un reloj global, se puede usar una aproximación centralizada (un proceso dedicado para actuar como sincronizador) ó una de forma distribuida (con un algoritmo de broadcast adecuado).

La simulación asincrónica permite mayor concurrencia, pero a un mayor costo de comunicación (como un procesador no puede simular el siguiente tick hasta que sepa que sus predecesores terminaron de simular el último tick, debe recibir un mensaje de cada uno de ellos para cada tick). En cambio, si la simulación es sincrónica, una vez que el reloj global se sincroniza, sólo se precisa transmitir los mensajes señalando

interacciones o actualizaciones de estados. Luego, si los estados cambian menos frecuentemente que cada tick, un reloj global puede ser más eficiente.
La simulación dirigida por tiempo parece ser menos eficiente que la manejada por eventos ya que puede haber ticks durante los cuales no haya eventos que deban simularse. Sin embargo, la simulación manejada por tiempo evita el overhead extra de sincronización necesario. Luego, puede ser adecuada para sistemas con topología dinámica, y para sistemas en los cuales pasan muchas cosas al mismo tiempo.

b) Simulación dirigida por eventos

En una simulación dirigida por eventos, el tiempo de un proceso se incrementa con el de los eventos que procesa (un evento representa un cambio en el estado). Este tipo de avance en la simulación, en donde el tiempo de un proceso se incrementa de "a saltos", en forma discreta, nos lleva a deducir que puede tener mayor aceleración potencial que los que avanzan por tiempo. Estas simulaciones también pueden ser:

- **Sincrónicas**: el reloj global se pone en el mínimo tiempo del próximo evento para todos los procesos. Puede haber un único proceso dedicado a sincronizar, o estar distribuido. Hay diversos métodos para mantener el reloj global.
- **Asincrónicas**: el reloj local de cada proceso se pone en el mínimo tiempo de evento para ese proceso. Tiene alto desempeño potencial, ya que los procesos pasan menos tiempo esperando a otros, y los eventos independientes pueden ser simulados simultáneamente, aunque ocurran en distinto tiempo simulado.

Esta última aproximación ha sido muy difundida, debido a que parece ser aquella en la que pueden obtenerse mayores grados de aceleración. Se denomina a la misma ***simulación paralela de eventos discretos*** (Parallel Discrete Event Simulation - PDES). Estudiaremos con detalle en el próximo capítulo los diversos protocolos de sincronización para PDES que se han desarrollado.

1.6. Simulación de un Sistema

Consideremos el problema de simular sistemas físicos (también llamados *redes* (*networks*)), los cuales consisten de uno o más procesos físicos. Cada proceso físico opera en forma autónoma, excepto para interactuar con otros procesos físicos del sistema. La interacción se realiza a través de mensajes. Es posible modelar a muchísimos sistemas reales en términos de procesos y mensajes.

Se describe a continuación los pasos típicos que se deben realizar para la construcción y el uso de un programa de simulación:

1. Se estudian y entienden las características de un sistema real.
2. Se construye un modelo del sistema real en donde se mantienen los aspectos relevantes a simular y se descartan todos los aspectos irrelevantes.
3. Se construye una simulación del modelo anterior, que pueda ser ejecutada en un sistema computador.
4. Se analizan los resultados de la simulación para entender y predecir el comportamiento del sistema real.

Se suele verificar y refinar al modelo y a la simulación mediante la revisión en forma iterativa de los pasos (2) y (3). A partir del próximo capítulo, nuestro trabajo se centrará en el estudio de la etapa (3), la cual involucra la construcción y ejecución de un modelo a simular.

Otro de los temas a atacar es cómo pasar del modelo descrito en el paso (2), al cual denominamos sistema físico, al paso (3), de forma tal que se pueda realizar la simulación en un sistema distribuido, por lo que será ejecutada en forma concurrente en varios equipos. Se detalla a continuación las técnicas usadas para tal fin.

1.6.1. Sistemas Lógico y Físico

Un sistema físico consiste de un número finito de procesos físicos. Cada proceso físico representa un componente del sistema real que se va a simular. Por ejemplo, en un sistema de computación, se pueden considerar como procesos físicos a la CPU, a cada

unidad de disco, cada banco de memoria y a las terminales. Se describe al estado de cada proceso físico mediante un conjunto de eventos. Cada evento tiene asociado un tiempo de ocurrencia. Se construye al simulador como un conjunto de *n* procesos lógicos, P_0, P_1,..., P_{n-1}, uno por cada proceso físico del modelo. Todas las interacciones entre procesos físicos se modelan con mensajes enviados entre los procesos lógicos correspondientes. Decimos que un sistema lógico simula correctamente a un sistema físico si el primero puede predecir la secuencia exacta de mensajes que ocurre en el sistema físico.

En una simulación, cada proceso lógico extraerá el evento con menor timestamp de la lista de entrada y lo procesará. Con este paradigma de ejecución, es crucial que siempre se seleccione el evento con menor timestamp. Si uno eligiera el mensaje con un timestamp mayor, podría modificar variables de estado usadas posteriormente por un evento de menor timestamp, provocando así errores de causalidad. En otras palabras, deben satisfacerse ciertas restricciones de secuencia para que los cálculos sean correctos. Para asegurar que no haya errores de causalidad se debe cumplir la restricción de causalidad local: una simulación de eventos discretos consistente de procesos lógicos obedece a la restricción de causalidad local si y sólo si cada proceso lógico procesa eventos en orden no decreciente de timestamps. El adherir a esta restricción es suficiente aunque no necesario para garantizar que no haya errores de causalidad. Puede no ser necesario, porque dos eventos dentro de un mismo proceso lógico pueden ser independientes, en cuyo caso procesarlos fuera de secuencia no provocaría errores de causalidad.

Una simulación distribuida está compuesta por un número finito de procesos lógicos y canales, los cuales se encargan de comunicar a un par de procesos. Cada proceso lógico tiene la habilidad de ejecutar eventos en forma secuencial, y puede invocar a dos comandos especiales: *enviar y recibir*. En un envío, el proceso lógico selecciona el canal de salida y el mensaje a enviar, tras lo cual deposita el mensaje en dicho canal. Los mensajes tardan un tiempo arbitrario, pero finito, en alcanzar su destino. Todos los

mensajes enviados a través de un mismo canal son distribuidos en el mismo orden en que fueron depositados para su envío. En la ejecución del comando de recepción, el proceso lógico menciona uno o más canales de entrada de los que desea recibir un mensaje. Es posible que el o los canales indicados estén vacíos, por lo que el proceso lógico deberá bloquearse a la espera de que arribe algún mensaje por alguno de estos canales.

Para simular un sistema físico dado, se construye al sistema lógico distribuido de la siguiente manera. Se asocia un proceso lógico (PL) con cada proceso físico (PF); el proceso lógico *i* simulará las acciones del proceso físico *i*. Si el proceso físico *i* le envía mensajes al proceso físico *j*, entonces habrá un canal entre el PL_i y el PL_j.

1.6.2. Abrazo mortal (deadlock) en un sistema distribuido

Un conjunto de procesos lógicos *D* está en un abrazo mortal (deadlock) en un determinado momento si se cumplen las siguientes condiciones:

1. Cada proceso lógico de *D* está esperando recibir un mensaje o ya ha terminado.
2. Al menos un proceso lógico en D está esperando recibir un mensaje.
3. Para cualquier proceso lógico *i* en D que esté esperando recibir un mensaje de algún proceso lógico *j*, el PL_j también estará en *D*, y no habrá mensajes en tránsito desde PL_j hasta PL_i.

Se deduce luego que ningún proceso lógico de D realizará cómputo alguno, dado que estará esperando algún mensaje de un proceso lógico del mismo conjunto.

En el próximo capítulo comenzaremos a explorar los diversos protocolos de simulación paralela de eventos discretos.

2. Estado del Arte: Protocolos de simulación de eventos discretos distribuidos

2.1. Protocolo "conservador" de simulación de eventos discretos distribuido

2.1.1. Introducción

Por definición, los protocolos de simulación conservadores evitan la posibilidad de que ocurran errores de causalidad. Logran esto mediante el uso de una estrategia que les permite procesar un evento sólo cuando es seguro hacerlo. Esto significa que un proceso lógico procesará un evento sólo cuando esta acción no pueda generar un posterior error de causalidad, es decir, cuando sabe que no recibirá un mensaje con menor timestamp al del evento que va a procesar. Veremos a partir de la próxima sección el protocolo conservador de simulación de eventos discretos distribuido definido originalmente por Misra [Mis86].

2.1.2. Definición del protocolo

Consideremos como ejemplo un típico servidor FCFS (First Come - First Served, primero en entrar, primero en ser servido), al cual le lleva diez unidades de tiempo en servir a un determinado trabajo. Asumimos que un trabajo arriba en el tiempo *t* y que el servidor está ocioso. Teniendo disponible toda la información sobre los mensajes de entrada hasta el tiempo t, podemos predecir el comportamiento del servidor hasta el tiempo *t*+10: no producirá salida alguna entre el tiempo *t* y el tiempo *t*+10, y producirá un mensaje de salida en el tiempo *t*+10, donde enviará el trabajo que acaba de servir a su siguiente destino. A partir de estas observaciones, se construye un algoritmo conservador para simulaciones distribuidas.

Para que la simulación sea correcta, debe cumplir que los timestamps de los mensajes que envíe un proceso lógico sean iguales a los timestamps que tendrán los mensajes

del sistema real cuando el proceso físico efectúe sus envíos. De aquí se desprende que, si el PF_i le envía un mensaje al PF_j en el tiempo *t*, entonces el PL_i deberá enviar al PL_j el mensaje (*t*,*m*) en algún momento de la simulación, donde *m* es el contenido del mensaje.

Existe una restricción de cronología: si un PL envía una secuencia de mensajes (..., (t_i,m_i),(t_{i+1},m_{i+1}), ...) a otro PL, se requerirá que $t_i<t_{i+1}$. Este requerimiento significa que, si el *PLj* recibe un mensaje (*t,m*) del PL_i, entonces conocerá todos los mensajes que el PF_i le habrá enviado al *PFj* en el sistema real hasta el tiempo *t* inclusive, dado que cualquier nuevo mensaje que reciba del PF_i tendrá un timestamp *r*, donde *t<r*.

Se define como valor de reloj de un canal a la componente *t* del último mensaje recibido a través de dicho canal; el valor de reloj de un canal es 0 si todavía no se ha recibido ningún mensaje a través del mismo.

Con estas definiciones, existe la seguridad de que los procesos lógicos conocerán todos los mensajes recibidos por su correspondiente proceso físico hasta el tiempo $T_i=\min\{t_j\}$ ($1<=j<=n$), donde t_j son los valores de reloj de los *n* canales entrantes a dicho proceso lógico (es decir, el PL conocerá con certeza todos los mensajes que recibió el PF que modela hasta el tiempo correspondiente al mínimo valor de sus canales de entrada). Si llamamos T_i al valor de reloj del PL_i, se puede asegurar que el PL_i simuló en forma segura hasta el tiempo T_i; esto se debe a que el PL_i puede deducir todos los mensajes que el que PF_i procesará hasta el tiempo T_i.

Se asume que todos los mensajes enviados en el sistema real tienen un timestamp *t* > 0. En base a todas las definiciones brindadas hasta aquí, se muestra en el Algoritmo 1 el pseudo código de una simulación paralela de eventos discretos conservador, el cual es ejecutado en cada proceso lógico de la simulación [Mis86].

```
1. T_i = 0 /* PL_i conoce todos los mensajes recibidos por el PF_i hasta
              T_i */
2. Mientras no se cumple el criterio de finalización de la
   simulación
   /* simular PF_i hasta T_i haciendo lo siguiente */
      Para cada canal de salida enviar los mensajes ((t_1,m_1), (t_2,m_2),
       ..., (t_r,m_r)) (con t_1 < t_2 < ... < t_r) aún no enviados tal que
       el proceso físico i envía, en el sistema real, el mensaje m_j
      en el tiempo t_j a través del canal correspondiente. Puede haber
      una pareja (t_k,m_k) vacía si no se envía mensaje alguno por un
      canal de salida.
3.    Enviar cada mensaje en secuencia a través del canal apropiado
      /* Es posible deducir todos los mensajes enviados por PF_i hasta
         el tiempo T_i, e incluso algunos con timestamp mayor a T_i. Se
         envían sólo los nuevos mensajes que todavía no han sido
         enviados */
4.    T_i' = T_i
5.    Mientras T_i'= T_i
6.       Bloquearse hasta recibir mensajes en todos los canales de
         entrada
7.       Calcular T_i como el mínimo tiempo de los relojes de los
         canales de entrada; sacar al mensaje con este tiempo del
         canal correspondiente, asignar al reloj del PL_i este valor
         y procesar el evento
8.    Fin Mientras
9. Fin Mientras
```

Algoritmo 1 - Simulación Paralela de Eventos Discretos Conservador (para cada proceso lógico)

2.1.3. Ocurrencia de abrazo mortal (deadlock)

El algoritmo conservador básico de simulación distribuida tiene el inconveniente de que no puede garantizar, por sí sólo, la ejecución correcta de una simulación. Esto significa que no puede garantizar la simulación lógica de todos los eventos que suceden en el sistema físico. Lo dicho se debe a que es posible la ocurrencia de abrazo mortal (deadlock) en el sistema lógico simulado. Estudiaremos el problema en esta sección y veremos a partir de las siguientes las posibles soluciones.

Analicemos el siguiente ejemplo: se muestra en la Figura 2.1 el esquema de un sistema de lavado de automóviles. El mismo está compuesto por la recepción, dos máquinas de

lavado ML1 y ML2, la etapa de Lustrado manual y la salida. Los autos arriban a la recepción y de aquí se los envía hacía ML1 o ML2 de acuerdo a las siguientes reglas:

- Si ambas máquinas están ocupadas, se encola a los autos en la recepción.
- Si ambas máquinas están desocupadas, se envía al primer automóvil de la cola ubicada en la recepción a ML1.
- Si sólo una máquina está desocupada, se envía el primer auto de recepción a dicha lavadora.
- Si se está lustrando un automóvil y hay otro que acaba de ser lavado, éste deberá esperar en la máquina lavadora hasta que la etapa de Lustrado quede libre.

Supongamos que ML1 tarda 8 unidades de tiempo y ML2 10 unidades en el lavado de un automóvil, y que se tardan 5 unidades en la etapa de Lustrado manual. Se desea simular este sistema físico para saber, en base a una distribución dada de arribo de autos, el tiempo promedio que pasa un automóvil dentro del sistema. En la simulación tendremos cinco procesos lógicos, asociados a los cinco procesos físicos del sistema real. El sistema lógico tendrá los siguientes canales de comunicación: (Recepción, ML1), (Recepción, ML2), (ML1, Lustrado), (ML2, Lustrado) y (Lustrado, Salida).

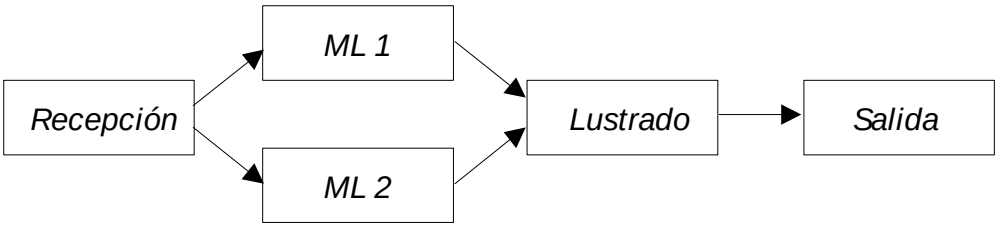


Figura 2.1 - Esquema de una lavadora de autos

Supongamos que se estudia al sistema físico durante un lapso de tiempo tal que ingresan 4 automóviles, en tiempos de 5, 20, 40 y 55. Se muestra en la Tabla 2-1 la secuencia de eventos producida.

Número Evento	Tiempo	Emisor	Receptor	Mensaje
1	5	Recepción	ML1	Auto1
2	13	ML1	Lustrado	Auto1
3	18	Lustrado	Salida	Auto1
4	20	Recepción	ML1	Auto2
5	28	ML1	Lustrado	Auto2
6	33	Lustrado	Salida	Auto2
7	40	Recepción	ML1	Auto3
8	48	ML1	Lustrado	Auto3
9	53	Lustrado	Salida	Auto3
10	55	Recepción	ML1	Auto4
11	63	ML1	Lustrado	Auto4
12	68	Lustrado	Salida	Auto4

Tabla 2-1 - Secuencia de eventos obtenida en el lavado de 4 automóviles que arriban en tiempos de 5, 20, 40 y 55.

Debido a estos tiempos de arribo de automóviles, será ML1 la máquina que siempre los atenderá para el lavado. Cuando se realice la simulación de este sistema bajo el protocolo PDES conservador, se observará que el proceso lógico correspondiente al Lustrado nunca recibirá un mensaje desde el proceso lógico ML2. Por lo tanto, el valor de reloj del canal (ML2, Lustrado) permanecerá siempre en 0.

Bajo estas condiciones, el PL de Lustrado no podrá avanzar en la simulación. En el tiempo 13, tendrá un mensaje en el canal que lo une con ML1, pero el canal que lo une con ML2 permanecerá vacío, tal como se describió anteriormente. En este punto, el proceso lógico no sabrá si es seguro procesar este mensaje, ya que no está en condiciones de saber si por el otro canal le llegará un mensaje con un timestamp menor a 13. Esto se debe a que el proceso lógico dispone sólo de información local, y no de información global del sistema, de donde podría deducir que no recibirá un automóvil

por el canal que lo une con ML2 antes del tiempo 13. En consecuencia, el protocolo conservador descrito en el Algoritmo 1 se detendrá en el paso 6, y quedará bloqueado indefinidamente. Por lo tanto, el sistema entrará en deadlock, y la simulación no podrá seguir avanzando en su ejecución.

Se describe a continuación otro ejemplo de deadlock, pero esta vez debido a una espera circular de procesos lógicos. En la Figura 2.2 se muestra el esquema de un sistema con tres procesos lógicos. Se incluye un número entero junto a cada canal, el cual indica el valor de reloj de ese canal. Se puede observar que el último mensaje enviado desde x hasta y fue en el tiempo 10, de y hasta z en tiempo 15, y de z a x en tiempo 18. El PL x tiene un mensaje con timestamp 25 en un canal de entrada. Ninguno de los tres procesos lógicos tienen mensajes pendientes de salida, por lo que sólo podrán mandar alguno en caso de que hayan recibido un nuevo mensaje. Podemos observar en el sistema que el proceso lógico *z* no enviará un mensaje a *x* hasta que no reciba un mensaje de *y*; de igual modo, *y* no procederá a menos que reciba un mensaje de *x*. En base a esta información global se puede deducir que *x* podría procesar el siguiente evento de entrada, ya que dado el estado actual no recibirá mensaje alguno de *z* menor a 25. Sin embargo, se produce abrazo mortal, ya que *x* no puede saber si del canal (*z*,*x*) puede llegar algún mensaje con timestamp mayor a 18 (que es el timestamp del último mensaje que recibió por ese canal) pero menor a 25. Si *x* tuviera información global del sistema, sabría que esto no es posible (por lo explicado anteriormente). De este modo el sistema queda en un estado de abrazo mortal.

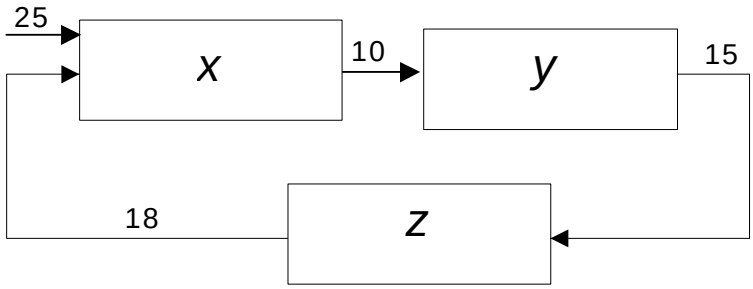


Figura 2.2 - Simulación PDES conservadora que entra en deadlock circular

2.1.4. Resolución de deadlock

Hemos visto que el protocolo conservador básico de simulación distribuida puede ocasionar deadlocks, incluso en sistemas que no poseen ciclos. Estudiaremos a partir de aquí un mecanismo sencillo que soluciona este problema.

En todos los ejemplos vistos, el reloj del simulador (es decir, el valor mínimo de los relojes de los procesos lógicos) se mantiene constante en un valor *t* en forma indefinida. Si *t* es menor que el tiempo hasta el cual se precisa realizar la simulación, el sistema entró en deadlock, por lo que se precisará aplicar alguna técnica para romper el abrazo mortal y permitir que la simulación avance. Por ejemplo, en el esquema de la Figura 2.2 se deberá avisar al PL x que no recibirá mensaje alguno por el canal (z,x) hasta que x envíe primero un mensaje de salida hacia y.

Comenzamos a describir la modificación del protocolo básico PDES conservador que evita deadlocks. Se define para tal fin un nuevo tipo de mensaje a usar en una simulación, llamado mensaje nulo. Se nota al mismo como el par (*t*, *null*), donde *t* es el timestamp del mensaje, y *null* el contenido del mismo. Cuando el PL_i envía al PL_j un mensaje de este tipo, le estará indicando que en el sistema real el PF_i no le enviará mensaje alguno al PF_j entre el tiempo actual del canal (i,j) y el tiempo *t*. Por lo tanto, cualquier mensaje que envíe el PL_i al PL_j tendrá indefectiblemente un timestamp mayor a *t*. En resumen, un mensaje nulo del PL_i al PL_j es una promesa de que no se enviará

un mensaje con timestamp menor al del mensaje nulo por el canal (i,j). Un proceso lógico actúa de igual modo ante la recepción de un mensaje "común" y uno nulo: actualiza el valor de su reloj interno con el del timestamp del mensaje, y podrá enviar en consecuencia nuevos mensajes de salida.

El reloj de cada enlace de entrada provee un límite inferior en el timestamp del siguiente evento sin procesar. Este límite de entrada puede usarse para determinar un límite inferior en el timestamp del siguiente mensaje de salida. Entonces, se modifica el algoritmo para que cuando un proceso termina de procesar un evento, envíe un mensaje nulo a cada uno de sus canales de salida indicando el límite. De este modo el receptor del mensaje nulo puede calcular nuevos límites en sus enlaces de salida, enviar esta información a sus vecinos, etc. De esta forma se rompe el ciclo de procesos bloqueados y se evita el abrazo mortal.

Sobre la base de estas premisas, se modifica ligeramente Algoritmo 1, con el fin de evitar la ocurrencia de abrazo mortal. La modificación se basa en que cada vez que el proceso lógico *i* lee un mensaje de un canal de entrada con timestamp T_i (con lo cual actualiza su reloj interno con ese valor y procesa el evento), podrá predecir si enviará mensajes por sus canales de salida y el timestamp de los mismos. Entonces, si el PL sabe que no enviará por el canal (*i*,*j*) un mensaje hasta al menos el tiempo t_j, podrá informar al PL_j de esta situación enviándole por dicho canal de salida el mensaje (t_j, *null*). De esta forma el receptor del mensaje nulo puede calcular nuevos límites en sus enlaces de salida, enviar esta información a sus vecinos, etc. Se rompe así la cadena de procesos bloqueados y se evita el abrazo mortal.

2.1.5. Precálculo de tiempos de servicio

Se define como precálculo (*lookahead*) a la habilidad de un proceso lógico de poder predecir qué pasará (o, más importante, qué NO ocurrirá) en el futuro simulado. Si un proceso lógico *i* en tiempo simulado T_i puede predecir todos los eventos que generará hasta el tiempo T_i+L, se dice que el proceso tiene precálculo L. Podrá lograr lo anterior sabiendo cual es el tiempo mínimo de procesamiento que le lleva ejecutar cada evento.

Por ejemplo, si el PL tarda un tiempo de 20 unidades como mínimo para procesar un evento, sabrá que hasta un tiempo de T_i + 20 (siendo T_i el valor de su reloj local) no enviará mensajes por sus canales de salida. El precálculo mejora la habilidad de predecir eventos futuros, que pueden usarse para determinar qué otros eventos son seguros de procesar.

Ya vimos en la sección anterior cómo se mejora el algoritmo de simulación paralela de eventos discretos conservador con el uso de eventos nulos. La solución final que se propone para evitar deadlocks es similar a ésta, pero cada proceso lógico envía por sus canales de salida, tras ejecutar un evento, un mensaje nulo que contenga como timestamp el tiempo $T + L$, donde T es el valor actual de su reloj, y L su precálculo. De este modo, al usar el precálculo de cada proceso, se logra acelerar aún más la simulación, ya que se informa con mayor precisión a los procesos lógicos destino sobre los tiempos que tendrán los mensajes que recibirán por sus canales de entrada.

Aplicamos este nuevo algoritmo en el ejemplo de la Figura 2.1, por lo que se obtiene en el sistema lógico simulado la secuencia de eventos mostrada en la Tabla 2-2. Recordamos que en este sistema la ML1 tarda 8 unidades de tiempo en lavar un auto, ML2 10 unidades de tiempo, y se tardan 5 unidades en el Lustrado; los automóviles ingresan en los tiempos 5, 20, 40 y 55.

Con el algoritmo original la simulación se bloqueaba en tiempo 13, ya que el proceso lógico de Lustrado no sabía nada acerca de la posibilidad de recibir eventos por el canal (*ML2*, *Lustrado*). Con la modificación propuesta, la etapa de Lustrado estará recibiendo un mensaje nulo de timestamp 15 por ese canal, por lo cual tendrá todas sus colas de entrada llenas, y podrá procesar al mensaje de aquella cola con el menor timestamp (en este caso el mensaje de timestamp 13, que produce el evento 3). La misma situación ocurre en los eventos 9 y 10, 15 y 16, 21 y 22, con lo que se evita la ocurrencia de deadlock.

Número Evento	Tiempo	Emisor	Receptor	Mensaje

1	5	Recepción	ML1	Auto1
2	5	Recepción	ML2	*null*
3	13	ML1	Lustrado	Auto1
4	15	ML2	Lustrado	*null*
5	18	Lustrado	Salida	Auto1
6	20	Lustrado	Salida	*null*
7	20	Recepción	ML1	Auto2
8	20	Recepción	ML2	*null*
9	28	ML1	Lustrado	Auto2
10	30	ML2	Lustrado	*null*
11	33	Lustrado	Salida	Auto2
12	40	Lustrado	Salida	*null*
13	40	Recepción	ML1	Auto3
14	40	Recepción	ML2	*null*
15	48	ML1	Lustrado	Auto3
16	50	ML2	Lustrado	*null*
17	53	Lustrado	Salida	Auto3
18	55	Recepción	ML1	*Auto4*
19	55	Recepción	ML2	*null*
20	60	Lustrado	Salida	*null*
21	63	ML1	Lustrado	Auto4
22	65	ML2	Lustrado	*null*
23	68	Lustrado	Salida	Auto4
24	70	Lustrado	Salida	null

Tabla 2-2 - Secuencia de eventos obtenida en la simulación del sistema lógico de la Figura 2.1 bajo el protocolo PDES conservador, con la inclusión de eventos nulos basados en el precálculo de cada proceso lógico.

2.1.6. Relación entre los sistemas físicos y lógicos con mensajes nulos

Mediante la utilización de mensajes nulos, se logra que una simulación paralela de eventos discretos que utiliza un protocolo conservador no entre nunca en deadlock. Si

el sistema físico entrara en deadlock, el simulador continuaría su ejecución mediante la transmisión de mensajes nulos con timestamp cada vez mayores. Esta simulación sería correcta respecto al sistema físico, ya que a medida que el tiempo progresa, no se transmite mensaje alguno en el sistema físico hasta un tiempo *t*, y ocurriría lo propio en el sistema simulado hasta el tiempo *t*.

2.2. Protocolo optimista de simulación de eventos discretos distribuido

2.2.1. Introducción

Los métodos optimistas no evitan errores de causalidad sino que los detectan y recuperan. En contraste con los mecanismos conservadores, no precisan determinar cuándo es seguro proceder, sino que detectan un error e invocan un procedimiento de recuperación, explotando paralelismo donde pueden ocurrir errores de causalidad pero de hecho no ocurren. El mecanismo Time Warp es el protocolo optimista más conocido [Jef85].

A partir de la siguiente sección comenzamos a analizar la teoría del tiempo virtual, la cual nos servirá de base para comprender luego al protocolo de Time Warp.

2.2.1. Tiempo Virtual

Un sistema de tiempo virtual es un sistema distribuido que ejecuta coordinadamente, y posee un reloj virtual imaginario que avanza de a ticks de tiempo virtual. Se utiliza el concepto de tiempo virtual para medir el progreso computacional de una computación distribuida.

El tiempo virtual puede o no tener conexión con el tiempo real. Se asume que el tiempo virtual adopta valores reales positivos (con el agregado de un valor infinito positivo *+inf*), y que éstos están totalmente ordenados a través de la relación <.

Desde el punto de vista lógico, el reloj global de un sistema distribuido siempre avanza en forma creciente (o al menos, nunca retrocede) con un ritmo independiente respecto al tiempo real. Sin embargo, desde el punto de vista de la implementación del sistema, existen muchos relojes virtuales locales (uno por cada proceso lógico), los cuales están débilmente sincronizados, y a pesar de que tienden a avanzar hacia mayores tiempos virtuales, ocasionalmente pueden llegar a retroceder.

Los procesos de un sistema distribuido se comunican a través del pasaje de mensajes. No existe el concepto de *canal* entre dos procesos (en contraste con el protocolo conservador), por lo que no existe la necesidad de definir un protocolo para el establecimiento de una comunicación.
Se almacenan cuatro valores en cada mensaje: el nombre del emisor, el tiempo virtual de envío, el nombre del receptor, y el tiempo virtual de recepción. El tiempo virtual de envío es el momento en que el mensaje es enviado. El tiempo virtual de recepción es el tiempo virtual en el que el mensaje debe ser recibido en el destino.
Los sistemas de tiempo virtual están basados en dos reglas básicas:
Regla 1: El tiempo virtual de envío de un mensaje debe ser menor que su tiempo virtual de recepción.
Regla 2: El tiempo virtual de un evento en un proceso debe ser menor que el tiempo virtual del siguiente evento en ese proceso.
Estas reglas implican que todos los mensajes de salida (de un proceso) son enviados en orden respecto al tiempo virtual de envío (pero no necesariamente respecto al tiempo virtual de recepción), y que todos los mensajes de entrada de un proceso son leídos en orden respecto al tiempo virtual de recepción (pero no necesariamente respecto al tiempo virtual de envío).

Veremos en la próxima sección la definición del protocolo optimista de Time Warp, tal como se lo definió originalmente en [Jef85]. En la definición del mismo, se usarán en forma indistinta los términos timestamp de un mensaje o tiempo virtual de recepción del mismo.

2.2.2. El mecanismo de Time Warp

INTRODUCCIÓN

No es deseable, por lo general, que una implementación requiera que el progreso de los procesos a través del tiempo virtual siga un ritmo similar al progreso en tiempo real,

ya que esto haría que la ejecución del sistema sea secuencial. En consecuencia, se permite que cada proceso lógico pueda avanzar en forma autónoma, por lo que en un momento dado, algunos procesos podrán estar adelantados en tiempo virtual respecto a otros que se hayan rezagado.

No es obvio ver cómo un proceso lógico puede procesar sus mensajes de entrada en orden creciente de timestamps, ya que éstos no arriban generalmente respetando dicho orden. Por ende, es imposible para un proceso saber, en base a su información local únicamente, cuando bloquearse y esperar al mensaje con el "próximo" timestamp. No es válido presumir que un determinado mensaje *m1* es realmente el "próximo", ya que siempre es posible que arribe un nuevo mensaje *m2* (enviado por un nuevo proceso) con tiempo virtual de recepción menor al de *m1*. Por lo tanto, a pesar de que el mensaje que llegue sea realmente el del "próximo" timestamp, no se podrá saber fehacientemente que el mismo cumple con esa propiedad. Este es el problema central de la implementación de tiempo virtual, pero es resuelta, como se verá más adelante, por el mecanismo de Time Warp.

Este mecanismo asume que la comunicación de mensajes es confiable, pero no se asume que los mismos arriben en igual orden al que fueron enviados; en realidad, esto sería un desperdicio, dado que generalmente no se procesa a los mensajes en idéntico orden al que fueron enviados, ya que en realidad importa, para tal fin, el tiempo virtual de recepción y no el de envío.

Time Warp está formado por dos componentes principales: el mecanismo de control local, el cual se ocupa de ejecutar los eventos en el orden correcto, y el mecanismo de control global, que se ocupa de temas globales, tales como administración de espacio, flujo de control, I/O, manejo de errores y detección de terminación. Detallaremos a partir de aquí cada uno de estos mecanismos.

EL MECANISMO DE CONTROL LOCAL

Aunque, en forma abstracta, hay un único tiempo virtual global en el sistema, no existe un reloj virtual global en la implementación del mismo. En lugar de esto, cada proceso

tiene su propio reloj virtual local. Estos relojes cambian su valor sólo en el momento de procesar un evento, y adoptan como valor actual al timestamp de dicho evento. Cada vez que un proceso envía un mensaje de salida, copia el valor de su propio reloj virtual en el tiempo virtual de envío del mensaje. Los campos referidos al receptor y al tiempo virtual de recepción son asignados por la aplicación que ejecuta en el sistema y que solicitó el envío de dicho mensaje.

Cada proceso tiene una única cola de entrada, en donde almacena a los mensajes entrantes en orden creciente respecto al tiempo virtual de recepción. La ejecución de un proceso consta, idealmente, de un simple ciclo en el cual recibe mensajes y ejecuta eventos en orden creciente de tiempo virtual. Esta ejecución ideal se desarrolla mientras no arribe un mensaje con tiempo de recepción virtual en el "pasado" del proceso. Se supone que este hecho debe ocurrir raramente, debido a la variación en las velocidades de cómputo de los distintos procesos y las demoras de transmisión en la red. Más allá de las razones del arribo tardío de un mensaje con un tiempo de recepción en el "pasado", la semántica de tiempo virtual requiere que se reciba a todos los mensajes entrantes en orden estricto de timestamps. El único modo de alcanzar esta meta en el receptor es que el mismo realice un rollback hacia un tiempo virtual anterior al timestamp del mensaje, cancele todos los efectos colaterales que esto genera, y ejecute nuevamente hacia adelante, por lo que recibirá así a este mensaje en el orden correcto.

Como quedó dicho, es imposible que un proceso espere al "próximo" mensaje, por lo que toma la decisión de ejecutar continuamente, procesando en orden creciente de timestamps a los mensajes que recibe. Sin embargo, toda su ejecución es provisional, ya que está constantemente apostando a que no arribará un mensaje con tiempo virtual de recepción menor al del mensaje que se está procesando en ese momento. A medida que gana esta apuesta, la ejecución avanza sin inconvenientes. Pero cada vez que pierda la apuesta, el proceso será castigado en performance, dado que deberá hacer un rollback al tiempo virtual en que "debería" haber estado cuando recibió el último mensaje. Esta situación es muy similar a la política que utiliza el mecanismo de

administración de memoria de un sistema operativo en la implementación de memoria virtual: está constantemente apostando a que cada referencia a memoria no causará un fallo de página; a medida que gana esta apuesta, la ejecución es eficiente, pero pasa a ser mucho más costosa cuando pierde la apuesta y debe acceder a una página residente en memoria secundaria.

Se podría describir a esta situación diciendo que cada proceso está constantemente haciendo cálculos adelantados, procesando mensajes "futuros" de su cola de entrada, y realiza esta acción siempre que los timestamps de los nuevos mensajes arriben en orden creciente.

Cada vez que un proceso haya procesado todos los mensajes de entrada, asignará un valor de $+inf$ a su propio reloj virtual, y se dirá, por convención, que el proceso ha terminado. Sin embargo, no se destruye al proceso, debido a que el arribo de un nuevo mensaje podría causar que el proceso realice un rollback al timestamp de dicho evento y vuelva a quedar en un estado de no terminación.

El nombre "Time Warp" deriva del hecho de que los relojes virtuales de los diferentes procesos no precisan estar sincronizados, y de que estos relojes avanzan y retroceden en el tiempo virtual. La ocurrencia de estos avances y retrocesos no viola la intención de que el reloj virtual global siempre "progrese hacia adelante (o al menos nunca retroceda)", ya que el mecanismo de rollback es completamente transparente al proceso que lo sufre. En conclusión, es transparente para un proceso que se encuentra en tiempo virtual t si sufrió o no algún rollback hasta ese momento, ya que el mecanismo asegura que, más allá de que haya habido o no rollbacks, la ejecución del proceso hasta el tiempo virtual t ha sido correcta.

En el caso de que haya más procesos que procesadores, sólo un subconjunto de los procesos podrá ejecutar en un momento dado. La regla de planificación natural es que siempre ejecuten aquellos procesos cuyo reloj virtual local sean de los más retrasados. En un esquema multiprocesador de memoria compartida, siempre ejecutarán los n procesos más atrasados, donde n es el número de procesadores en uso. En una red,

siempre ejecutará, en cada procesador, el proceso con menor valor de reloj que resida en dicho procesador.

Suele ser complicado el manejo de rollbacks en un ambiente distribuido: un proceso que realiza rollback puede haber enviado mensajes a otros procesos, los cuales causarán efectos en sus destinos y podrán haber provocado el envío de más mensajes a más procesos, y así sucesivamente. Los caminos seguidos por estos mensajes directos e indirectos podrían converger o formar ciclos. Sin embargo, es posible revertir el efecto de todos estos mensajes, mediante la anulación de cada envío. Se verá a continuación cómo hace el mecanismo Time Warp para realizar lo dicho en forma eficiente.

ANTIMENSAJES Y MECANISMO DE ROLLBACK

Para entender el mecanismo de rollback, pasamos a describir más en detalle la estructura de los procesos y los mensajes de un sistema distribuido. Se muestra en la Figura 2.3 la estructura de un proceso de nombre A. La representación del proceso en tiempo de ejecución está compuesta por:

1. Un nombre de proceso, que debe ser único en el sistema (A en este caso).
2. Un reloj virtual local (Local Virtual Time - LVT). En la Figura 2.3 el valor de este reloj es de 50, lo que indica que se está procesando un mensaje con tiempo de recepción virtual de 50.
3. Un estado, que en general es el espacio de datos del proceso, incluyendo su pila (stack), sus propias variables, y su contador de programa. En la figura se muestra como ejemplo un estado con dos variables numéricas.
4. Una cola de estados, que contiene las copias de los estados recientes en que estuvo el proceso. El mecanismo de Time Warp debe, en forma regular, salvar el estado de un proceso, con el fin de poder aplicar la técnica de rollback. Se asume por simplicidad que se realiza lo anterior luego del procesamiento de cada evento en cada proceso. Es posible reducir o incrementar esta frecuencia, y una buena

estrategia es la de salvar el estado del proceso cada vez que utiliza M segundos del procesador. Tal como veremos en la sección de mecanismo de control global, no es necesario retener todos los estados desde el comienzo del tiempo virtual, pero debe haber al menos un estado almacenado que sea menor que el tiempo virtual global (Global Virtual Time - GVT).

5. Una cola de entrada, que contiene todos los mensajes arribados en orden creciente de timestamp. Algunos de estos mensajes ya han sido procesados debido a que su tiempo de recepción virtual es menor que 162. Sin embargo, no son borrados de la cola porque pueden ser necesarios en el caso de rollback, en donde habría que volver a procesarlos. Los mensajes con tiempo virtual de recepción mayores a 162 no han sido procesados todavía, o pudieron haber sido procesados y haber sido anulados de su procesamiento por rollbacks un número igual de veces. Sólo se debe retener a los mensajes entrantes con tiempo virtual de envío mayor o igual que el GVT.
6. Una cola de salida, que contiene copias negativas de los mensajes que el proceso ha enviado recientemente, almacenadas por el orden de su tiempo virtual de envío. Se precisa a los mismos en caso de rollback, para poder deshacer el efecto de haberlos enviado. Al igual que en el caso anterior, se debe retener solamente a aquellos mensajes con tiempo virtual de envío mayor o igual al GVT.

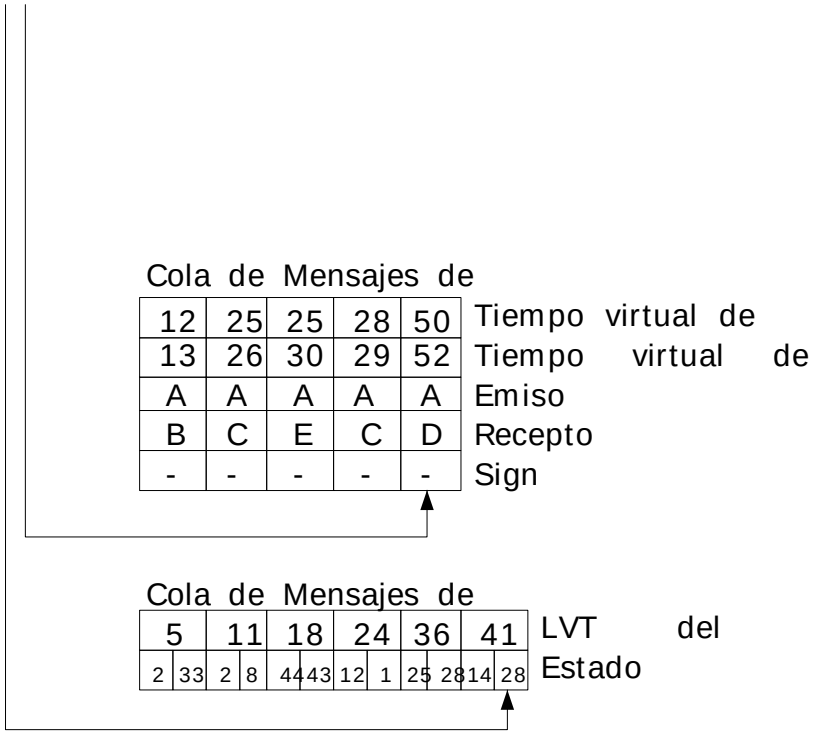


Figura 2.3 - Estructura de un proceso lógico en Time Warp

Se define un *antimensaje* como un mensaje similar a uno previamente enviado, de igual contenido a éste excepto en el signo. Dos mensajes que son idénticos excepto en que tienen signos opuestos son antimensajes del otro. Todos los mensajes enviados explícitamente por los programas de usuario tienen un signo positivo (+); sus antimensajes tienen signo negativo (-). Cada vez que un proceso envía un mensaje, lo que realmente ocurre es que se envía una copia a destino, y se almacena una copia negativa del mismo (el antimensaje) en la cola de salida, para usarlo en caso de que el proceso sufra un rollback.

Cada vez que haya un mensaje y su antimensaje en la misma cola, se aniquilarán inmediatamente entre ellos. Por ende, es posible obtener una cola más corta tras el encolado de un mensaje, ya que se podría anular con su antimensaje. No es de importancia si el mensaje que arriba por primera vez a la cola es un mensaje o un antimensaje; en el momento en que su mensaje opuesto ingrese a dicha cola, sucederá el aniquilamiento de ambos. En general, se crean los mensajes y sus antimensajes de a pares, y también se los aniquila de a pares, por lo que en todo momento la suma algebraica de todos los mensajes en un sistema Time Warp es cero.

Cuando un mensaje arriba a un proceso, se encola, por orden de timestamp, en la cola de entrada de dicho proceso. Si el timestamp del mensaje es mayor que el tiempo del reloj virtual del proceso destino, se encola luego del último mensaje procesado, y será atendido cuando el proceso haya procesado todos los anteriores a él. Sin embargo, este no será el caso si llegara un mensaje con menor tiempo de recepción. Tomemos como ejemplo al proceso A de la Figura 2.3, cuyo reloj virtual es de 50, si le llegara un mensaje con tiempo virtual de recepción 20. El mensaje entrante ingresaría en la cola de entrada en un lugar anterior al correspondiente al último mensaje procesado. Por ende, todos los cálculos realizados por el proceso desde el tiempo virtual 20 podrían ser incorrectos (no sabemos a priori si la ejecución del nuevo evento produciría un igual comportamiento en el proceso a no haberlo procesado), por lo que deben ser deshechos a través de un rollback.

El primer paso en el mecanismo de rollback consta de una simple búsqueda por la cola de estados hasta hallar al último estado almacenado antes del tiempo 20, y luego restaurarlo (en el ejemplo se debe restaurar el estado con tiempo virtual 18). También se restaura el valor del reloj virtual del proceso A, asignando al mismo un valor de 20 (ya que se le asigna el timestamp del evento que se procesa). Luego de esto, es posible descartar de la cola de estados a todos aquellos grabados luego del tiempo 20 y comenzar la ejecución de A nuevamente hacia adelante.

Sin embargo, restan corregir todavía los efectos producidos por los mensajes que A envió a otros procesos entre el tiempo 20 y 50. La forma que utiliza Time Warp para

deshacer el efecto de un mensaje enviado es transmitir su antimensaje. En nuestro ejemplo, A transmite los antimensajes a los procesos destino de todos los mensajes de la cola de salida con tiempo virtual de envío entre 20 y 50, por lo que no queda registro alguno en A de que estos mensajes alguna vez hayan existido (en el ejemplo, se transmiten todos los antimensajes de la cola de salida menos el primero, que tiene un timestamp de envío de 12). Es importante notar que no se borraron los mensajes de entrada una vez que fueron procesados, ya que al suceder un rollback deberán ser tratados nuevamente, como si nunca se los hubiera atendido.

En este momento, el proceso A y todas sus colas estarán en el estado en que hubieran estado si el mensaje con timestamp 20 hubiera llegado en el orden correcto. Los antimensajes enviados provocarán un rollback en cada uno de los procesos destino, si su tiempo virtual de recepción es menor que el tiempo virtual del receptor. Es importante notar que ocurre lo mismo si en lugar de un antimensaje el que arriba con un timestamp menor es un mensaje.

Dependiendo del tiempo virtual en que se encontraba el receptor, es posible la ocurrencia de varias secuencias de eventos, las cuales terminarán obteniendo que el receptor se halle en el estado que habría alcanzado si no hubiera recibido el o los mensajes enviados por el proceso A luego del tiempo 20. En resumen, todas las posibilidades de acción respecto al envío de mensajes del proceso A a diversos procesos destinos son:

1. Si el mensaje original (positivo) enviado por A ha arribado a destino, pero todavía no ha sido procesado, indicará que su tiempo virtual de recepción es mayor que el del reloj virtual del receptor. Al llegar el antimensaje correspondiente, será también encolado en la cola de entrada, ya que su timestamp también será mayor que el tiempo virtual del receptor. En este caso, al existir un mensaje y su antimensaje en la misma cola, se produce un aniquilamiento automático de ambos. Esto implica que el receptor no realiza ningún rollback, y ejecuta como si nunca hubiera llegado el mensaje original desde el proceso A, que es exactamente lo que se deseaba.

2. Si el mensaje original (positivo) ya fue recibido en el destino, y posee un tiempo virtual de recepción que está en el presente o en el pasado con respecto al **actual** valor del reloj virtual del receptor, este proceso pudo haber procesado, parcial o totalmente, a este mensaje, e incluso a otros con igual o mayor timestamp que el mismo. Bajo estas circunstancias, el antimensaje también arribará en el pasado respecto al tiempo virtual del receptor, por lo que le causará un rollback hasta el tiempo anterior a recibir el mensaje positivo respectivo. En este momento, el proceso seguirá ejecutando hacia adelante como si nunca hubiera recibido el mensaje original. En el caso de que el proceso receptor hubiera enviado mensajes a otros procesos, deberá enviar a los mismos los antimensajes respectivos.
3. Podría ocurrir también que un antimensaje arribe antes a destino que el propio mensaje (ya que no se asume preservación del orden de envío en el medio de comunicación). En este caso, se encolará al antimensaje del modo usual, y será eventualmente aniquilado cuando el mensaje respectivo arribe al destino. Si el proceso debe procesar al antimensaje antes de que su mensaje positivo haya arribado, podrá tomar cualquier decisión (por ejemplo, no realizar operación alguna). El método más óptimo será el de ignorar a los mensajes de signo negativo de la cola de entrada, y cuando lleguen sus respectivos mensajes positivos, se procederá a aniquilar a ambos, sin que ocurra rollback alguno.

El protocolo de antimensajes es extremadamente robusto, y trabaja correctamente en todas las circunstancias posibles. El nivel de indirección puede ser de cualquier profundidad, e incluso pueden hallarse ciclos en el grafo realizado por los antimensajes circulantes, sin que esto cause ejecuciones erróneas. El proceso de rollback no debe ser atómico, y podrían interactuar varios rollbacks en forma simultánea sin ninguna sincronización entre ellos. No hay posibilidad de abrazo mortal (deadlock), ya que en ningún momento hay bloqueos de procesos. Tampoco existe la posibilidad de "efecto dominó" (es decir, una cascada de rollbacks hacia el pasado), dado que el peor caso es que todos los procesos del sistema realicen rollback al mismo tiempo virtual al que

debió volver el proceso original que comenzó el rollback, pero luego todos ejecutarán nuevamente hacia adelante.

El Mecanismo de Control Global

El concepto principal sobre el que se basa el mecanismo control global es el del Tiempo Virtual Global (Global Virtual Time - GVT). Se define al tiempo virtual global de la siguiente manera: El GVT en el tiempo real r es el mínimo de

(1) todos los tiempos virtuales de los relojes virtuales locales para el tiempo r, y

(2) del tiempo de envío virtual de todos los mensajes que han sido enviados pero todavía no han sido procesados en el tiempo r.

Dentro de los mensajes que no han sido procesados se incluyen a aquellos que están en tránsito o en el futuro de alguna cola de entrada de un proceso. Es fácilmente demostrable por inducción en el número de actos de comunicación de mensajes (envíos, arribos y recepciones) que el GVT nunca decrece, a pesar de que los relojes virtuales locales realizan rollback en forma frecuente. El GVT sirve como cota inferior para los tiempos virtuales de un proceso, por lo que no podrá realizar rollback más allá de ellos.

Estas propiedades hacen que sea apropiado considerar al GVT como el reloj virtual de todo el sistema, y utilizarlo como medida de progreso del mismo. Por lo tanto puede verse al GVT como una línea base para la realización de "commits": cualquier evento con tiempo virtual menor al GVT no podrá realizar rollback, por lo que puede ser procesado con seguridad (es decir, se podría hacer un commit tras la ejecución de este evento, ya que el mismo no provocará rollbacks).

No es sencillo implementar la definición del GVT en términos de un momento instantáneo en un sistema distribuido con mensajes en tránsito. Por lo general es imposible para el mecanismo de Time Warp conocer en el tiempo real *r* el valor exacto del GVT. Por supuesto, los procesos de usuario no necesitan acceder al GVT, pero el mecanismo de Time Warp lo utiliza para el control global del sistema. Entonces, se caracteriza al GVT en forma más práctica, como el valor menor o igual al mínimo de:

a) Todos los tiempos virtuales de todos los relojes virtuales en un determinado momento,
b) todos los tiempos virtuales de envío de mensajes que han sido enviados pero todavía no produjeron acuses de recibo (y pueden por lo tanto estar en tránsito en ese momento), y
c) todos los tiempos virtuales de envío de mensajes de las colas de entrada que no han sido procesados por el proceso receptor.
Esta caracterización nos permite obtener un algoritmo de estimación del GVT rápido y distribuido, de orden O(*d*), donde *d* es la demora requerida para realizar un "broadcast" a todos los procesadores del sistema. Este algoritmo ejecuta concurrentemente con el de simulación principal y retorna un valor que está entre el verdadero GVT al momento en el cual el algoritmo comenzó y el verdadero GVT al momento en que este finalizó. En consecuencia, el algoritmo da un valor "apenas vencido" del GVT, el cual es lo mejor que se puede hacer sin tener que sincronizar todo el sistema.
Durante la ejecución de un sistema de tiempo virtual, el mecanismo Time Warp debe estimar el GVT en forma bastante frecuente. La frecuencia a adoptar es una medida de compromiso: una alta frecuencia produce un tiempo de respuesta más rápido y una mejor utilización del espacio (debido al almacenamiento más frecuente de estados), pero también usa mayor ancho de banda de la red y tiempo del procesador, por lo que demora el progreso del sistema.

USO DEL GVT PARA MANEJO DE MEMORIA Y CONTROL DE FLUJO

Una de las características atractivas del mecanismo de Time Warp es que es posible dar un algoritmo natural y sencillo para la administración de memoria. Además de la memoria usada para el código y datos actuales de los procesos (de los cuales el programador es responsable de administrar), hay tres tipos de memorias adicionales para administrar:

1. Memoria de estados viejos en la cola de estados.
2. Memoria de mensajes almacenados en las colas de salida.

3. Memoria de mensajes "pasados" (en las colas de entrada) que ya han sido procesados.

Se administra a estas tres clases de almacenamiento, que son usadas únicamente para soportar rollback, de forma similar. Se puede descartar cualquier mensaje de una cola de entrada o salida cuyo tiempo virtual de recepción sea menor que el GVT. Del mismo modo, los procesos pueden descartar todos los estados almacenados con tiempo menor al GVT, excepto al mayor de ellos. Se suele llamar al proceso de destruir información menor que el GVT como *recolección de fósiles*.

Detección de Terminación Normal

Bajo el mecanismo de Time Warp, se detecta la terminación de una simulación mediante el uso del GVT. Se debe recordar que cada vez que un proceso se queda sin mensajes para procesar, termina su ejecución, asignando un valor de *+inf* a su reloj virtual local. Esta es la única circunstancia en que un reloj virtual puede valer este valor. Por lo tanto, cada vez que el GVT alcance el valor de *+inf*, es porque todos los relojes virtuales locales valdrán *+inf*, y no habrá mensajes en tránsito. Ningún proceso podrá realizar un rollback hacia un tiempo virtual finito (ya que no habrá nuevos mensajes de entrada), por lo que toda vez que el cálculo del GVT arroje un valor de *+inf*, le estará avisando al mecanismo de Time Warp sobre la terminación de la simulación.

Entrada y Salida

Cuando un proceso envía un comando a un dispositivo de salida o cualquier otro agente externo, es importante que no se realice en forma inmediata esta actividad de salida, dado que el proceso podría realizar un rollback y cancelar así este requerimiento. Sólo se puede realizar la salida física del comando cuando el valor del GVT excede al del tiempo virtual de recepción del mensaje que contiene el comando. Luego de este punto, no será posible que se genere un antimensaje para el comando, por lo que es seguro realizar un "commit" de la salida.

CUESTIONES DE PERFORMANCE

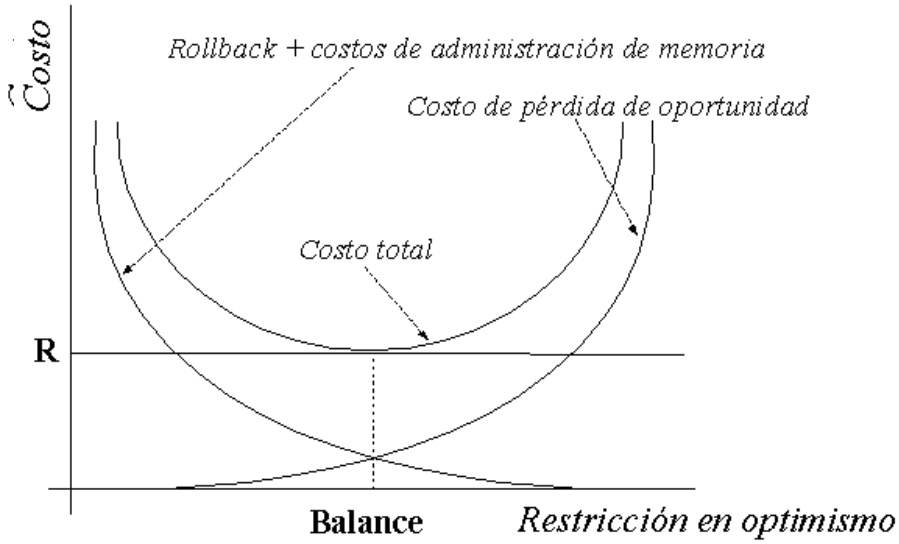
Se asume que el protocolo de TimeWarp es eficiente, ya que se podría asumir que el tiempo perdido por los procesos cuando incurren en rollbacks es igual o menor al tiempo que hubieran perdido por estar bloqueados a la espera de tener todas sus colas de entrada llenas, en el protocolo conservador.

2.3. Protocolos híbridos de simulación paralela de eventos discretos

Hemos estudiado hasta aquí dos clases de protocolos de sincronización bien opuestos: el optimista y el conservador, es decir, el agresivo y no agresivo respectivamente. La diferencia abrupta que existe entre ellos consiste en la forma de encarar la sincronización entre los procesos lógicos, es decir, que política adoptar para determinar eficientemente cuando un proceso lógico debe simular un evento.

Tal como hemos visto, con el protocolo conservador, un proceso lógico ejecuta un evento sólo después de determinar que el mismo es seguro, es decir, que su ejecución no producirá un error. Con el protocolo agresivo, los procesos lógicos ejecutan eventos sin la garantía de no producir errores; si ocurriera un error tras la ejecución de un evento, se adopta una política de recuperación del error usando un mecanismo de rollback. Esto implica que los protocolos agresivos incluyen un riesgo, el cual indica la posibilidad de propagar un error hacia otros procesos lógicos [Rey88]. Se suele utilizar el término *optimismo* para señalar conjuntamente al *riesgo* y a la *agresividad*, es decir, cuánto crece la probabilidad de realizar un cálculo erróneo en base a la agresividad con que cada PL simula los nuevos eventos.

En base a estos conceptos, se propone una nueva clase de protocolos PDES, los cuales son adaptables y se basan en información de estado casi perfecta (IECP) (Near Perfect State Information - NPSI) [Sri98]. Los mismos se caracterizan por el uso de información de estado casi perfecta para controlar, en forma adaptable, el optimismo en el avance de la simulación. Estos protocolos operan mediante el cómputo de un error potencial, basado en información de estado casi perfecta, tras lo cual se traduce en control sobre la agresividad y el riesgo. La tasa de progreso del tiempo simulado de un PL es controlada por su error potencial, el cual típicamente se calcula en función del estado del PL y el de sus predecesores. De este modo, podemos ver a estos protocolos como un híbrido entre los protocolos optimistas y los conservadores. Tratan de aprovechar el paralelismo inherente de los primeros, pero usando técnicas de restricción en el avance de la simulación, más cercana a los últimos. La idea entonces



es tratar ividad
desmedi sería
contrapr

2.

En los p to de estados, de rollback y de manejo de memoria. La limitación del optimismo propuesta por estos protocolos incluye un cuarto costo: el costo de pérdida de oportunidad. El mismo se refiere a la pérdida potencial de desempeño cuando un proceso lógico detiene la ejecución de sus eventos o envío de mensajes aún cuando era seguro que continúe. El costo del tiempo de almacenamiento de estados es una función definida en términos del tamaño del estado que es modificado por los eventos que se reciben y por la frecuencia del almacenamiento, pero no está definido en términos del nivel de optimismo, por lo que podemos ignorar dicho costo en este estudio. Entonces, para obtener un buen desempeño, los protocolos que controlan el optimismo deben minimizar la función de costo total :

costo total = costo de rollback + costo de manejo de memoria + costo de pérdida de oportunidad

Aunque la limitación del optimismo disminuye los dos primeros costos, incrementa al tercero, conduciendo al compromiso que se muestra en la Figura 2.4. El mejor desempeño posible para protocolos con optimismo controlado está caracterizado por R=0, donde R es el costo residual total cuando se alcanza el balance. Se logra R=0 cuando el optimismo controlado elimina tanto el costo de rollback como el de manejo de memoria sin agregar costo de pérdida de oportunidad (se obtienen valores positivos de R en la Figura 2.4 a partir de que se sube en el eje vertical respecto a la línea punteada con rótulo *R* en la parte izquierda).

Figura 2.4 - Compromiso obtenido por el optimismo controlado en el costo total de tiempo

Para alcanzar un buen balance, los protocolos deben identificar cálculos erróneos y limitar su propagación. Desafortunadamente, los requerimientos de sincronización PDES son dinámicos, irregulares y dependientes de los datos [Nic90] [Fuj90]. Esto se debe típicamente al hecho de que los sistemas simulados tienen un flujo de información que es traducido al simulador como una cadena de causalidad de eventos entre procesos lógicos; dado que la propagación de tales cadenas se basa en decisiones probabilísticas y parámetros de entrada, es imposible determinar a priori los flujos. En consecuencia, los dos requerimientos claves para que un protocolo de simulación sea eficiente son:

- que sea adaptable, y
- que utilice realimentación desde la simulación

Idealmente, estos requerimientos son satisfechos proveyendo a los procesos lógicos de información de estado perfecta, tal como cualquier cambio relevante en el estado del sistema que pueda ser instantáneamente visible. Dado que es imposible en la práctica lograr una información de estado perfecta, se explora el uso de información de estado casi perfecta (IECP). Se asume la existencia de un sistema de realimentación que opera en forma asincrónica con respecto a cada proceso lógico y le provee de IECP a bajo costo. Se describe a continuación el modelo de reducción para el sistema de realimentación, y se discute luego como debe ser llevado a la práctica.

2.3.2. Modelo de Reducción

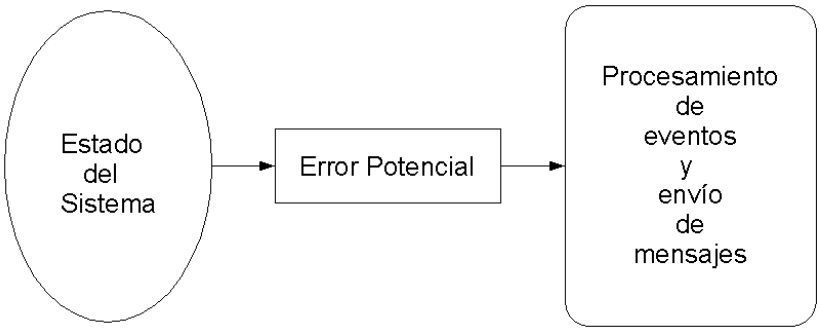
En el modelo de reducción [Sri98], el proceso lógico *i* codifica las partes relevantes de su estado en un conjunto de valores V^j_i, llamado vector de estados de entrada (input state vector - ISV). Un Calculador de Información de Estado Casi Perfecta (CIECP) procesa estos vectores para producir vectores de estado de salida (output state vector-OSV) cómo se detalla a continuación:

Sean ISV_1, ISV_2, ..., ISV_n los vectores de estado de entrada entregados por los *n* procesos lógicos de un sistema tal que ISV = $<V^1_i, V^2_i, ..., V^m_i>$, donde *m* es el tamaño de los vectores de estado de cada proceso lógico.

Teóricamente, el conjunto predecesor del proceso lógico PL_i podría ser cualquier subconjunto de los *n* procesos lógicos del sistema. En general, el subconjunto está basado en el grafo de comunicaciones definido por los procesos lógicos tal que existe un eje dirigido desde PL_a al PL_b si el PL_a puede enviar un mensaje al PL_b. El conjunto de predecesores del PL_i incluye al PL_j si existe un camino desde PL_j hasta PL_i en este grafo de comunicaciones. Ya que el conjunto de predecesores puede ser diferente para distintos procesos lógicos, se denomina a la información computada por el CIECP como *información de objetivo específico* [Pan93]. En el caso donde el grafo de comunicaciones es fuertemente conexo (existe un camino entre cada par de nodos), la información de objetivo específico es equivalente a la información global.

Como los procesos lógicos y el CIECP son mutuamente asincrónicos, se usan vectores de estado para forzar a que las interacciones entre ellos sean atómicas: un PL debe proveer un ISV completo y debe leer un OSV completo. Sin embargo, no se requiere que cada PLi lea cada OSVi generado por el CIECP. Se ha establecido en [Rey et al 93] que el criterio de atomicidad es suficiente para proveer consistencia secuencial, una propiedad que facilita el diseño de algoritmos de sincronización probablemente correctos usando este modelo. El modelo de reducción es suficientemente poderoso para proveer información útil tal como el tiempo virtual global. Es posible llevar a la práctica el modelo de reducción sobre una arquitectura de memoria compartida, dedicando un conjunto de procesos a la tarea de computar los OSVs usando los ISVs almacenados en memoria compartida, y que son actualizados por los otros procesos que participan de la simulación. También puede llevarse a cabo el modelo de reducción en arquitecturas de memoria distribuida usando una red asincrónica de reducción de alta velocidad tal como las redes de reducción paralela [Rey et al 93].

2.3.3. Protocolos Adaptables IECP



Los protocolos IECP son protocolos optimistas que controlan la agresividad y el riesgo de los procesos lógicos de manera dinámica, usando información de estado casi perfecta.

Hay dos fases en el diseño de un protocolo IECP:

- Identificar la información de estado sobre la cual se basan las decisiones de control del optimismo.
- Diseñar el mecanismo que traducirá esta información en control sobre el optimismo de los procesos lógicos.

Hay muchas opciones para implementar cada fase. Para facilitar el estudio independiente de cada una, las desacoplamos mediante la introducción de un error potencial (EP_i) asociado con cada proceso lógico *i*, para controlar el optimismo del PL_i. Se muestra el marco propuesto en la Figura 2.5. El protocolo mantiene cada EP_i actualizado a medida que la simulación progresa, mediante la evaluación, a una alta frecuencia, de una función de nombre *M1*, usando la información de estado que recibe desde el sistema de realimentación. De modo similar, una función *M2* traduce dinámicamente nuevos valores de EP_i en demoras en la ejecución de eventos y frecuencias en la comunicación de mensajes. De este modo, se pueden implementar diferentes protocolos mediante el cambio de las definiciones de las funciones M1 y M2. La dificultad del diseño radicará, a partir de aquí, en la identificación de mapeos para M1 y M2 que produzcan protocolos adaptables eficientes.

Figura 2.5 - Esquema propuesto para el estudio de un protocolo IECP

2.3.4. M1 - Cálculo de Errores Potenciales (EP)

El término EP_i indica la probabilidad de que la ejecución del PL_i se torne incorrecta en el futuro cercano. La clave para obtener un buen desempeño consiste en tener una función *M1* que prediga con una alta precisión si la computación de un proceso lógico deberá sufrir rollback. Un *M1* impreciso o mal diseñado puede producir un *EP* bajo cuando la ejecución del *PL* es errónea, lo que resulta en costos de rollback más altos, o un *EP* alto cuando la ejecución es correcta, por lo que ocurren costos de pérdida de oportunidad más altos. La información usada en el cálculo de *M1* debe ser tal que pueda ser obtenida usando el modelo de reducción. Uno de estos valores reducidos es el tiempo virtual global (Global Virtual Time - GVT).

Para ilustrar la naturaleza de la función M1, se listan algunos mapeos posibles para obtener el Error Potencial de un proceso lógico *i*:

1. EP_i = número de eventos locales ejecutados con timestamp > GVT. Este mapeo está basado en dos conceptos: el costo de un rollback es generalmente proporcional al número de eventos que realizaron rollback, y ningún proceso lógico puede hacer rollback a un tiempo lógico menor que el GVT. Por ende, el *EP* será proporcional a la máxima profundidad posible de rollback. Desafortunadamente, este mapeo no captura la probabilidad de que un *PL* realice rollback. Por ejemplo, consideremos dos proceso lógicos, uno en el tiempo lógico 100 que ha ejecutado dos eventos con timestamp mayor que el GVT, y otro con tiempo lógico 50 que ha ejecutado 10 eventos con timestamp mayor que el GVT. Este mapeo limitará al último de ellos aún cuando sea más probable de que el primero, al estar más lejos del GVT, sea el que deba sufrir rollback. Este razonamiento sugiere que el tiempo lógico del proceso lógico debería estar incluido en el mapeo.
2. Ep_i = reloj lógico del PL_i - GVT. El fundamento es que si un proceso lógico está adelantado respecto a los otros en el tiempos lógico y puede recibir mensajes de ellos, es bastante probable de que deba sufrir rollback. Este mapeo remedia el problema descrito en 1.

3. $EP_i = \eta_i$ - GVT, donde η_i es el tiempo del próximo evento planificado en el PL_i. Es razonable esperar que este mapeo supere el rendimiento del mapeo 2, puesto que η_i predice el próximo evento del PL_i mientras que el reloj lógico sólo describe al estado actual. Sin embargo, el proceso lógico con el mínimo η_i no puede determinar el hecho de que su próximo evento sea seguro de procesar cuando hay mensajes en tránsito, y por lo tanto continuará esperando por una cantidad de tiempo proporcional a su EP_i (el cual puede ser arbitrariamente grande) aún cuando pudiera proceder. En consecuencia, este mapeo puede conducir a una pérdida sustancial de oportunidad.

2.3.5. M2 - Control de Agresividad y Riesgo

M2 es una función que traduce un valor de error potencial (*EP*) en control sobre la agresividad y el riesgo de un proceso lógico. Debe ser diseñada de modo tal que a valores mayores de EP_i resulte en menor agresividad y/o riesgo en el PL_i. Un esquema simple es establecer un umbral tal que, cada vez que el valor del *EP* lo exceda, se suspenda la ejecución de eventos y comunicaciones del proceso lógico. Un esquema más sofisticado puede llevar a reducir gradualmente el índice de ejecución de eventos y comunicaciones, a medida que el *EP* se incrementa. Puede lograrse esta desaceleración insertando retardos en puntos apropiados, y luego *M2* sería una función que mapee el EP en un retardo de tiempo.

Un proceso lógico que envía mensajes a otros basándose en computación agresiva presenta riesgo [Rey88]. Dado que el riesgo contribuye al costo de rollback, debe incluirse algún mecanismo para controlarlo. Este mecanismo podría ser similar al usado para controlar agresividad: un proceso lógico podría detener el envío de mensajes mientras el *EP* exceda un umbral o podría gradualmente decrementar la tasa de envío de mensajes a medida que aumente el *EP*. Otra posibilidad es un mecanismo común que controle tanto agresividad como riesgo. Para el control de riesgo de los procesos lógicos, se puede generar cualquier esquema híbrido entre los dos extremos: una

estrategia con riesgo ilimitado tal como lo es TimeWarp, y una estrategia libre de riesgo tal como un mecanismo conservador.

2.3.6. Inserción de un protocolo adaptable IECP en Time Warp

Tal como se explicó en la sección 2.2.2 (El mecanismo de Time Warp), Time Warp es un protocolo de simulación optimista. Esto implica que adopta una política de simulación agresiva en cada proceso lógico en el desarrollo de una simulación. Dado que los protocolos IECP están basados en modificaciones o mejoras de los protocolos agresivos u optimistas, es inmediato observar la viabilidad de la implementación de estos nuevos protocolos en el protocolo de Time Warp. Bastará con agregar, para cada proceso lógico, el cálculo de un error potencial (función M1), y en base a éste implementar la función M2. El resultado de esta función puede determinar que el proceso lógico deba simular agresivamente, tal como lo estipulaba TimeWarp, o que simule en un modo conservador, en donde se deberá suspender por un lapso de tiempo la ejecución del proceso lógico, hasta que sea "seguro" que continúe.
Estas premisas nos permitirán insertar las técnicas usadas por los protocolos IECP en una implementación existente del mecanismo de Time Warp. Se detallarán las acciones realizadas para lograr este fin a partir del Capítulo 4.2 (Implementación de protocolos IECP en Warped).

2.4. Nuevos protocolos de simulación de eventos discretos distribuidos

A partir de los protocolos de simulación PDES descritos hasta aquí, muchos investigadores pretendieron usar y mejorar estas técnicas para lograr ejecuciones más rápidas y eficientes de distintos tipos de simulaciones. Por ende, presentaremos en esta sección diversas técnicas propuestas para enriquecer los mecanismos que ya hemos visto.

2.4.1. Modelo Frecuencia de Rollbacks

El objetivo de realizar una simulación en forma distribuida en lugar de secuencial en un sólo procesador es la de mejorar su aceleración, por lo que el tiempo total de ejecutar la misma será menor. Sin embargo, pueden darse casos en que algunos procesadores de la red funcionen más o menos rápido que otros, provocando de esta manera la ocurrencia de defasajes entre los tiempos de simulación de cada equipo, lo que equivale a decir, de cada proceso lógico. Esto puede provocar la ocurrencia de muchos rollbacks, y en consiguiente la pérdida de la aceleración buscada en la simulación distribuida.

Se estudió en consecuencia en qué estado puede encontrarse un proceso lógico de acuerdo a su *LVT* y a sus mensajes pendientes de entrada [Wai 98]:

- El tiempo simulado en el proceso lógico avanza más lentamente que el de los procesos lógicos vecinos. Esto, en general, provocará que cuando llegue un mensaje, el proceso lógico tenga un tiempo de reloj local inferior al de los timestamps de los mensajes recibidos desde los vecinos.
- El tiempo simulado en el proceso lógico avanza más rápidamente que el de los procesos lógicos vecinos. Por lo tanto, los errores de causalidad ocurrirán frecuentemente en este PL, con lo que será común la ocurrencia de rollbacks, ante la llegada de un evento de entrada con menor timestamp que el reloj virtual local.

Como fuera mencionado, una cuestión crítica enfrentada por las aproximaciones optimistas está relacionada con el overhead incurrido en el procesamiento de rollbacks. La ocurrencia de rollbacks provoca la explosión de antimensajes, que deben ser tratados.

En su primera definición, el comportamiento de los algoritmos de sincronización optimista fue considerado similar al de los usados para administración de memoria virtual en sistemas operativos. La observación de un alto grado de overhead en el sistema es equivalente a un comportamiento de thrashing en sistemas con memoria virtual, pero aquí la mayoría del tiempo de procesamiento de la simulación se pierde en calcular resultados incorrectos, que provocan nuevos rollback. Considerando estos

hechos, se propone emplear una técnica que ha dado buenos resultados en cuanto a la relación overhead/uso de recursos para sistemas de memoria virtual. Esta técnica, llamada "Modelo de Frecuencias de Faltas de Página" [Wul69], permite evitar el thrashing con bajo overhead.

La técnica modificada propuesta será llamada Modelo de Frecuencia de Rollbacks [Wai 98], y usará la misma estrategia para estudiar el número de rollbacks y evitar así su ocurrencia excesiva. La idea se basa en que cuando el número de Procesos Lógicos activos aumenta, el número de rollbacks también crecerá (exponencialmente), debido a que el número de antimensajes se multiplicará. En cambio, si el número de rollbacks es bajo, puede interpretarse como una señal de que el grado de paralelismo también será bajo, ya que la falta de rollbacks puede indicar un bajo grado de intercambio de mensajes.

Como en una simulación optimista o pesimista el número de procesos lógicos es fijo, la detección de una frecuencia de rollback alta para un proceso lógico es una indicación de que la simulación ejecutada por el procesador local debería demorarse. En este caso sólo se aceptarán nuevos mensajes de entrada que se agregarán en la lista de eventos locales, pero esta lista, al estar el proceso demorado, no será procesada. De esta forma se reduce el número total de procesos lógicos activos y, por ende, en determinado momento el número de rollbacks se reducirá.

Por otro lado, si el número de rollbacks es muy bajo, esto es una indicación de que el procesamiento de la lista de eventos local está avanzando lentamente. Por ende, si existe algún proceso lógico que haya sido demorado, deberá ser reactivado para que vuelva a ejecutar normalmente.

En resumen, esta técnica busca demorar la ejecución de aquellos procesos lógicos que sufrieron un número excesivo de rollbacks, ya que se parte de la premisa de que están generando cálculos erróneos y que inundan de antimensajes a sus vecinos. Se permite así que avancen en la simulación sólo los procesos lógicos que realizan cómputos correctos. En algún momento de este avance, los procesos demorados tendrán un

número muy bajo de rollbacks (ya que no estaban ejecutando), por lo que se podrá decidir reactivarlos para que vuelvan a ejecutar del modo normal.

Se destaca de este mecanismo que puede ser implementado como un protocolo IECP. La función M1, que calcula el error potencial de un proceso lógico, deberá realizar el cálculo de cuántos rollbacks sufrió el proceso lógico; la función M2 decidirá una simulación agresiva o conservadora en base a si este número es o no elevado.

2.4.2. Protocolo de sincronización distribuida usando analogía con Memoria Virtual

En [Jef85] se definió por primera vez el algoritmo de Time Warp. En este paper se desarrolló una extensa relación entre los conceptos de tiempo virtual y de memoria virtual, concepto conocido de la teoría de sistemas operativos. En base a estas comparaciones, repasaremos a continuación una política de administración de memoria virtual, llamada Working Set, veremos luego las analogías entre tiempo y memoria virtual, y finalizaremos utilizando todos estos conceptos para definir un nuevo protocolo de sincronización distribuida, al que llamaremos working set PDES**.**

EL MODELO DE WORKING SET

El modelo de Working Set [Sil98] para la administración de memoria virtual está basado en el concepto de *localidad* de un proceso. El modelo utiliza un parámetro Δ, para definir una ventana de trabajo. La idea es examinar los Δ accesos más recientes a páginas de memoria. Se denomina a este conjunto de páginas como *working-set*.
Si se accede a una página en forma frecuente, ésta pertenecerá a dicho *working-set*. Si la misma deja de ser accedida, abandonará al *working-set* luego de Δ referencias a memoria luego de su último acceso. Por tal razón, se dice que el *working-set* se refiere a la *localidad* de ejecución de un programa. Se muestra en la Figura 2.6 un ejemplo para este modelo, con $\Delta = 5$. El conjunto de las páginas que pertenecen al *working-set* en el tiempo t_1 es el formado por {67543}, ya que fueron las últimas 5 páginas

accedidas. En cambio, el *working-set* en el tiempo t_2 es {3,4}, ya que éstas fueron las únicas páginas requeridas en los últimos 5 accesos.

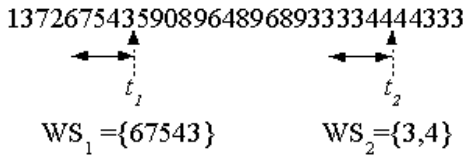


Figura 2.6 - Modelo de working-set con Δ=5

La exactitud del *working-set* depende de la elección de Δ. Si este valor es muy pequeño, no estará conteniendo a la *localidad* de páginas, por lo que los fallos de página serán muy comunes; si el valor es muy grande, podría estar conteniendo varias *localidades*, ocupando así páginas físicas que no serán accedidas en lo inmediato, con el desperdicio consiguiente que esto provoca. Si se tomara Δ como infinito, el *working-set* estaría compuesto por todas las páginas del programa en ejecución.

Es muy sencilla la utilización de este modelo en la administración de memoria virtual en un sistema operativo. El mismo monitorea el working-set de cada proceso y le asigna al mismo tantas páginas en memoria como el tamaño del working-set. Si existen suficientes páginas en memoria para ejecutar a otro proceso, se aplica el criterio anterior, y así sucesivamente. Si la suma total de working-sets excede a la cantidad de páginas en memoria, el S.O. escoge un proceso y lo suspende, por lo que graba sus páginas en memoria secundaria y asigna en su lugar al nuevo proceso.

ANALOGÍA ENTRE MEMORIA VIRTUAL Y TIEMPO VIRTUAL

Es interesante resaltar que existe una analogía entre el paradigma de tiempo virtual y otro fenómeno ya estudiado en teoría de sistemas operativos: la memoria virtual [Jef85]. En realidad, el término "tiempo virtual" evoca al concepto "memoria virtual". Se

podrían sustituir los términos temporal y espacial, ya sea que se trate de tiempo virtual o memoria virtual respectivamente. Se presenta a continuación una lista de conceptos en donde el tiempo y el espacio juegan roles casi simétricos:

- Consideramos a una página en memoria virtual como análoga a un evento en tiempo virtual. La dirección virtual de una página es su coordenada espacial; el tiempo virtual de un evento es su coordenada temporal.
- Consideramos a una página residente en memoria principal en el tiempo *t* como análoga a un evento con un tiempo virtual en el futuro del proceso *x* (es decir, que tiene un tiempo virtual mayor que el reloj virtual del proceso *x*); una página fuera de memoria en el tiempo *t* es análoga a un evento en el presente o pasado del proceso *x*.
- Acceder a una página residente en memoria principal es comparativamente barato, pero acceder a una página fuera de la memoria principal causa una falla de página muy cara. En forma similar, el envío de un mensaje que arriba en el futuro virtual del proceso receptor es comparativamente barato, pero el envío de un mensaje en el pasado virtual del receptor causa una falla de tiempo muy cara, o sea, un rollback.
- Bajo un sistema de memoria virtual, sólo es eficiente ejecutar programas que obedecen al principio de localidad espacial, de modo que la mayoría de los accesos a memoria son a páginas que ya están residentes en memoria, y por ende son relativamente escasos los fallos de página. Del mismo modo, bajo un sistema de tiempo virtual, sólo es eficiente ejecutar programas que obedecen al principio de localidad temporal, es decir, la mayoría de los mensajes arriban en el futuro virtual del proceso destino, de modo tal que los fallos de tiempo sean relativamente escasos. Nota: el costo de un fallo de página está en el orden de 10000 veces el costo de un simple acceso a memoria. Más allá de que no se hayan realizado medidas empíricas, es difícil imaginar una implementación en donde un fallo de tiempo causada por un mensaje cueste más que cien veces respecto a un mensaje que no cause fallo alguno. Por lo tanto, podemos esperar que se pueda tolerar una fracción considerablemente mayor de fallos de tiempo que fallos de página.

- El término "mapeo de memoria" se refiere a la traducción de direcciones virtuales a direcciones reales. Podríamos usar el término "mapeo de tiempo" para referirnos al mapeo de tiempo virtual en tiempo real (es decir, decidir cuando en el tiempo real un evento con un tiempo virtual dado será ejecutado). Sin embargo, ya existe un término para este concepto: "planificación". Es importante notar que la misma dirección virtual podría ser mapeada a diferentes direcciones reales en diferentes momentos, y de modo similar, el mismo tiempo virtual podría ser mapeado (o planificado) en diferentes tiempos reales en diferentes lugares.
- Si un proceso puede tener suficientes páginas en memoria principal, su tasa de fallos de página puede verse reducida a cero; en general, esto no es deseable porque llevaría a un uso ineficiente de la memoria principal. Análogamente, si se demora la ejecución de un proceso lo suficientemente tal que se convierta en el proceso más lento del sistema, se reduciría su tasa de fallos de tiempo a cero; esto tampoco es deseable porque se convertiría en un proceso "cuello de botella" para la simulación, ya que retendría el avance del GVT, haciendo de este modo un uso ineficiente del tiempo real.

DISEÑO DEL PROTOCOLO DE SINCRONIZACIÓN DISTRIBUIDA USANDO ANALOGÍA CON MEMORIA VIRTUAL

Utilizaremos las analogías dadas en [Jef85] para memoria y tiempo virtual, con el fin de diseñar un nuevo protocolo de sincronización PDES. Así como es deseable que un proceso que ejecuta en memoria virtual obedezca al principio de localidad espacial, también es deseable que un proceso que ejecuta en tiempo virtual obedezca al principio de localidad temporal. La localidad espacial se refiere a que todos los accesos a memoria virtual por parte de un proceso sean a un conjunto determinado de páginas, las cuales deben estar residentes en memoria. De este modo, se minimizan los fallos de página. Análogamente, la localidad temporal se refiere a que todos los eventos que simula un proceso estén dentro de un lapso de tiempo dado, a partir del futuro simulado. De este modo, al no recibir eventos en el presente o pasado, se está minimizando la cantidad de rollbacks que un proceso sufre.

La propiedad de localidad espacial es deseable, pero es imposible lograr el compromiso de que un proceso la cumpla, ya que dependiendo de la etapa en que se encuentre su ejecución podrán ser distintas las páginas de memoria a las que precisa acceder. Del mismo modo, la propiedad de localidad temporal es deseable, pero no garantiza que la simulación de eventos dentro del "working set" de cada PL (es decir, dentro de una ventana de tiempo dada a partir del presente simulado) evite en el futuro la llegada de un mensaje en el pasado, y le provoque un rollback al proceso receptor del mismo. En el caso de memoria virtual se busca optimizar el uso de la memoria física, y lograr un desempeño acorde del procesador en la ejecución del proceso si hay pocos fallos de página. En el caso de tiempo virtual se busca optimizar el desempeño de cada proceso lógico, ya que si hay pocos rollbacks se podrá realizar la simulación en un tiempo menor.

En base a estos conceptos, definimos el protocolo **working set PDES**, el cual se comporta de la siguiente manera:

- Todo proceso lógico simulará eventos cuyo tiempo virtual esté comprendido en una ventana de tiempo. La misma estará comprendida desde el Tiempo Virtual Global (Global Virtual Time - GVT) hasta el GVT más un *intervalo T* de tiempo, el cual debe ser especificado al iniciar la simulación. Si un proceso lógico tuviera sólo mensajes de entrada cuyo timestamp excede al tiempo dado por *GVT+T*, se suspende la ejecución del mismo (aunque seguirá aceptando mensajes de sus procesos vecinos). El PL queda en esta situación hasta tanto posea en la cola de entrada un mensaje que cumpla la condición de estar dentro de la ventana de tiempo [*GVT*, *GVT+T*]. Esta situación se puede dar ya sea por el arribo de un nuevo mensaje, o por el avance de la simulación, con el posterior incremento del GVT.

Con esta definición se logra limitar la longitud de las cadenas de rollback, y por lo tanto se logra un mejor control de los rollbacks en cascada. Esto se justifica en el hecho de que un proceso lógico no puede simular hasta un tiempo mayor a *GVT+T*, y que nunca

recibirá un evento con timestamp menor al GVT (o sea, nunca hará rollback a un tiempo menor al GVT).

Se destaca de este mecanismo que, al igual que el Modelo Frecuencia de Rollbacks estudiado en 2.4.1, también puede ser implementado como un protocolo IECP. La función M1, que calcula el error potencial de un proceso lógico, deberá realizar el cálculo (tiempo virtual del próximo evento - GVT); la función M2 decidirá una simulación agresiva o conservadora en base a si este número es menor o mayor respectivamente al valor de *T*.

2.4.3. NoTime (Simulación Paralela de Eventos Discretos No Sincronizada)

Hasta aquí hemos estudiado protocolos de simulación de eventos paralela y distribuida (PDES) que abarcan desde los más optimistas o agresivos hasta los más pesimistas o conservadores. Sin embargo, todos ellos se basan en la noción de causalidad y de sincronización entre procesos lógicos. Debido a esto, se realizan constantes investigaciones para reducir el overhead que tiene cada tipo de simulación: lookahead y deadlock en los conservadores, rollbacks y almacenamiento de estados en los optimistas. Muchos de estos overheads pueden ser atribuidos a la sincronización (o mantenimiento de la causalidad) requerida en el protocolo de simulación. En consecuencia, para atacar este problema, se han propuesto técnicas de simulación que proponen relajar la causalidad estricta requerida en simulaciones distribuidas.
Puede ocurrir que, dependiendo de la simulación, el usuario sepa de antemano que algunos mensajes serán incorrectos pero que, sin embargo, decida ignorarlos. Esto se puede deber a que se desea obtener resultados menos precisos pero con una mayor rapidez. Desde ya que esto depende del tipo de aplicación que se simula y el tipo de resultados que se desea monitorear.

PROTOCOLO NOTIME

Algunos autores de la herramienta de simulación Warped propusieron en [RTRW98] ignorar las violaciones de causalidad (evitando de este modo la necesidad de sincronización en la simulación). Desarrollaron e implementaron una herramienta que permite ejecutar simulaciones de eventos discretos sin restricciones de causalidad. La implementación de las simulaciones es similar a la usada para simulaciones de TimeWarp bajo Warped, por lo que lograron realizar simulaciones con ambos protocolos y con una interface consistente.

DIFERENCIAS DE IMPLEMENTACIÓN CON TIMEWARP

La diferencia principal entre NoTime y TimeWarp es que, al no respetar NoTime las restricciones de causalidad, la cola de eventos de entrada de cada objeto obedece a una política FIFO (First Input First Output) en lugar de ser una lista ordenada de eventos. De este modo, cada objeto simula el evento que primero le llegó, en lugar del que tiene menor timestamp de llegada.

VENTAJAS DEL PROTOCOLO NOTIME

Entre las ventajas que este protocolo tiene sobre TimeWarp, se mencionan [RTRW98]:

- Menores tiempos de ejecución de las simulaciones
- Menor uso de memoria, debido a que en NoTime no se deben almacenar los estados de un objeto para el caso de incurrir en un rollback.
- Los resultados obtenidos con este protocolo tienen un pequeño error respecto al protocolo de TimeWarp, por lo que en muchos casos conviene utilizarlo.
- No es preciso modificar el paradigma de simulación para utilizar este protocolo.

3. Implementación del protocolo de sincronización Time Warp: WARPED

3.1. Características del kernel de simulación Warped

El proyecto warped se creó en el "Computer Architecture Design Laboratory" de la Universidad de Cincinnati, logrando una versión de desarrollo hacia fines de 1997. La finalidad del proyecto es la de brindar un kernel de simulación para el protocolo de Time Warp, que sea gratuito, portable a varias plataformas y sistemas operativos, y sencillo de modificar y extender [MMRW97].

Warped está implementado en C++, y utiliza la orientación a objetos y los conceptos de clase y sobrecarga de operadores que brinda el lenguaje.

Este kernel respeta el protocolo de simulación optimista y distribuida. Se basa en el paper original publicado por Jefferson sobre TimeWarp [Jef85].

Warped modela a los objetos de una simulación como entidades, las cuales reciben y envían mensajes, y actúan en consecuencia a ellos mediante la actualización de su estado interno.

En Warped, se utiliza el término *evento* para referirse al elemento de comunicación entre los objetos de una simulación (en este trabajo utilizamos el término *mensaje* para este fin), y el término *mensaje* para la comunicación entre procesos del mismo kernel.

Otra diferencia del kernel respecto al paper de Jefferson es que se define a cada entidad del sistema lógico con el nombre de *objeto*. Luego, se agrupan los objetos que ejecutan en un mismo procesador en un mismo *proceso lógico.* Entonces, se utiliza el término proceso lógico sólo para referirse a la entidad que agrupa a un conjunto de objetos de una simulación, pero cuando haya que hacer referencia en Warped a un proceso lógico que modela un proceso físico del sistema real, se lo deberá hacer con el nombre de *objeto*.

Los objetos ubicados en el mismo proceso lógico (o sea, que ejecutan en el mismo procesador) se comunican entre sí sin la intervención del sistema de mensajes, lo cual permite una comunicación mucho más rápida. Por ende, se deben ubicar en el mismo proceso lógico a todos los objetos que se comunican entre sí con una alta frecuencia. Además de albergar a un conjunto de objetos, un proceso lógico también es responsable de la planificación de dichos objetos en la simulación.
Al ejecutar una simulación, se irán alternando en el control de la misma el kernel de Warped y el código a nivel de aplicación. Se logra lo dicho mediante el uso cooperativo de llamadas a funciones. Con el fin de que el kernel interactúe correctamente con el código de aplicación, el usuario debe proveerle al kernel varias funciones. Estas funciones definen, entre otras cosas, como inicializar cada objeto de la simulación, y que hace cada uno de ellos durante un ciclo de la misma. Además, el usuario puede utilizar una definición de tiempo virtual que no se corresponda con un número entero; para tal fin, le alcanza con utilizar una nueva clase para la implementación del tiempo.
Mediante el aprovechamiento de varias características del lenguaje C++, se logra la interacción entre el código de aplicación y el kernel. El kernel de simulación está definido como varias clases modelo (template), por lo que el usuario puede definir parámetros del sistema sin reescribir código del mismo. El código de la aplicación se deriva del código del kernel, por lo que puede acceder a funciones del mismo en forma transparente.

Warped fue desarrollado y compilado usando el compilador g++ (compilador C++ de GNU), versión 2.7.0. Fue diseñado para funcionar en forma paralela y distribuida, ya sea en equipos multiprocesador o en redes heterogéneas de computadoras. Para tal fin, se utiliza el protocolo de envío de mensajes MPI.

3.2. MPI (Message Passing Interface)

MPI es una especificación estándar para la creación de librerías de pasaje de mensajes. La implementación usada del estándar MPI por Warped es el paquete

mpich, versión 1.1. Dado que la misma es de dominio público, es posible obtener una copia en http://www.mcs.anl.gov/home/lusk/mpich. La librería es portable a varios entornos y sistemas operativos.

Mpich permite ejecutar una aplicación en un entorno distribuido, y que las distintas instancias que ejecutan en diversos procesadores se puedan comunicar entre sí. Para lograr este objetivo, se pueden incluir en el código fuente C++ llamadas a diversas funciones, que permiten la comunicación entre pares de procesos, o de uno a varios procesos (broadcast). Se debe vincular (link) al programa original con las librerías mpich, por lo que se genera un único ejecutable.

Para lanzar la ejecución paralela, se define a un equipo como "maestro", por lo que inicia esta ejecución, y le avisa a los demás procesadores del inicio de la misma. Debido a este mecanismo de ejecución, se debe incluir en el equipo "maestro" un archivo de configuración mediante el cual se le indica qué equipos de la red ejecutarán la aplicación en forma paralela.

3.3. Interacción entre el kernel y el código de aplicación

3.3.1. Definición de los objetos

La clase *TimeWarp* define los métodos y datos que cada objeto de la simulación precisa para operar en el sistema. El usuario puede definir muchos tipos diferentes de objetos (dependiendo el proceso físico del sistema real que modela), pero todos estarán derivados de la clase *Timewarp*.

Esta clase define otras variables que necesitan ser correctamente utilizadas en la simulación. Por ejemplo, el usuario precisa asignar un número entero en el campo *id* de cada objeto, antes de comenzar la simulación. Cada identificador debe ser único. En una simulación con *n* objetos, los identificadores deben estar en un rango de 0 a *n*-1.

Otro de los identificadores definidos en la clase *TimeWarp* es *name*. El mismo es una cadena de caracteres que no precisa ser única, pero probablemente sea más útil que

así fuera; cuando el kernel genera mensajes de error en un objeto, suele mostrarlo junto al nombre del mismo.

En cada ciclo de la simulación, un objeto usará su *estado* para determinar las acciones que tomará en el siguiente ciclo ante la recepción de un evento.

Los elementos básicos definidos en cada objeto de la simulación incluyen a las colas de entrada y salida, y a la cola de estados. La cola de entrada almacena eventos que el objeto necesitará procesar (y posiblemente alguno que ya haya procesado). La cola de salida almacena eventos que el objeto ha generado y enviado a otros objetos. El objeto tiene un registro de cada mensaje que envió durante la simulación, de modo que, en caso de un rollback, pueda enviar eventos de signo negativo o antimensajes (como lo indica el algoritmo de TimeWarp). La cola de estados almacena los estados anteriores que se precisarán en caso de rollback en algún tiempo posterior al GVT. Las tres clases utilizadas para el manejo de colas se derivan de la clase template *SortedList*, que es una lista ordenada de elementos.

3.3.2. La clase InputQueue

La clase *InputQueue* es una lista ordenada que contiene elementos de la clase *BasicEvent* (o sea, eventos de la simulación). La función de comparación dada a *InputQueue* es una comparación de *BasicEvent*, permitiendo así que el ordenamiento se realice en base al timestamp de cada evento.
La función de inserción de eventos aniquila automáticamente a dos eventos iguales con signo opuesto, como debe ocurrir de acuerdo a la metodología TimeWarp. Luego de insertar un evento, la función de inserción retorna un valor de verdad que indica si el tiempo de recepción del evento es anterior al del último evento ejecutado, de modo que el objeto pueda hacer un rollback si es necesario. La función retorna *true* para indicar que el mensaje fue en el pasado, y retorna *false* para indicar lo contrario.

Otra función disponible es *gcollect(VTime)*. La misma invalida todos los elementos de una cola con un tiempo virtual que esté en el pasado.

3.3.3. La clase OutputQueue

Esta clase es una lista ordenada que contiene elementos de la clase *BasicEvent*. La función de comparación dada es una comparación de *BasicEvent*, permitiendo así que el ordenamiento se realice en base al tiempo virtual de envío de cada evento. La clase *OuputQueue* tiene una función llamada *gcollect(Time)*, cuya esencia es similar a la descrita para *InputQueue*.

3.3.4. La clase StateQueue

La clase *StateQueue* es una lista ordenada que contiene referencias a objetos de la clase *BasicState* (estados del objeto). La función de comparación dada a la *SortedList* es una comparación entre *BasicState*, donde la clave de ordenación es la *LVT* (Local Virtual Time) de cada uno de los estados.
Como las clases *InputQueue* y *OutputQueue*, la clase *StateQueue* tiene una función llamada *gcollect(VTime)*, que es la encargada de remover todos los estados de la cola con tiempo anterior al *LVT* actual. Sin embargo, en esta clase, se debe almacenar un estado anterior a este tiempo, de acuerdo al algoritmo de TimeWarp. En el caso de que ocurriera un rollback en el tiempo *x*, se deberá restaurar el estado del objeto con el tiempo inmediatamente anterior a *x*.

3.3.5. Métodos que el código de aplicación le provee al kernel

Hay tres métodos principales que el kernel invoca del código de aplicación (es decir, la simulación que programa el usuario). El primero es *initialize*. Este método no tiene parámetros, y es invocado exactamente una vez en cada objeto, al comienzo de la simulación. Dado que warped requiere que cada objeto de la simulación sea manejado "por eventos", una tarea que suele realizar la función mencionada es la de que algún objeto comience la simulación mediante el envío de un evento a sí mismo. El siguiente

método de aplicación que el kernel invocará es *executeProcess*. En el mismo se implementa el comportamiento del objeto, cada vez que es planificado para ejecutar. Si un objeto no tiene un evento de entrada para procesar, es considerado ocioso, y no será planificado para ejecución. Cada vez que un objeto ejecuta este método, el kernel habrá pasado todo el control a la aplicación; hasta que el control no retorne al kernel, ningún otro objeto del proceso lógico podrá ser planificado.
Cada objeto realiza las siguientes actividades cuando es planificado para ejecutar:

- Hace una o más llamadas a la función *getEvent()*, para obtener los eventos de entrada pendientes.
- Procesa el/los eventos recibidos, y almacena el nuevo estado resultante por esta acción
- Envía cualquier nuevo evento a otros objetos a través de llamadas sucesivas a la función *sendEvent()*
- Devuelve el control al kernel

El último método definido en el nivel de aplicación y que es invocado por el kernel es *finalize*. Este método es opcional, y es llamado por única vez en cada objeto al final de una simulación. Esta llamada tiene la intención de permitirle al objeto coleccionar y mostrar estadísticas o de realizar cualquier última tarea.

3.3.6. Eventos

La clase básica para el uso de eventos en una simulación es *BasicEvent*. La misma almacena la cantidad mínima de información necesaria para enviar un evento de un objeto a otro: objeto emisor, objeto receptor, tiempo de envío y de recepción y tamaño del evento. El signo es manejado internamente por el kernel. Los eventos generados por algún objeto para ser enviado a otro objeto son completados con el signo *POSITIVO*. En el caso de la ocurrencia de un rollback, el kernel envía *antimensajes*, es decir, eventos con signo *NEGATIVO*. Es posible definir nuevos tipos de eventos más complejos (que contengan más información), tomando como base la clase *BasicEvent*.

3.3.7. La clase LogicalProcess (Proceso Lógico)

Esta clase agrupa a uno o más objetos de la simulación, los cuales comparten un *GVTManage*r (controlador del GVT de la simulación), un *CommManager* (administrador de las comunicaciones entre objetos) y un *Scheduler* (Planificador). El ciclo de ejecución principal de la simulación reside en el método *LogicalProcess::simulate*. Este le dice al proceso lógico que mantenga la ejecución de la simulación hasta que reciba un nuevo broadcast de GVT (Global Virtual Time) con tiempo de simulación *PINFINITY*. En la implementación actual, puede haber un único proceso lógico por proceso Unix. Se pretende cambiar esto en el futuro para permitir un proceso lógico por hilo (thread) de ejecución.

CONSTRUCCIÓN DE PROCESOS LÓGICOS

Para crear una instancia de esta clase, se debe llamar a un constructor con tres parámetros enteros:

- El número total de objetos en la simulación
- El número de objetos locales al proceso lógico
- El número de procesos lógicos en la simulación

REGISTRANDO OBJETOS CON UN PROCESO LÓGICO

Todo objeto perteneciente a un proceso lógico debe llamar al método *registerObject* del proceso lógico al que pertenece. De este modo le indica al mismo que éste será el responsable del objeto en la simulación. Esta acción se realiza generalmente en la rutina *main* de la simulación.

UN PROCESO LÓGICO COMIENZA LA SIMULACIÓN

Luego de que todos los objetos se hayan registrado con su proceso lógico, se debe invocar a los métodos *LogicalProcess::allRegistered* y *LogicalProcess::simulate* - que toman un parámetro del tipo VTime (tiempo virtual) en forma opcional -. Si se las llama con un parámetro, la simulación correrá hasta que el tiempo de simulación global haya

excedido el tiempo especificado. Si no se provee dicho parámetro, la simulación correrá hasta alcanzar su completitud.

CICLO PRINCIPAL DE UNA SIMULACIÓN

El ciclo de ejecución principal de una simulación en cada proceso lógico (o sea, en cada procesador de la red) reside en el método *LogicalProcess::simulate*. El mismo realiza las siguientes acciones :

- Si es necesario, el GVT Manager realiza cálculos locales del GVT.
- El Communication Manager recibe mensajes del subsistema de mensajes y los distribuye a los objetos destino correspondientes.
- El Planificador (Scheduler) selecciona el siguiente objeto de la simulación para ejecutar.

COLA DE EVENTOS DE ENTRADA

El planificador contenido en cada proceso lógico planifica de acuerdo a un único tipo de planificación: objeto con menor timestamp primero (lowest timestamp first - LTSF). Para realizar esta planificación, se declara *static* a la cola de entrada de los objetos *TimeWarp*, para que todos compartan la misma instancia de cola. Estando todos los eventos en una misma cola, resulta trivial la selección del próximo evento a ejecutar en cada proceso lógico. El criterio de ordenación de los eventos en la cola es: primero por menor timestamp de recepción del evento, y luego por menor *id* del objeto receptor.

3.3.8. El archivo de configuración config.hh

Este archivo de configuración le permite al usuario cambiar varias características del kernel en tiempo de compilación. El archivo está situado en el mismo directorio que el kernel de Warped, y en el mismo se describen todas las opciones disponibles para modificar. Entre estas opciones se hallan las siguientes:

- *Estrategias de cancelación* - Las alternativas son: cancelación agresiva, ociosa (lazy) y permutación dinámica entre las dos cancelaciones dinámicas.

- *Un antimensaje por rollback* - Se envía sólo un antimensaje (mensaje de signo negativo) por rollback, aquel con menor timestamp. Se suprimen los demás antimensajes. En el lado del receptor, cada vez que se recibe un antimensaje, se eliminan todos los mensajes con timestamp mayor al del antimensaje recibido. Esto permite una reducción significativa en el número de mensajes que se envían en el sistema, por lo que se mejora la performance del mismo.

3.3.9. Entrada/Salida en Warped

Warped permite que los objetos realicen entrada/salida (I/O) de archivos durante la simulación, ya que maneja en forma apropiada la clase *FileQueue* durante la ocurrencia de rollbacks y garbage collection. Actualmente, sólo se pueden abrir archivos para escritura; todavía no hay soporte en el kernel para manejar archivos de entrada. Si un objeto necesita realizar una operación de escritura durante la simulación, deberá informar al kernel de esta situación antes de comenzar la ejecución de la simulación. Para tal fin, se deben realizar tres tareas; informar al kernel del número de archivos a utilizar, crear el número apropiado de objetos de la clase *FileQueue*, e informar a cada uno de estos el nombre del archivo físico con el que estarán asociados.
Cuando se actualiza el *GVT*, se escriben físicamente al archivo de disco todas las líneas de la cola cuyo tiempo sea menor o igual al mismo.

3.3.10. Detección de terminación de una simulación

Warped utiliza un tipo especial de mensaje, llamado *CheckIdleMsg*, para poder detectar el fin de una simulación. Estos mensajes circulan por todos los procesos lógicos para determinar si hay uno de ellos que no esté ocioso. Este mensaje tiene un campo llamado *cancel*, que le dice al proceso lógico receptor si el emisor desea o no cancelar la circulación del mensaje, y un campo entero llamado *numCirculations*, que le indica al proceso lógico receptor cuantas veces el mensaje visitó **todos** los procesos lógicos. Cuando un proceso lógico recibe este mensaje, reenvía el mismo al proceso lógico con identificador *id* inmediatamente superior (o al de *id* = 0 si él es el último proceso lógico),

en caso de que se encuentre ocioso. En caso contrario, reenvía el mensaje al proceso lógico 0 con el campo *idle* = *false*. Cuando este evento haya circulado por todos los objetos de la simulación y no haya sufrido modificación en su campo *cancel*, notará que todos los objetos de la simulación están ociosos, por lo que se podrá finalizar en forma correcta y segura la simulación.

4. Implementación de otros protocolos en Warped

4.1. Implementación del protocolo de sincronización de Misra sobre WARPED

Una vez analizada la estructura interna de Warped, nos aprestamos a realizar desarrollos a partir de ella. El primer paso es el de adaptar al kernel de modo tal que el mismo se pueda comportar respetando el protocolo de sincronización pesimista de Misra, al cual ya hemos estudiamos en la sección 2.1.

4.1.1. Diferencias entre los protocolos

Como se explicó en la sección 3.3.7, Warped utiliza una misma cola de entrada de eventos para todos los objetos de un proceso lógico. Mediante la utilización de un planificador LTSF(lowest timestamp first), se planifica para ejecutar el objeto cuyo evento en la cola posea el menor tiempo de recepción. Sin embargo, debemos modificar esta estructura para implementar el protocolo pesimista de Misra. Veremos esta necesidad a partir del esquema de ejemplo de la Figura 4.1. En la misma, los objetos 1 y 2 pertenecen a un mismo proceso lógico, el *PL1*. El objeto 1 puede recibir mensajes de los objetos 3, 4 y 5, que pertenecen al proceso lógico 2. El objeto 2 puede recibir mensajes sólo del objeto 5. Se muestra esta dependencia a través de flechas desde el PL origen al PL destino.

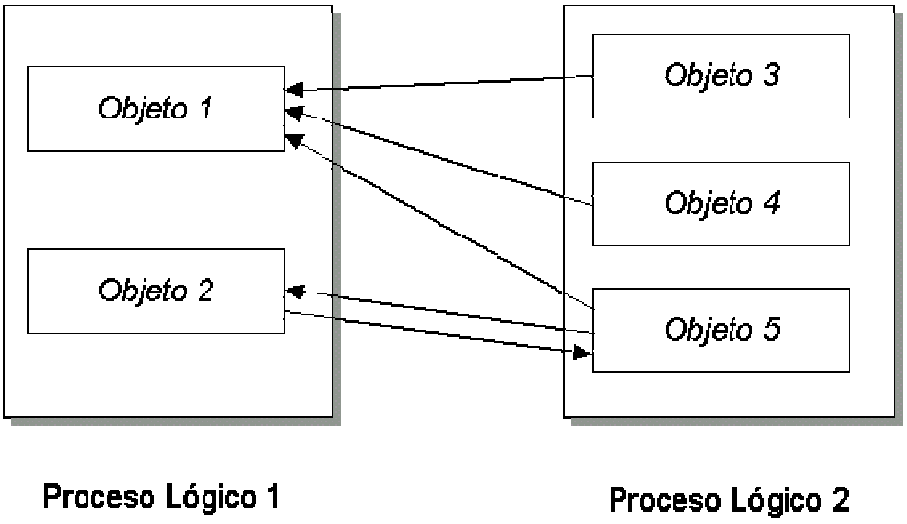


Figura 4.1 - Ejemplo de simulación con dos procesos lógicos y 5 objetos.

Supongamos que el objeto 1 y del objeto 2 tienen un LVT (Local Virtual Time) de 5, y que la cola de entrada del PL1 contiene los eventos que se observan en la siguiente tabla:

Número evento	Emisor	Receptor	Tiempo de recepción
1	3	1	6
2	5	2	7

Con el esquema tradicional usado en Warped, el planificador del proceso lógico 1 trataría de ejecutar al objeto 1 con el mensaje que recibió del objeto 3, ya que siempre toma una postura de simulación optimista y agresiva. Sin embargo, las entradas del objeto 1 respecto a los objetos 4 y 5 están vacías, por lo que el protocolo conservador no dejaría ejecutar a este objeto, ya que no sabe si es "seguro" procesar el evento. Se observa así que se debe modificar la rutina *getEvent()* del kernel de simulación warped, que es la encargada de entregarle a un objeto el evento con menor timestamp recibido, para que le entregue el evento al objeto sólo si éste tiene todas sus colas de entrada con al menos un mensaje (en nuestro ejemplo, debería informarle al objeto 1 del evento que le envió el objeto 3 sólo cuando este objeto también haya recibido eventos de los

objetos 4 y 5, y el tiempo de recepción de éstos sea mayor al del primero). Entonces, la función *getEvent()* debe, en vez de devolver el primer evento de la cola de entrada al objeto receptor del mismo, recorrer esta cola para ver si el objeto receptor del evento tiene todas sus entradas "ocupadas". Si fuera así, la función se comportaría de igual modo a como lo hacía originalmente, devolviendo el primer evento; en caso contrario, no devuelve evento alguno, ya que, según el protocolo conservador, el objeto no está listo para ejecución, hasta no estar completamente seguro de cuál será el próximo evento con menor timestamp que recibirá. Luego, el puntero de la cola quedaría inmóvil en el primer evento, hasta que el objeto receptor del mismo reciba eventos en todas sus entradas.

4.1.2. Modificaciones en la estructura de los objetos

Ya vimos en la definición formal del protocolo de Time Warp que no existía canal o enlace entre los objetos de una simulación optimista que intercambian eventos. El protocolo no se preocupa de qué objetos pueden enviarle eventos a un objeto dado, ya que es de naturaleza optimista: simula cada evento que llega, y en caso de que luego se reciba un evento de timestamp menor, se realiza un rollback.

Sin embargo, ya se demostró que el protocolo pesimista de Misra sí se preocupa por el grafo de comunicación entre objetos, el cual indica cuales objetos pueden enviar eventos a otros objetos. También se encarga de especificar un lookahead para cada objeto, de modo de implementar el envío de mensajes nulos y evitar de este modo la ocurrencia de deadlock.

Por lo tanto, nos vemos en la obligación de cambiar la estructura interna de un objeto en Warped, de modo que éste pueda conocer:

- qué objetos pueden enviarle eventos,
- a quien él podrá enviar eventos, y
- su lookahead.

4.1.3. Diseño e implementación de las modificaciones

Se define para cada objeto de la simulación las siguientes estructuras:

- Un arreglo para almacenar los identificadores de los procesos lógicos que enviarán eventos de entrada. Este arreglo es independiente de la cola de eventos de entrada original que el objeto tenía en Warped.
- Un arreglo para indicar los objetos que pueden recibir eventos de salida
- Un entero, para almacenar el lookahead del objeto

Se observa un esquema de esta nueva estructura de objeto en la Figura 4.2.

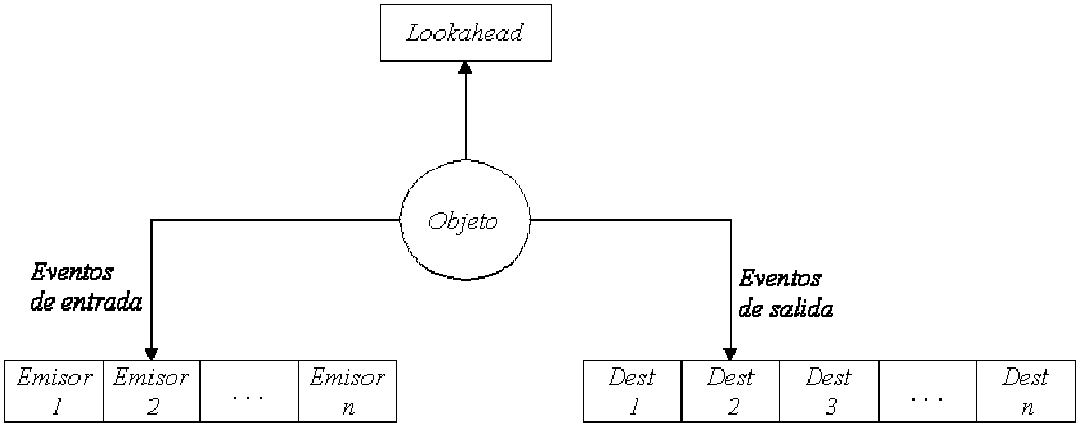


Figura 4.2 - Esquema de la nueva estructura de objeto en Warped

EVENTOS DE ENTRADA

El arreglo de eventos de entrada contiene, en cada posición, un identificador de objeto, el cual puede enviar un evento a este objeto en algún momento de la simulación. La finalidad de este arreglo es que el objeto sepa de qué otros objetos de la simulación debe esperar eventos de entrada.

Es posible especificar el largo máximo del arreglo mediante el uso de la variable MAX_MSG_INPUT en el archivo *config.hh*.

En base a esta cola modificamos la función *getEvent()*, a la cual cada objeto llama para recibir el próximo evento que lo tiene como destino. Debido a que la cola de eventos de entrada de un proceso lógico (y, por ende, de todos los objetos que pertenecen al mismo) tenía los eventos ordenados en base a su timestamp e identificador de objeto receptor, la función *getEvent()* simplemente debía devolver el primer evento de esta cola al objeto correspondiente. Las acciones que realiza esta función luego de las modificaciones propuestas en la estructura de un objeto (mostrada en la Figura 4.2) son:

- Se obtiene el objeto receptor del primer evento de la cola de eventos de entrada (original de Warped).
- Se recorre en forma lineal el arreglo de eventos de entrada para chequear si hay eventos de entrada de **todos** los objetos que pueden enviar eventos al objeto destino obtenido en el paso anterior (se obtiene la lista de objetos "antecesores" de la cola de entrada que estamos tratando).
- En caso de que el objeto tenga eventos de todos sus antecesores, la función getEvent devuelve al primer evento, de igual modo a como lo hacía originalmente en Warped. En caso contrario, devuelve *NULL*, ya que debido al protocolo conservador, se debe esperar a que todas las colas del objetos receptor estén con al menos un evento.

De este modo, logramos con una pequeña modificación que el kernel de Warped pueda devolver eventos al código de aplicación de acuerdo al protocolo de sincronización conservador.
Como todas las acciones descritas se realizan en el kernel, el código de la simulación no se entera de las mismas, por lo que no es necesario adaptar el código fuente de una simulación para que la misma ejecute bajo el protocolo de simulación pesimista o conservador.

EVENTOS DE SALIDA

El arreglo de eventos de salida contiene, en cada posición, un identificador de objeto destino, quien puede recibir un evento enviado por este objeto en algún momento de la simulación. La finalidad de este arreglo es que el objeto sepa a quien debe enviar eventos nulos durante la simulación con protocolo pesimista.
Es posible especificar el largo máximo del arreglo mediante el uso de la variable MAX_MSG_OUTPUT en el archivo *config.hh*.

LOOKAHEAD

Cada objeto almacena su lookahead, ya que se utiliza el mismo junto a la cola de eventos de salida para el envío de mensajes nulos a los objeto destino, para evitar la ocurrencia de deadlocks.

4.1.4. Formato del archivo de configuración

Debido a los cambios propuestos en warped para la implementación del protocolo de sincronización pesimista, surge la necesidad de poder especificar, para cada objeto de la simulación, el listado de objetos que podrían enviar o recibir mensajes del mismo, y su lookahead; estos valores son asignados en las nuevas estructuras de un objeto de Warped explicadas en la sección anterior.
Para lograr este fin el usuario deberá generar, en el directorio donde se halla el código de la simulación, un archivo de configuración específico. Se debe especificar el nombre del mismo en el archivo *config.hh*, junto a la constante CONFIG_FILE; por defecto, la misma está definida como:

Se debe completar el archivo de configuración de modo tal de incluir, para cada objeto de la simulación - en cualquier orden- el siguiente conjunto de líneas:
id_objeto
IN: in_1, in_2, ..., in_n
OUT: out_1, out_2, ..., out_n

LOOKAHEAD: *num*

donde: *id_objeto*, in_1, out_1 ($1 \leq i \leq n$) y *num* son enteros

Descripción de cada campo:
id_objeto: Número entero que identifica unívocamente al objeto en la simulación
IN: Lista de identificadores de objetos, separados por coma, que envían mensajes al objeto *id_objeto* en la simulación
OUT: Lista de identificadores de objetos, separados por coma, que son los destinos de los eventos que enviará el objeto *id_objeto* en la simulación
LOOKAHEAD: Lookahead del objeto *id_objeto*

4.1.5. Lectura del archivo de configuración en la aplicación

Se comentan a continuación los pasos a seguir para que el kernel pueda leer el archivo de configuración descrito en el párrafo anterior. Se incluyó en la creación de cada proceso lógico una llamada a una nueva función, llamada *readConfig*, la cual fue agregada al kernel con el objetivo de parsear el archivo de configuración descrito en la sección anterior. Esto implica que cada PL leerá su configuración al iniciar cada simulación, sin que el usuario deba preocuparse por incluir esta acción en el código de la simulación. La función parsea el archivo de configuración en busca de los datos del proceso lógico que la invocó, y lee los datos de los objetos que éste alberga, sin importar el orden en que se especifican los objetos.
Si al iniciar la simulación no se llegara a encontrar en este archivo de configuración todos los datos referentes a un objeto, o se llegara a detectar un error, la función de parsing informará por pantalla el error, y abortará la ejecución de la simulación.
Resumiendo, la función busca en el archivo de configuración los datos de los objetos que se instancian: las colas de entrada y salida de eventos, y su lookahead.

4.1.6. Deadlock: Implementación en Warped de mensajes nulos

Los eventos enviados entre los objetos pueden tener signo positivo o negativo. El kernel maneja en forma autónoma el signo de cada evento (positivo para eventos entre objetos y negativos para *antimesajes* usados en rollback), y el usuario no puede modificarlo.
Nuestra propuesta, que implementa el protocolo de simulación conservador, no precisará usar, en la ejecución de una simulación, los eventos negativos. Esto se debe a las características del protocolo, por el cual no pueden ocurrir rollbacks. Sin embargo, nos vemos en la necesidad de generar un nuevo tipo de evento, correspondiente a los eventos *nulos*. Mediante los mismos, cada objeto informa a sus posibles objetos destino que le llegó un evento, por lo que el próximo evento que les podría enviar tendrá un tiempo virtual de, al menos, el timestamp del evento recibido más su propio lookahead. De este modo se evita la posible generación de un deadlock en la simulación. Se implementa este tipo de eventos del mismo modo que se describe en el paper original de Misra [Mis86].

Al procesar un evento, puede ocurrir que el objeto decida enviar uno o más eventos a otro/s objeto/s de la simulación (dado que el proceso físico que simula realiza lo mismo en el sistema real); estos eventos podrían tener el mismo tiempo de recepción que el de los eventos *nulos* enviados por el objeto al principio de la simulación del evento actual. Por lo tanto, en las colas de salida correspondientes a estos nuevos eventos, habrá un evento *nulo* y, a continuación, otro de signo *positivo*, ambos con igual tiempo de recepción. En este caso, el evento *nulo* deja de tener sentido, ya que de nada sirve avisarle a un objeto que no se le enviará un evento con timestamp menor a *t* si a continuación se le está enviando un evento con tiempo *t*. No se llega a utilizar la información que llevaba el evento *nulo*, ya que el nuevo evento estará diciendo lo mismo que aquel: "no se le enviará al objeto destino un evento con tiempo menor a *t*". En consecuencia, cada vez que ocurra esta situación en una cola de salida de un objeto, se eliminará de la misma el evento *nulo*, dejando sólo el evento de signo *positivo* que estaba inmediatamente después del mismo.

Se puede observar un ejemplo de lo explicado anteriormente en la Figura 4.3. En la misma, el objeto 1, que tiene un lookahead de 3, recibe un evento de timestamp 10 por su único canal de entrada. Los objetos 2 y 3 son los posibles receptores de eventos de salida de este objeto. Al procesar el nuevo evento de timestamp 10, el objeto 1 envía un evento nulo por cada canal de salida, de timestamp 13 (debido al tiempo del evento más su propio lookahead). Tras procesar el evento, el objeto 1 envía un evento al objeto 2, que tiene timestamp de 13. En este caso, deja de tener sentido el envío del evento nulo enviado por ese canal, ya que tiene el mismo timestamp que el evento enviado. Por esta razón, se aniquila al evento nulo de la cola.

Es importante destacar que el kernel es el encargado de enviar eventos nulos por los canales de salida en el momento de procesar un nuevo evento; esta acción se desarrolla cuando la función *getEvent()* devuelve el nuevo evento al objeto que corresponde. Sin embargo, es la aplicación que se simula la que decide enviar uno o varios eventos al terminar de procesar el evento actual, dado que el proceso físico que simula así lo haría. Entonces, Warped no tiene manera de saber, a priori, si habrá en la cola de salida de un objeto un evento nulo y un evento positivo con igual timestamp, por lo que si se da este caso, se aniquila al evento nulo, tal como se explicó hasta aquí.

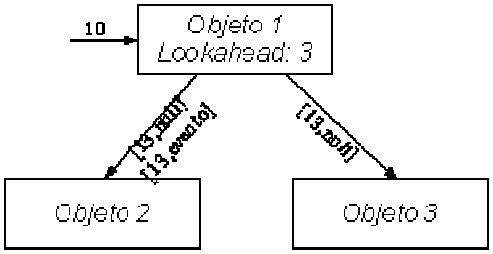


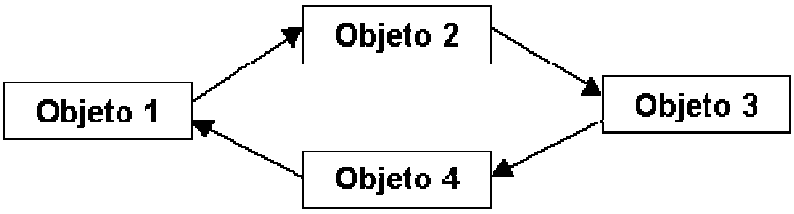
Figura 4.3 - Envío de mensajes nulos en Warped cuando simula bajo protocolo conservador

Por último, debido a la inclusión de eventos nulos, se debió modificar nuevamente al método *getEvent()*.

La primer tarea que realizaba, luego de la modificación introducida, era la de devolver a un objeto el evento con menor timestamp siempre y cuando todas sus colas de entrada estén ocupadas. Las últimas dos acciones que la función *getEvent()* realizaba, en caso de que devolvía un evento, eran las de actualizar el *LVT* del objeto receptor con el timestamp del evento que la función va a devolver, y devolver finalmente al nivel de aplicación dicho evento. Ahora, cuando están llenas todas las colas de entrada y el evento con menor timestamp es nulo, se debe actualizar de igual modo el *LVT* del objeto receptor del evento, pero, al ser este evento utilizado por el protocolo conservador como información de control para evitar deadlocks, no debe ser visible a nivel de usuario. Por lo tanto, la última acción de la función *getEvent()* pasa a ser la de chequear el signo del evento: si éste es positivo, se devuelve al nivel de aplicación; en caso contrario, no se devuelve ningún evento, aunque internamente el objeto actualizó su *LVT*.

4.1.7. Terminación de la simulación

La última modificación realizada al kernel de warped se debió a la condición de terminación de una simulación. Como se describió en la sección 3.3.10 (Detección de terminación de una simulación), el algoritmo original finaliza cuando circula por todos los objetos de la simulación (comenzando desde el objeto con *id* 0) un mensaje del tipo *CheckIdleMsg* que no sufre modificación en su campo *cancel*, lo cual implica que todos los objetos están ociosos. Recordamos que básicamente un objeto está ocioso cuando no está simulando evento alguno y no tiene eventos sin procesar en su cola de entrada. Sin embargo, debido a la inclusión en la simulación de eventos nulos, la condición anterior puede llegar a no ser válida. Como cada objeto envía un mensaje nulo a todos sus posibles destinos cada vez que puede procesar un mensaje de entrada, y estos mensajes nulos generarán otros con los siguientes objetos sucesores, se cumple en las simulaciones que contienen objetos formando un ciclo que, a pesar de que no circulen más mensajes positivos en la simulación, y ésta debiera ya haber concluido, lo único que ocurrirá es la circulación indefinida de mensajes nulos.



Podemos ver lo anterior en el ejemplo de la Figura 4.4, en donde cuatro objetos simulan el envío circular de un balón. Las flechas indican el flujo que debe seguir el balón en cada ciclo de la simulación.

Figura 4.4 - Esquema de la simulación de "ping pong" con cuatro objetos

Supongamos que hacemos circular sólo un balón en la simulación, y que cada objeto tiene un lookahead de 3. Se muestra en la Tabla 4-1 la secuencia de eventos de la simulación. Se agregan a la misma los eventos nulos. Recordar que si hay un evento nulo con igual tiempo de recepción y objeto destino que un evento positivo, el kernel borrará al evento nulo de la cola de salida. Se observa en el paso 8 que el objeto 1 recibe la bola (en tiempo 10), por lo que decide no continuar con el envío de la misma, ya que se debería finalizar la simulación. Sin embargo, en el mismo paso 8, el objeto 1 le comunica al objeto 2 que hasta un tiempo de 13 no le enviará un evento, debido a su lookahead. Del mismo modo el objeto 2 envía un evento nulo al objeto 3, y así sucesivamente. Al formar los objetos de la simulación un ciclo, circularán por el mismo eventos nulos en forma indefinida, a pesar de que la simulación debió haber finalizado en el paso 8. Es importante recordar que, cuando el kernel recibe este último evento que representa a la bola enviada por el objeto 4, no tiene forma de saber que luego no se continuará con el envío de la bola, ya que esta decisión se toma a nivel de usuario, es decir, al nivel de la aplicación que se simula. En consecuencia, nos vimos en la necesidad de reformular el método de control con el que Warped chequea si una simulación ha finalizado.

La modificación propuesta se basa en el esquema original pero propone que, cuando circula el mensaje de tipo *CheckIdleMsg* por los objetos de la simulación, se asumirá que un objeto está ocioso cuando:

- No está simulando evento alguno y
- No tiene eventos - **que no sean nulos** - sin procesar en su cola de entrada

Es importante notar que esta condición es casi equivalente a la original, ya que en el esquema optimista no se utilizaban mensajes nulos, por lo que alcanzaba con pedir que no haya eventos pendientes en la cola de entrada.
De este modo, queda solucionado el problema propuesto, ya que a partir del paso 9 todos los objetos cumplirán la condición anterior, y la simulación finalizará correctamente como lo hacía anteriormente.

Paso	Objeto 1 recibe	Objeto 1 envía	Objeto 2 recibe	Objeto 2 envía	Objeto 3 recibe	Objeto 3 envía	Objeto 4 recibe	Objeto 4 envía
1		(1,b1						
2			(1,b1	(4,null)				
3				(4,b1)				
4					(4,b1)	(7,null)		
5						(7,b1)		
6							(7,b1)	(10,null)
7								(10,b1)
8	(10,b1)	(13,null)						
9			(13,null)	(16,null)				
10					(16,null)	(19,null)		
11							(19,null)	(22,null)
12	(22,null)	(25,null)						
13			...	...				

Tabla 4-1 - Secuencia de eventos en la simulación de la Figura 4.4, donde cada objeto tiene un lookahead de 3

4.2. Implementación de protocolos IECP en Warped

4.2.1. Introducción

Warped es un kernel de simulación optimista. Esto implica que adopta una política de simulación agresiva en cada proceso lógico en el desarrollo de una simulación. Dado que los protocolos IECP están basados en modificaciones o mejoras de los protocolos agresivos u optimistas, es inmediato observar la viabilidad de la implementación de estos nuevos protocolos en el kernel existente. Bastará con agregar, para cada proceso lógico, en cada ciclo de la simulación, el cálculo de un error potencial (función M1), y en base a éste implementar la función M2. Esta puede resultar en simulación agresiva del PL, tal como lo estipulaba TimeWarp, por lo que no se deberá modificar el código de Warped, hasta simulación conservadora, en donde se deberá suspender por un lapso de tiempo la ejecución del proceso lógico, hasta que sea "seguro" que continúe.

4.2.2. Ciclo de simulación en Warped

Como se explicó dentro de sección 3.3.7 (Ciclo principal de una simulación), Warped ejecuta en una simulación, en cada proceso lógico y en forma cíclica, una misma serie de acciones. Las mismas se realizan en el método *LogicalProcess::simulate*, y constan de:

- Si es necesario, el GVT Manager realiza cálculos locales del GVT.
- El Communication Manager recibe mensajes del subsistema de mensajes y los distribuye a los objetos destino correspondientes.
- El Planificador (Scheduler) selecciona el siguiente objeto de la simulación para ejecutar.

Para incluir un protocolo IECP en el kernel, se debe incluir a la función M1 antes del paso 3. Luego, se inserta el código correspondiente a la función M2 dentro del mismo paso 3, ya que se trata de decidir si el próximo objeto simulará en forma agresiva, o es suspendido debido a una decisión de simulación conservadora.

4.2.3. Modelo Frecuencia de Rollback

Comenzaremos a diseñar a partir de aquí dos protocolos IECP basados en el modelo de frecuencia de rollback explicado en la sección 2.4.1. La diferencia entre ellos es que usan información local a cada proceso lógico o global de la simulación para adoptar decisiones de avance agresivas o conservadoras.

4.2.4. Modelo Frecuencia de Rollback Local

El primer protocolo que proponemos se basa exclusivamente en información local a cada proceso lógico. Esto significa que cada objeto de un proceso lógico adoptará decisiones agresivas o conservadoras de simulación en base a la cantidad de rollbacks que él mismo sufrió. En consecuencia, el esquema IECP propuesto es el que se detalla a continuación:

Función M1: Se calcula el error potencial (*EP*) de un objeto de un proceso lógico como la cantidad de rollbacks que dicho objeto sufrió desde un tiempo *T1* hasta el tiempo actual *T2, con T2-T1 <= T*, donde *T* es un intervalo de tiempo tras el cual se recomienza con la cuenta de rollbacks local, restaurando su valor en cero.

Función M2: Si la cantidad de rollbacks sufridos por el objeto en un intervalo de tiempo *T* predefinido es superior a un umbral dado, se suspende la simulación del objeto, adoptando de ese modo una política conservadora. Sin embargo, el PL que contiene al objeto seguirá aceptando a los eventos que envíen los vecinos del objeto (pueden ser objetos locales al PL o remotos), aunque estos no serán procesados hasta que el objeto simule. Si la cantidad de rollbacks no supera el umbral, se simula adoptando una política agresiva, tal como la adoptada en TimeWarp.

Esta definición genera la necesidad de incluir en el kernel un mecanismo mediante el cual se le pueda informar a cada PL de dos nuevos valores, *u* y *T.* Los mismos serán usados de igual modo por todos los objetos de un proceso lógico. Estos indican que un objeto simulará agresivamente hasta que alcance un número máximo o umbral de rollbacks *u* en un período *T* de tiempo.

Se muestra en Algoritmo 2 el pseudo código que implementa este protocolo en cada PL para cada ciclo de simulación en Warped. En el algoritmo se utilizan las variables *max_rollbacks* y *periodo* para referirse a los valores de umbral *u* y período *T* mencionados en el párrafo anterior.

Algoritmo 2 - Protocolo modelo frecuencia de rollbacks local en Warped

1. En cada proceso lógico, al inicio de la simulación:
   Leer cantidad máxima de rollbacks locales y el período de chequeo del archivo de configuración y almacenar estos valores en las variables *max_rollbacks y periodo*
2. En cada objeto, al comienzo de la simulación
   tiempo_previo = 0
3. En cada objeto, cada vez que el planificador del proceso lógico lo invoca para ejecutar
   tiempo_actual = Warped.*TiempoTotalDeSimulacion()*
   Si (*tiempo_actual - tiempo_previo >= periodo)*
      simularProxEvento()
      tiempo_previo = tiempo_actual
      cantidad_rollbacks = 0
   Si no
      Si (*cantidad_rollbacks < max_rollbacks*)
         simularProxEvento()
   /*Si no, no se realiza ninguna acción, ya que se suspende la ejecución del objeto*/

Nota: El kernel Warped tiene implementada una función que realiza la tarea descrita aquí para Warped.TiempodeSimulacion()

Al comienzo de la simulación, se debe leer del archivo de configuración el valor máximo permitido de rollbacks, y el período de tiempo en el que se realiza el chequeo de este umbral, tras el cual se vuelve a restaurar el número de rollbacks local en cero y se recomienza el ciclo. Mediante la inclusión del mecanismo de frecuencia de rollback local, en cada ciclo de la simulación se simulará en un PL a un objeto con evento recibido con menor timestamp (tal como se hacía en Warped) si además cumple que la cantidad de rollbacks que sufrió dentro de un período *T* no supera al umbral *u*; si este fuera el caso, se suspenderá al objeto hasta que comience un nuevo período de tiempo *T*, ya que se restaura el número local de rollbacks sufridos a cero.

Con el fin de que un proceso lógico pueda simular objetos que no deben ser demorados, se modificó la rutina del planificador para escoger el próximo objeto a simular. En lugar de tomar al primer objeto receptor de la lista de eventos de entrada del proceso lógico, se simula al mismo si sólo está en condiciones, es decir, no debe suspendérselo; caso contrario, se chequea con el siguiente objeto receptor, y así sucesivamente, hasta que algún objeto del proceso lógico esté en condiciones de simular o se haya finalizado el recorrido por la cola de eventos de entrada.

FORMATO DEL ARCHIVO DE CONFIGURACIÓN

Para indicarle a una simulación de que deberá ejecutarse respetando el protocolo Modelo Frecuencia de Rollback Local, se deberá incluir un archivo de configuración en el mismo directorio donde se lanza la simulación, de nombre local_rfm.config. El contenido del mismo debe ser:

max: x

rfm_interval: y

donde *x* es el umbral de rollbacks, *y* es el período de tiempo en segundos

Ejemplo:

max: 10

rfm_interval: 0.05

4.2.5. TimeWarp Modelo Frecuencia de Rollbacks Local pos Pasos (MFR Local Step)

En base a los resultados que se obtuvieron con las simulaciones en algunas simulaciones con Modelo Frecuencia de Rollbacks Local, se observó que los tiempos obtenidos no eran buenos porque algunos objetos perdían más tiempo suspendiéndose que el tiempo que les hubiera llevado hacer los rollbacks y continuar con la simulación con TimeWarp. Esto se debe a que cuando un objeto decide suspenderse, porque llegó a un umbral de rollbacks, lo hace hasta que finaliza el tiempo definido como intervalo de chequeo, tras el cual se reinician a cero los contadores de rollbacks. Entonces, la idea es que cuando un objeto se suspende lo haga por una cantidad fija (llamémosla N) de ciclos de simulación, tras lo cual el objeto reinicia sus contadores y comienza nuevamente con el control de rollbacks. De este modo no se corre el riesgo de que si un objeto se suspende al comienzo de un intervalo *t*, no esté sin simular durante casi todo el tiempo *t*, si no tan sólo N ciclos de simulación en warped.
En todas las simulaciones de este trabajo se ejecutó con N=10 ciclos de simulación.

4.2.6. Modelo Frecuencia de Rollback Global

Proponemos aquí un protocolo similar al anterior, pero utiliza información "global" para determinar en cada PL si éste adoptará decisiones agresivas o conservadoras de simulación. En lugar de suspender la simulación de eventos de un objeto cuando la cantidad de rollbacks que realizó superaron un umbral, se tomara esta decisión cuando su número de rollbacks sea el mayor respecto al número de rollbacks de todos los demás PLs de la simulación.
Se detalla a continuación el esquema propuesto:

Función M1: Se calcula el error potencial (*EP*) de un proceso lógico como la diferencia entre la cantidad de rollbacks que sufrió dicho PL y el número mayor de rollbacks que sufrieron los demás PLs de la simulación. Este cálculo se realiza desde un tiempo *T1* hasta el tiempo actual *T2, con T2-T1 <= T*, donde *T* es un intervalo de tiempo tras el cual se recomienza con la cuenta de rollbacks local, restaurando su valor en cero.

Función M2: Si la cantidad de rollbacks sufridos por el PL en un intervalo de tiempo predefinido es superior al de todos los demás PLs de la simulación (es decir, *EP* > 0), se suspende la simulación del PL, adoptando de ese modo una política conservadora (sin embargo, este PL seguirá aceptando los eventos que le envíen sus PL vecinos, aunque no procese evento alguno). Si no, simular adoptando una política agresiva, tal como la adoptada originalmente en TimeWarp.

Se deduce de aquí que se deberá incluir en Warped un mecanismo mediante el cual se pueda informar a cada PL del intervalo de tiempo *T*. Un PL tomará a *T* como ventana de tiempo para comparar su cantidad local de rollbacks contra la de los demás PLs; tras ese intervalo, se reinicializarán todas estas cantidades en cero, comenzando nuevamente el proceso de comparación para el nuevo intervalo.

Una modificación significativa respecto al protocolo MFR Local es que, al final de cada ciclo de la simulación, cada PL deberá informar a los demás sobre la cantidad de rollbacks que sufrieron todos los objetos de la simulación que pertenecen al mismo. En base a lo anterior y a los detalles de implementación de Warped ya explicados, proponemos el pseudo código que implementa el protocolo de MFR Global en el Algoritmo 3.

Algoritmo 3 - Protocolo modelo frecuencia de rollbacks global en Warped

1. En cada proceso lógico, al inicio de la simulación:
   Leer el período de chequeo del archivo de configuración y almacenarlo en la variable *periodo*
2. En cada objeto, al comienzo de la simulación

tiempo_previo = 0

max_rollbacks = 0

3. En cada objeto, cada vez que el planificador del proceso lógico lo invoca para ejecutar

tiempo_actual = Warped.*TiempoTotalDeSimulacion()*

Si (*tiempo_actual - tiempo_previo >= periodo)*

simularProxEvento()

tiempo_previo = tiempo_actual

cantidad_rollbacks = 0

Si no

Si (*cantidad_rollbacks* < *max_rollbacks*)

simularProxEvento()

/*Si no, no se realiza ninguna acción, ya que se suspende la ejecución del objeto*/

4. Para i desde 1 hasta la cantidad de procesos lógicos de la simulación

Si (i es distinto del id de este PL)

Enviar al PL i la cantidad de rollbacks de todos nuestros objetos

5. Rutina que recibe la cantidad de rollbacks de los demás procesos lógicos:

Para j desde 1 hasta la cantidad de números recibidos

Si (rollbacks[j] > *max_rollbacks*)

max_rollbacks = rollbacks[j]

Nota: El kernel Warped tiene implementada una función que realiza la tarea descrita aquí para Warped.TiempodeSimulacion()

Es importante destacar en el algoritmo que la función que cumple ahora la variable *max_rollbacks* es similar a la que cumplía la variable *u* para el algoritmo MFR local; en

lugar de comparar la cantidad de rollbacks local contra un umbral, se compara contra el número máximo de rollbacks que se recibió de los restantes PLs.

FORMATO DEL ARCHIVO DE CONFIGURACIÓN

Para indicarle a una simulación de que deberá ejecutarse respetando el protocolo Modelo Frecuencia de Rollback Local, se deberá incluir un archivo de configuración en el mismo directorio donde se lanza la simulación, de nombre local_rfm.config. El contenido del mismo debe ser:

rfm_interval: x

donde *x* es el período de tiempo en segundos

Ejemplo:

rfm_interval: 0.2

5. Ejecución de simulaciones y análisis de resultados

5.1. Arquitecturas utilizadas

Para poder comparar los protocolos de simulación descritos en el presente trabajo, como así también los nuevos propuestos, se programaron varias simulaciones de diversos sistemas de la vida real. Se ejecutaron las simulaciones en dos entornos de arquitecturas bien diferenciados:

1) Red de cinco equipos Sun netra1. 512Mb de RAM. Sistema operativo: Solaris 2.7. Se ejecutaron las simulaciones en redes Ethernet de 10Mhz y 100Mhz.

2) Red de cinco equipos Linux. La distribución de Linux utilizada fue RedHat 6.2. Se ejecutaron las simulaciones en una red Ethernet de 10Mhz. La descripción de cada equipo es la siguiente:

1) Pentium III 450Mhz, 64Mb RAM
3) Pentium III 500Mhz, 128Mb RAM
3) AMD K6 233Mhz, 64Mb RAM
4) Pentium 266Mhz, 64Mb RAM
5) 486 DX/2, 16Mb RAM

En las simulaciones con equipos Linux que se evaluarán en el presente trabajo, identificaremos a cada equipo por el número dado en esta lista. Ejemplo: TimeWarp 1-5 indica la ejecución de una simulación con el protocolo de TimeWarp entre los equipos 1 (Pentium III 450Mhz) y 5 (486 DX/2).
La idea es emplear siempre a los equipos más rápidos e ir agregando a los más lentos a medida que precisemos ejecutar la simulación con más procesadores. Esto indica que las simulaciones con mas procesadores en la red Linux van a ser mas lentas, pero

el presente trabajo busca comparar los distintos protocolos de simulación (por lo que a iguales condiciones de ejecución los tests son válidos), y no en ver cuánto se gana en el paralelismo con mas procesadores.

5.2. Simulación de Ping-pong

Esta simulación consta de *N* objetos. El objeto 0 envía una “bola” al objeto 1, éste la reenvía al objeto 2, y así siguiendo. El objeto *N-1* reenvía la bola al objeto 0, haciendo de ese modo que la bola realice un recorrido circular. Sólo cuando el objeto 0 recibe la bola enviará una nueva al objeto 1. Se simulan tantos ciclos como se desee, siendo siempre el objeto 0 el que comienza y detiene la circulación de la bola. Se llama a esta simulación como “ping-pong”, debido a que si sólo dos objetos formaran parte de la misma, la simulación se asemejaría a una partida de este juego.

5.2.1. Configuración de la simulación

Se define en el archivo *ping.config*, ubicado en el mismo directorio de la simulación, la cantidad de objetos, el número de bolas que serán enviadas por el objeto 0 y la cantidad de procesos lógicos en los que se divide la simulación.
Los parámetros de la simulación utilizados son: 600 objetos, 100 eventos, 2, 3, 4 y 5 procesos lógicos.

Es posible compilar la simulación para que ejecute según una “mejor partición” o “peor partición” de los objetos:

- Mejor partición: Se minimiza la comunicación de los procesos lógicos a través de la red. Si la simulación consta de *n* objetos y *p* procesos lógicos, el proceso lógico 0 contendrá a los objetos numerados desde el 0 al n/p - 1, el PL2 contendrá a los

objetos n/p hasta 2*(n/p) - 1, y así siguiendo, almacenando en el último PL tantos objetos como n/p + n%p. De este modo sólo habrá comunicación entre los diversos procesos lógicos (o sea, eventos a través de la red) cuando el objeto n/p-1 envíe un evento al objeto n/p, cuando el objeto 2* n/p - 1 envíe un evento al objeto 2*n/p, y así sucesivamente.

- Peor partición: Se maximiza la comunicación entre procesos lógicos. Se ubica al objeto 0 en el PL0, al objeto 1 en el PL1, al objeto p-1 en el proceso lógico p, al objeto p en el objeto 0, y así siguiendo. De este modo, todos los eventos que se envíen los objetos serán a través de la red, y nunca quedarán en un mismo proceso lógico.

5.2.2. Resultados obtenidos

♦ Ping-pong best partition, equipos Sun netra, red de 10 Mhz

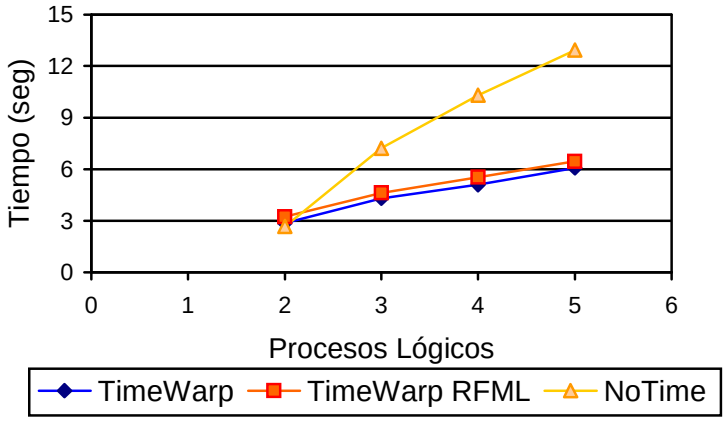
2 Procesos Lógicos	Promedio	Mejor Tiempo	Peor Tiempo
TimeWarp	2.87	2.83	2.95
TimeWarp RFML	3.23	3.20	3.26
NoTime	2.66	2.59	2.71

3 Procesos Lógicos	Promedio	Mejor Tiempo	Peor Tiempo
TimeWarp	4.30	4.28	4.36
TimeWarp RFML	4.62	4.62	4.64
NoTime	7.21	7.21	7.34

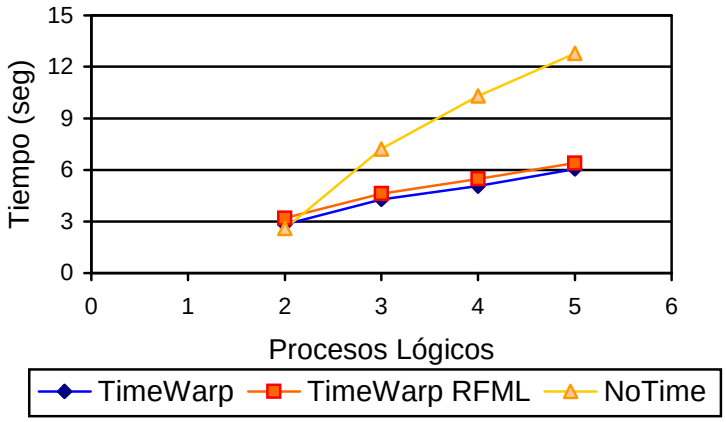
4 Procesos Lógicos	Promedio	Mejor Tiempo	Peor Tiempo
TimeWarp	5.10	5.06	5.13
TimeWarp RFML	5.54	5.47	5.67
NoTime	10.30	10.30	10.41

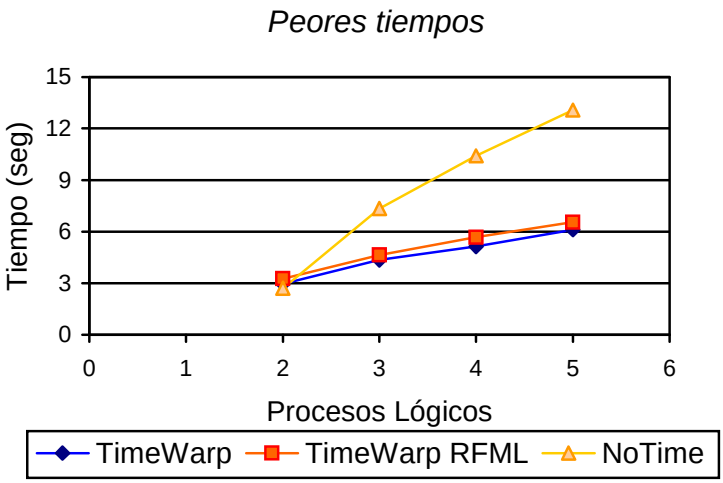
5 Procesos Lógicos	Promedio	Mejor Tiempo	Peor Tiempo
TimeWarp	6.07	6.05	6.12
TimeWarp RFML	6.46	6.41	6.54
NoTime	12.94	12.79	13.07

Promedio de tiempos



Mejores tiempos




Peores tiempos
15
12
9
6
3
0
Tiempo (seg)
0
1
2
3
4
5
6
Procesos Lógicos
TimeWarp
TimeWarp RFML
NoTime

- Ping-pong worst partition, equipos Sun netra, red de 10 Mhz

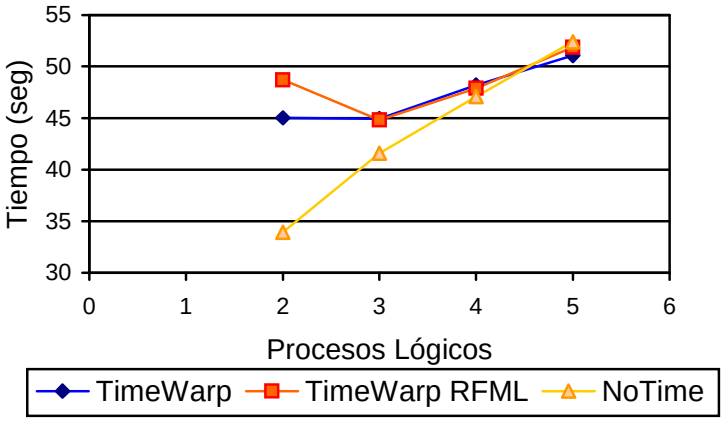
2 Procesos Lógicos	Promedio	Mejor Tiempo	Peor Tiempo
TimeWarp	45.03	44.29	46.42
TimeWarp RFML	48.71	48.41	49.02
NoTime	33.93	33.78	34.10

3 Procesos Lógicos	Promedio	Mejor Tiempo	Peor Tiempo
TimeWarp	44.94	44.74	45.15
TimeWarp RFML	44.81	44.81	45.33
NoTime	41.58	41.43	41.69

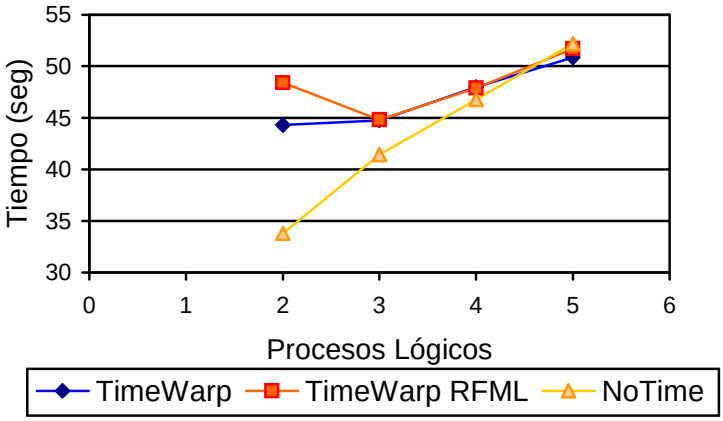
4 Procesos Lógicos	Promedio	Mejor Tiempo	Peor Tiempo
TimeWarp	48.20	48.00	48.43
TimeWarp RFML	47.90	47.90	48.47
NoTime	47.09	46.79	47.30

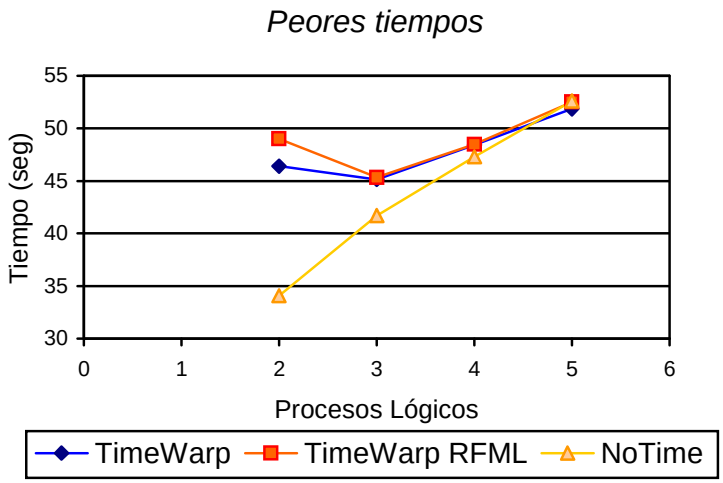
5 Procesos Lógicos	Promedio	Mejor Tiempo	Peor Tiempo
TimeWarp	51.06	50.85	51.86
TimeWarp RFML	51.90	51.70	52.52
NoTime	52.41	52.19	52.60

Promedio de tiempos



Mejores tiempos





- Ping-pong best partition, equipos Sun netra, red de 100 Mhz

1 Procesos Lógicos	Promedio	Mejor Tiempo	Peor Tiempo
Secuencial	0.38	0.37	0.39

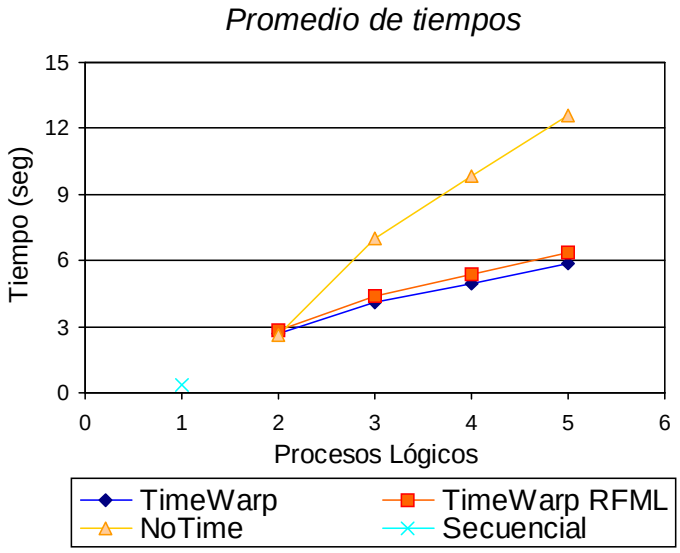
2 Procesos Lógicos	Promedio	Mejor Tiempo	Peor Tiempo
TimeWarp	2.66	2.65	2.67
TimeWarp RFML	2.84	2.81	2.90
NoTime	2.59	2.57	2.63

3 Procesos Lógicos	Promedio	Mejor Tiempo	Peor Tiempo
TimeWarp	4.11	4.08	4.13
TimeWarp RFML	4.37	4.37	4.37
NoTime	6.97	6.97	6.98

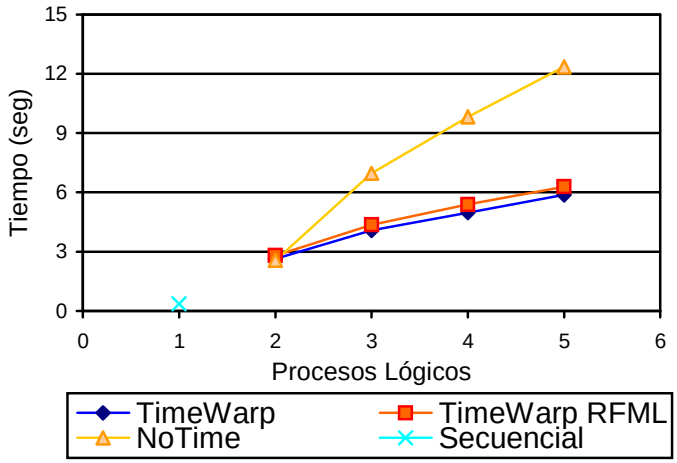
4 Procesos Lógicos	Promedio	Mejor Tiempo	Peor Tiempo
TimeWarp	4.98	4.98	5.02

TimeWarp RFML	5.41	5.39	5.43
NoTime	9.82	9.82	9.93

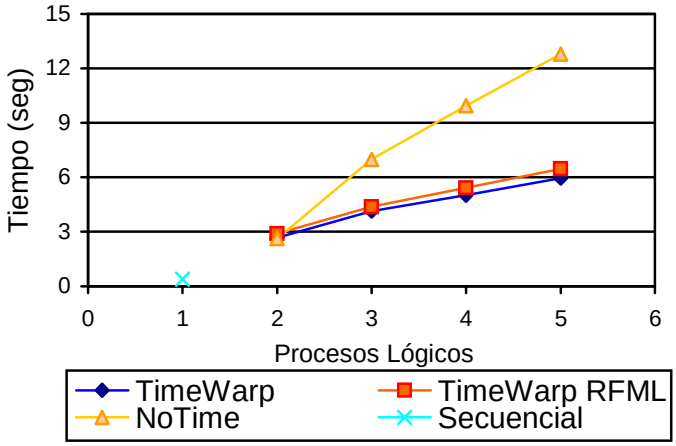
5 Procesos Lógicos	Promedio	Mejor Tiempo	Peor Tiempo
TimeWarp	5.90	5.88	5.95
TimeWarp RFML	6.39	6.29	6.47
NoTime	12.58	12.35	12.78



Mejores tiempos



Peores tiempos



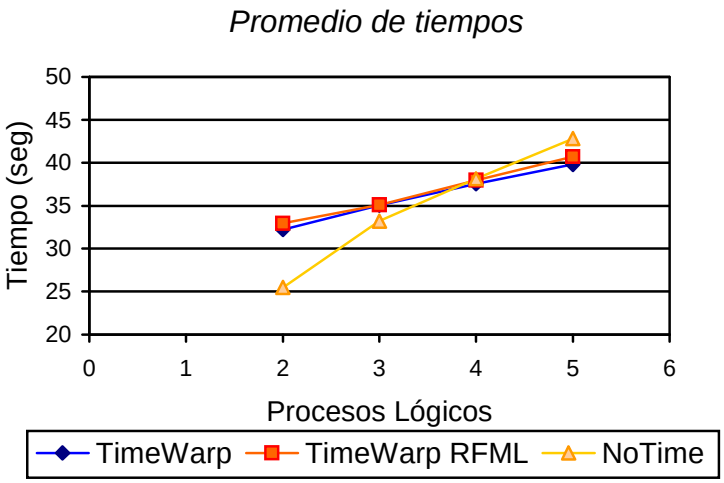
- Ping-pong worst partition, equipos Sun netra, red de 100 Mhz

2 Procesos Lógicos	Promedio	Mejor Tiempo	Peor Tiempo
TimeWarp	32.23	30.65	33.37
TimeWarp RFML	32.96	32.84	33.07
NoTime	25.47	25.32	25.68

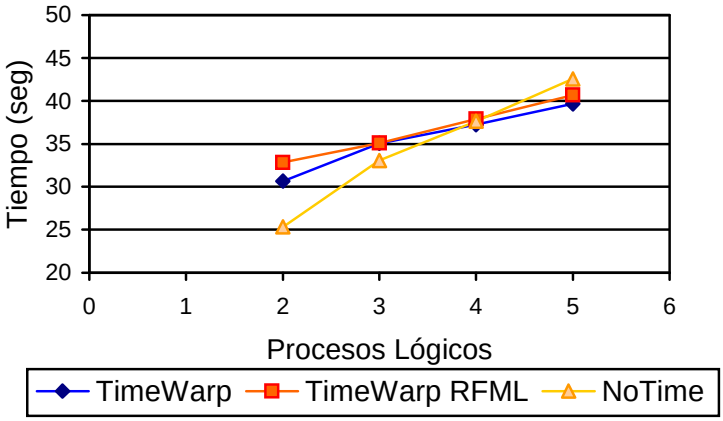
3 Procesos Lógicos	Promedio	Mejor Tiempo	Peor Tiempo
TimeWarp	35.07	35.07	35.36
TimeWarp RFML	35.10	35.10	35.36
NoTime	33.22	33.07	33.34

4 Procesos Lógicos	Promedio	Mejor Tiempo	Peor Tiempo
TimeWarp	37.56	37.23	37.76
TimeWarp RFML	37.95	37.90	38.10
NoTime	38.14	37.64	38.65

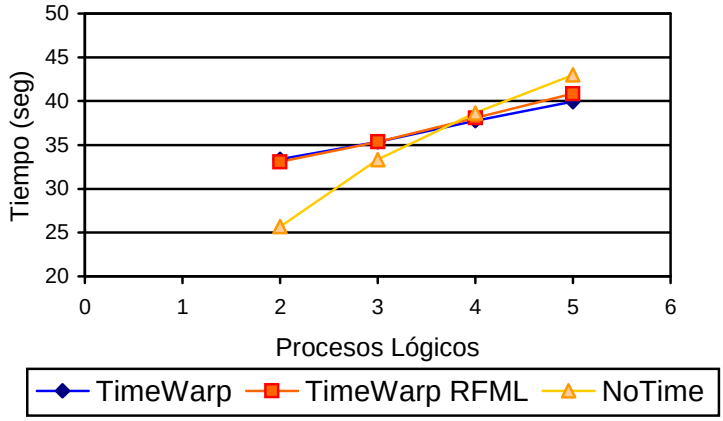
5 Procesos Lógicos	Promedio	Mejor Tiempo	Peor Tiempo
TimeWarp	39.80	39.65	39.93
TimeWarp RFML	40.72	40.69	40.87
NoTime	42.83	42.56	43.01



Mejores tiempos



Peores tiempos



- Ping-pong best partition, equipos Linux, red de 10 Mhz

1 Procesos Lógicos	Promedio	Mejor Tiempo	Peor Tiempo
Secuencial 1	0.23	0.18	0.29
Secuencial 5	3.78	3.65	3.88

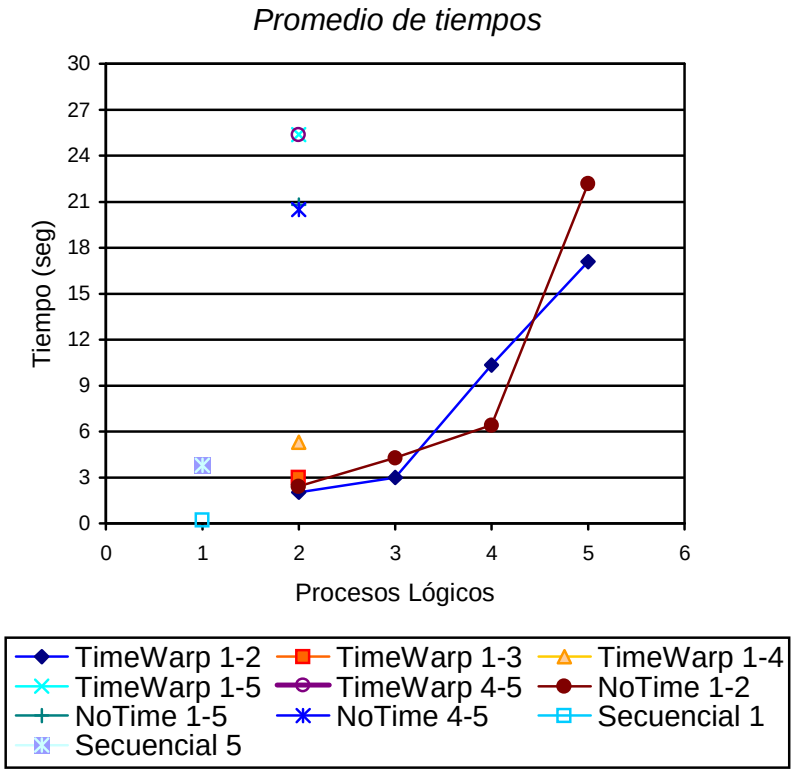
2 Procesos Lógicos	Promedio	Mejor Tiempo	Peor Tiempo

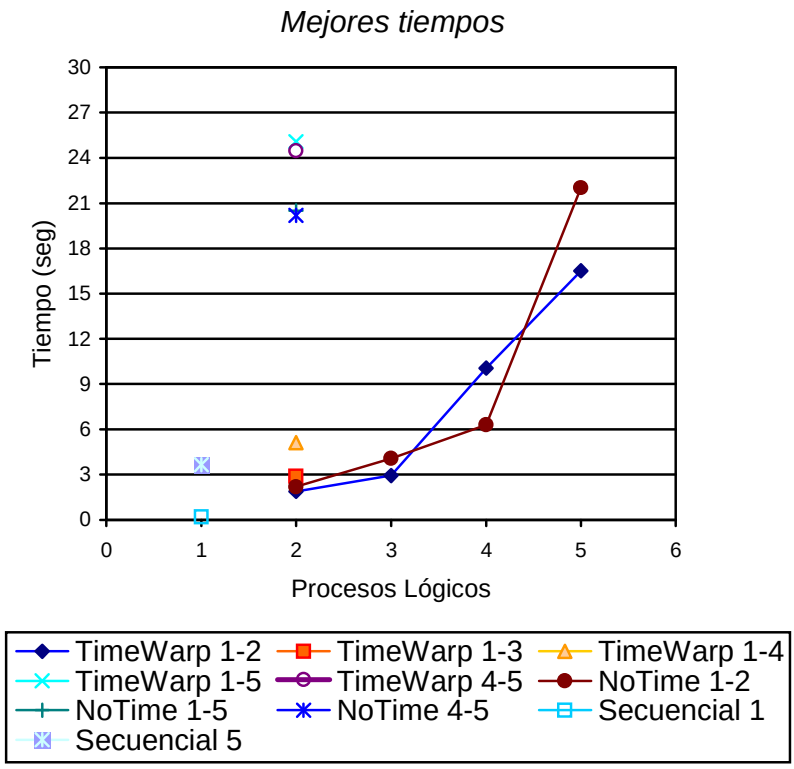
TimeWarp 1-2	2.03	1.88	2.22
TimeWarp 1-3	3.00	2.90	3.17
TimeWarp 1-4	5.28	5.12	5.58
TimeWarp 1-5	25.39	25.09	25.70
TimeWarp 4-5	25.38	24.48	26.22
NoTime 1-2	2.42	2.18	2.76
NoTime 1-5	20.78	20.49	20.94
NoTime 4-5	20.49	20.18	20.65

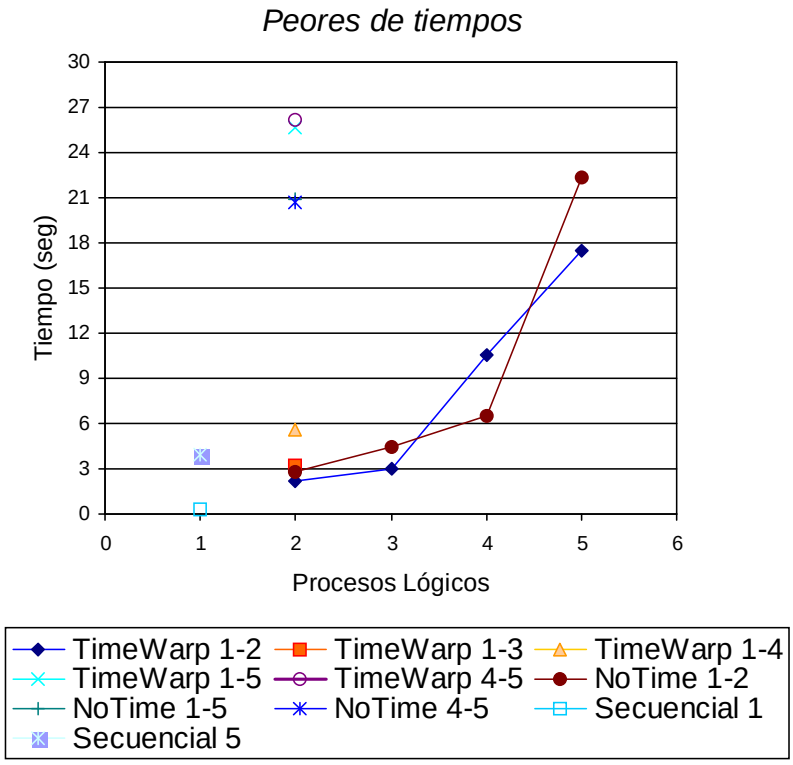
3 Procesos Lógicos	Promedio	Mejor Tiempo	Peor Tiempo
TimeWarp 1-2	2.98	2.93	3.04
NoTime 1-2	4.28	4.07	4.40

4 Procesos Lógicos	Promedio	Mejor Tiempo	Peor Tiempo
TimeWarp 1-2	10.33	10.07	10.56
NoTime 1-2	6.40	6.31	6.48

5 Procesos Lógicos	Promedio	Mejor Tiempo	Peor Tiempo
TimeWarp 1-2	17.09	16.51	17.44
NoTime 1-2	22.18	22.02	22.30


Promedio de tiempos
Tiempo (seg)
30
27
24
21
18
15
12
9
6
3
0
0
1
2
3
4
5
6
Procesos Lógicos
TimeWarp 1-2
TimeWarp 1-3
TimeWarp 1-4
TimeWarp 1-5
TimeWarp 4-5
NoTime 1-2
NoTime 1-5
NoTime 4-5
Secuencial 1
Secuencial 5


Mejores tiempos
Tiempo (seg)
30
27
24
21
18
15
12
9
6
3
0
0
1
2
3
4
5
6
Procesos Lógicos
TimeWarp 1-2
TimeWarp 1-3
TimeWarp 1-4
TimeWarp 1-5
TimeWarp 4-5
NoTime 1-2
NoTime 1-5
NoTime 4-5
Secuencial 1
Secuencial 5


Peores de tiempos
Tiempo (seg)
30
27
24
21
18
15
12
9
6
3
0
0
1
2
3
4
5
6
Procesos Lógicos
TimeWarp 1-2
TimeWarp 1-3
TimeWarp 1-4
TimeWarp 1-5
TimeWarp 4-5
NoTime 1-2
NoTime 1-5
NoTime 4-5
Secuencial 1
Secuencial 5

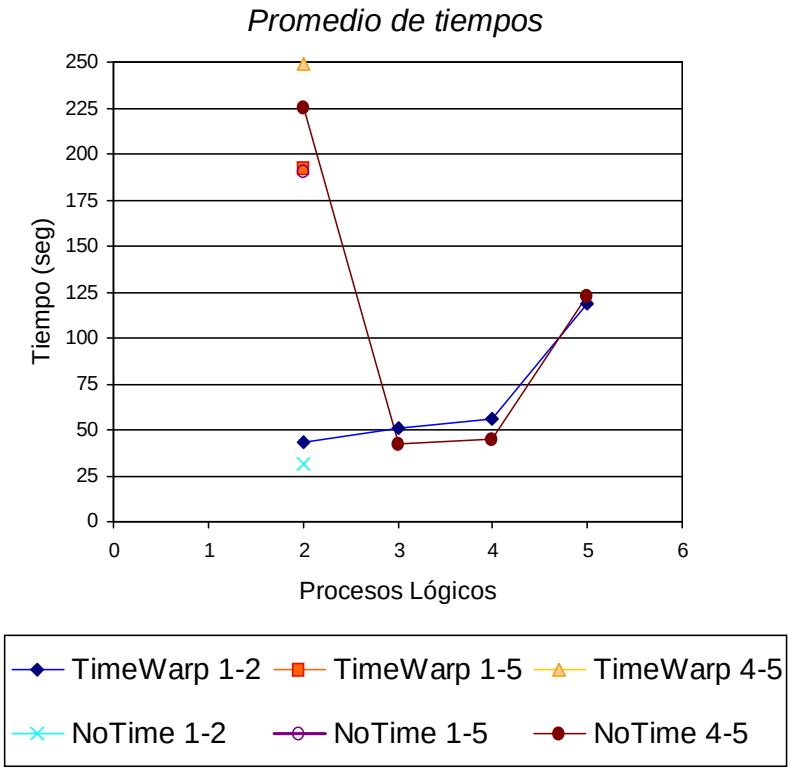
- Ping-pong worst partition, equipos Linux, red de 10 Mhz

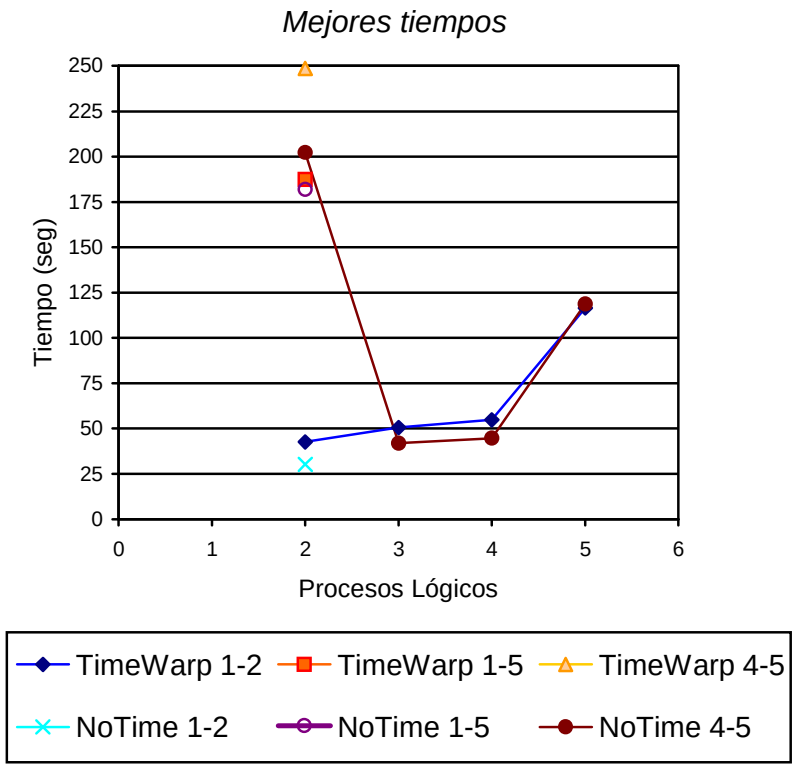
2 Procesos Lógicos	Promedio	Mejor Tiempo	Peor Tiempo
TimeWarp 1-2	43.01	42.67	43.52
TimeWarp 1-5	192.17	187.34	194.94
TimeWarp 4-5	249.55	248.71	251.20
NoTime 1-2	31.35	30.21	32.48
NoTime 1-5	190.30	181.77	195.74
NoTime 4-5	225.73	202.21	263.72

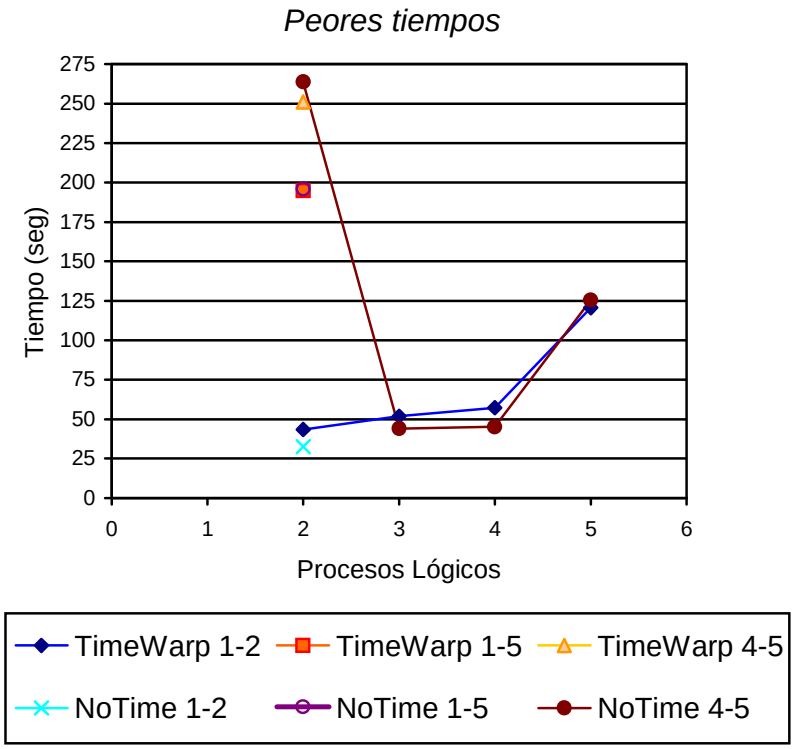
3 Procesos Lógicos	Promedio	Mejor Tiempo	Peor Tiempo
TimeWarp	50.97	50.45	51.90
NoTime	42.77	41.89	44.07

4 Procesos Lógicos	Promedio	Mejor Tiempo	Peor Tiempo
TimeWarp	55.76	54.75	57.31
NoTime	44.83	44.63	45.21

5 Procesos Lógicos	Promedio	Mejor Tiempo	Peor Tiempo
TimeWarp	118.67	116.46	120.65
NoTime	123.04	118.69	125.55


Promedio de tiempos
Tiempo (seg)
250
225
200
175
150
125
100
75
50
25
0
0
1
2
3
4
5
6
Procesos Lógicos
TimeWarp 1-2
TimeWarp 1-5
TimeWarp 4-5
NoTime 1-2
NoTime 1-5
NoTime 4-5


Mejores tiempos
250
225
200
175
150
125
100
75
50
25
0
Tiempo (seg)
0
1
2
3
4
5
6
Procesos Lógicos
TimeWarp 1-2
TimeWarp 1-5
TimeWarp 4-5
NoTime 1-2
NoTime 1-5
NoTime 4-5


Peores tiempos
275
250
225
200
175
150
125
100
75
50
25
0
Tiempo (seg)
0
1
2
3
4
5
6
Procesos Lógicos
TimeWarp 1-2
TimeWarp 1-5
TimeWarp 4-5
NoTime 1-2
NoTime 1-5
NoTime 4-5

5.2.3. Observaciones y conclusiones

- Ninguna simulación incurrió en rollbacks, ya que la simulación es totalmente secuencial. Todos los objetos reciben eventos sólo de su inmediato antecesor, y cómo éstos siempre le entregan los eventos ordenados por tiempo, nunca reciben un evento en “el pasado”.
- Debido a la ausencia de rollbacks se observa un mejor comportamiento de las simulaciones de TimeWarp contra las de Modelo Frecuencia de Rollback Local. Esto se debe a que éste último es una modificación del protocolo de TimeWarp, en donde se agrega un overhead para comparar los rollbacks que sufrió un objeto, para luego analizar su suspensión. A pesar de que este overhead de cálculo es pequeño (sobre todo respecto a las demoras de la red, como luego veremos), hace que las simulaciones MFRL demoren un poco más que las de TimeWarp puro.
- Las simulaciones con el protocolo de NoTime mostraron tener un inicio más lento que aquellas con TimeWarp. Luego, como esta simulación no tuvo rollbacks, y al ser relativamente pequeña la cantidad de eventos a simular y por ende la cantidad de estados a salvar, esta técnica no pudo obtener tiempos menores que las simulaciones de TimeWarp. De todos modos es destacable que los tests bajo este protocolo se diferenciaron más en base a la arquitectura utilizada, es decir, las redes de equipos Sun frente a la red Linux.
- Las diferencias en los tiempos obtenidos entre los casos “best partition” y “worst partition” son abismales. A pesar de tratarse de la misma simulación, y que la cantidad total de eventos simulados fuera la misma, en el peor caso las simulaciones tardaron al menos ocho veces más que en el de mejor caso.
- Por otra parte, se obtuvieron tiempos sensiblemente menores con la red de 100Mhz que en la red de 10Mhz (redes de equipos Sun). Esto se observa sobre todo en el caso de “peor partición”, en donde la comunicación por la red se torna crucial, ya que todos los eventos son transmitidos a través de la misma. Con la red de 100Mhz

se observaron mejoras en los tiempos de entre 25 y 30%. En el caso de “mejor partición”, al haber una participación mucho menor de la red, las diferencias de tiempos entre las redes de 10Mhz y 100Mhz son, en algunos casos, imperceptibles.

- Mientras se ejecutaron las simulaciones (tanto en una red de 10Mhz como en una de 100Mhz), el led de colisiones del hub que conectaba en red a todos los equipos de la simulación, destelló de modo ininterrumpido. Esto se debe a que las librerías de comunicación de red, mpich (que implementan el estándar MPI, Message Passing Interface) precisan una comunicación constante entre todos los procesos lógicos de la simulación.
- Esto demuestra claramente que el cuello de botella para explotar el paralelismo de la simulación es la red Ethernet. Usar como red un medio compartido, en lugar de alguno orientado, por ejemplo, a tokens, hace que haya demasiadas colisiones en la red, demorando de modo notable el avance de la simulación. Por esta razón los creadores de Warped han experimentado con varios tipos de redes de computadoras, y han obtenido mejores resultados en redes ATM.
- En la red Linux, hubo mucha diferencia en los tiempos de ejecución de Time Warp entre 2 maquinas rápidas o entre dos máquinas lentas. En el caso de best-partition, éstas llegaron a tardar 12 veces más que las primeras. En el caso de worst-partition, la pareja de equipos lentos demoraron hasta 6 veces más en ejecutar la misma simulación, ya que el efecto de lentitud de un procesador se mostró “suavizado” por el efecto de la lentitud de la red, que en esta simulación es crucial.
- Con best-partition, cuando se simuló con los equipos 1 y 5 (el más rápido y más lento) o con los dos más lentos (equipos 4 y 5) no hubo una diferencia significativa de tiempos, ya que el equipo más lento fue el que marcó el ritmo de la simulación, siendo así el cuello de botella de la misma. En cambio, lo anterior cambia para worst-partition, en donde la inclusión de un equipo rápido cambió los tiempos respecto a ejecutar con 2 equipos lentos.
- Lo dicho en los dos párrafos anteriores demuestra que la poca eficiencia de un procesador ralentiza a toda una simulación. Esto nos indica que no siempre es

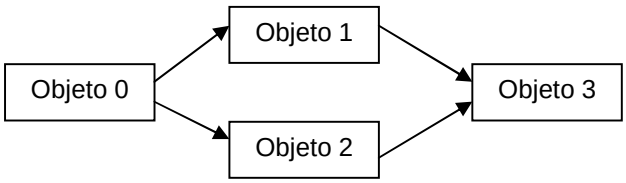
beneficioso incluir un equipo más en la red para ejecutar una simulación, si éste equipo tiene mucho peor performance que el promedio general de los restantes equipos.

- Las simulaciones con los 2 equipos más rápidos en la red Linux arrojaron mejores resultados que con 2 equipos en la red Sun (en redes de 10Mhz), demostrando que los procesadores Pentium III pueden llegar a tener mejor rendimiento que los de las netra.
- Se obtuvo un excelente tiempo de ejecución con el protocolo secuencial en uno de los equipos Linux más rápidos, aunque no así con el equipo más lento. Dado que la red Ethernet es el cuello de botella para explotar el paralelismo de la simulación, observamos que con un equipo potente la ejecución secuencial arrojó mejores tiempos que las simulaciones con varios equipos.

5.3. Simulación Cartas

Aquí se desea simular a una empresa que envía cartas con publicidad a sus clientes. Para tal fin existe un generador de cartas (objeto 0), dos entidades que se encargan de supervisar y corregir las mismas antes de ser enviadas (objetos 1 y 2) y finalmente el objeto que se encarga de la distribución final del correo (objeto 3). El objeto 0 envía de a una carta por vez, en forma alternada, a los objetos 1 y 2; se envían en total, a partir del tiempo 10, tantas cartas (eventos en la simulación) como se definen en el archivo de configuración de la simulación.

Se muestra en la siguiente figura el esquema de la simulación:



Se asignó un objeto a cada proceso lógico de la simulación, por lo que la misma fue ejecuta en cuatro procesadores.

Nota: Nuestra implementación de modelo pesimista o conservador hace que un objeto envíe eventos nulos a sus vecinos sólo cuando recibe un evento. Por esta razón se debió modificar ligeramente el código fuente de esta simulación, antes de compilarla para ejecutar bajo el protocolo pesimista, haciendo que el objeto 0 le envíe a los objetos 1 y 2 un evento nulo con tiempo infinito (luego de enviarle a estos objetos los eventos especificados en el archivo de configuración). Se evita así un deadlock en el objeto 3 para finalizar la simulación, ya que éste podrá procesar los últimos eventos de los objetos 1 y 2 con la seguridad de que realmente son los últimos.

5.3.1. Configuración de la simulación

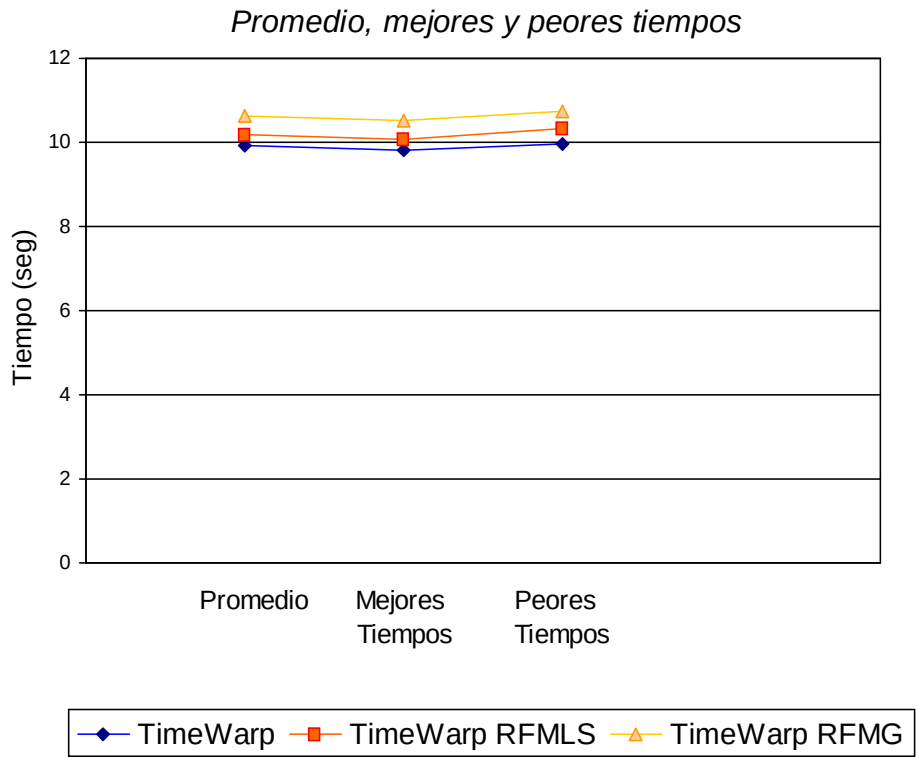
Configuración TimeWarp Modelo Frecuencia de Rollbacks Local y Modelo Frecuencia de Rollbacks Local por Pasos:
max: 10 (Umbral de rollbacks tras el cual el objeto se “duerme” o suspende)
rfm_interval: 0.05 (Intervalo de tiempo en el que el objeto va sumando los rollbacks que va sufriendo, y tras el cual se reinicia la cuenta).

Configuración TimeWarp Modelo Frecuencia de Rollbacks Global:
rfm_interval: 0.2

5.3.2. Resultados obtenidos

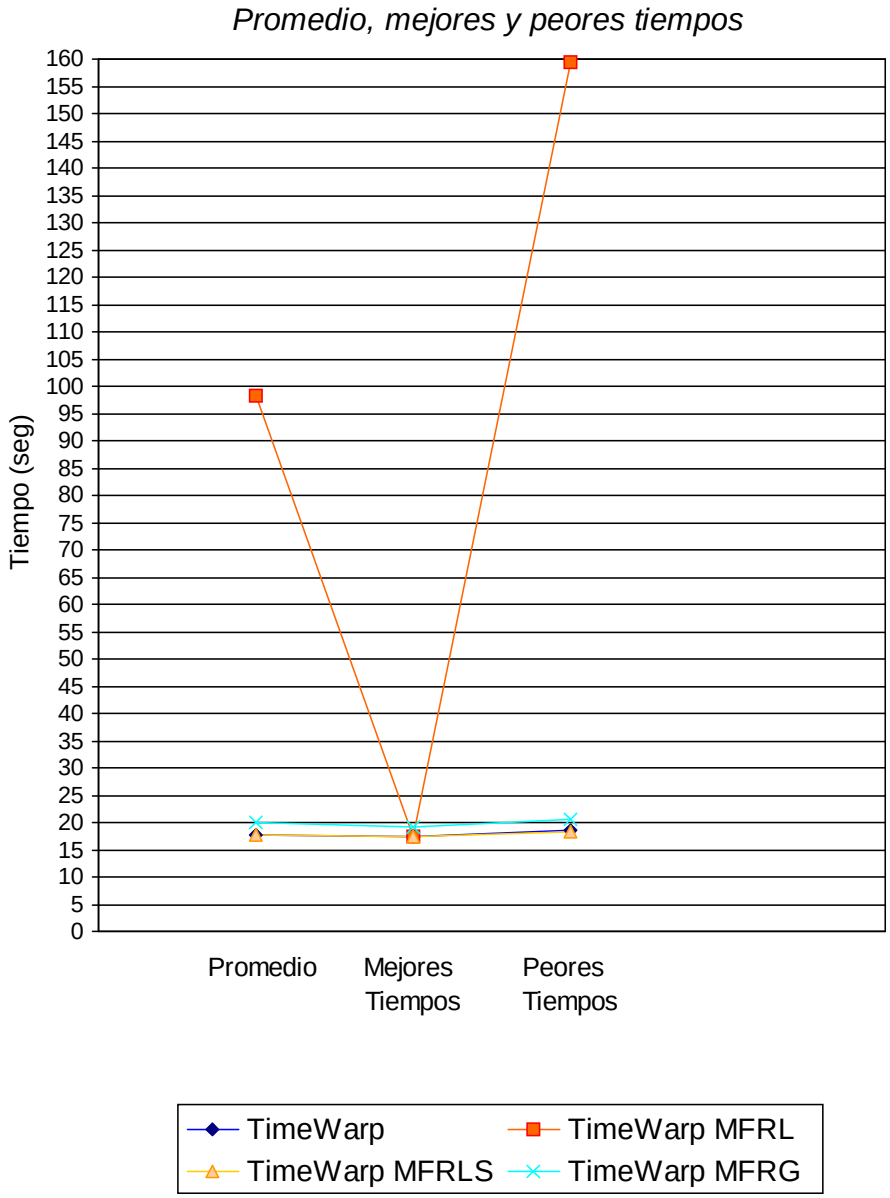
- Equipos Sun netra, red de 100 Mhz, 20000 eventos

4 Procesos Lógicos	Promedio	Mejor Tiempo	Peor Tiempo
TimeWarp	9.91	9.81	9.96
TimeWarp RFMLS	10.18	10.07	10.35
TimeWarp RFMG	10.62	10.53	10.73



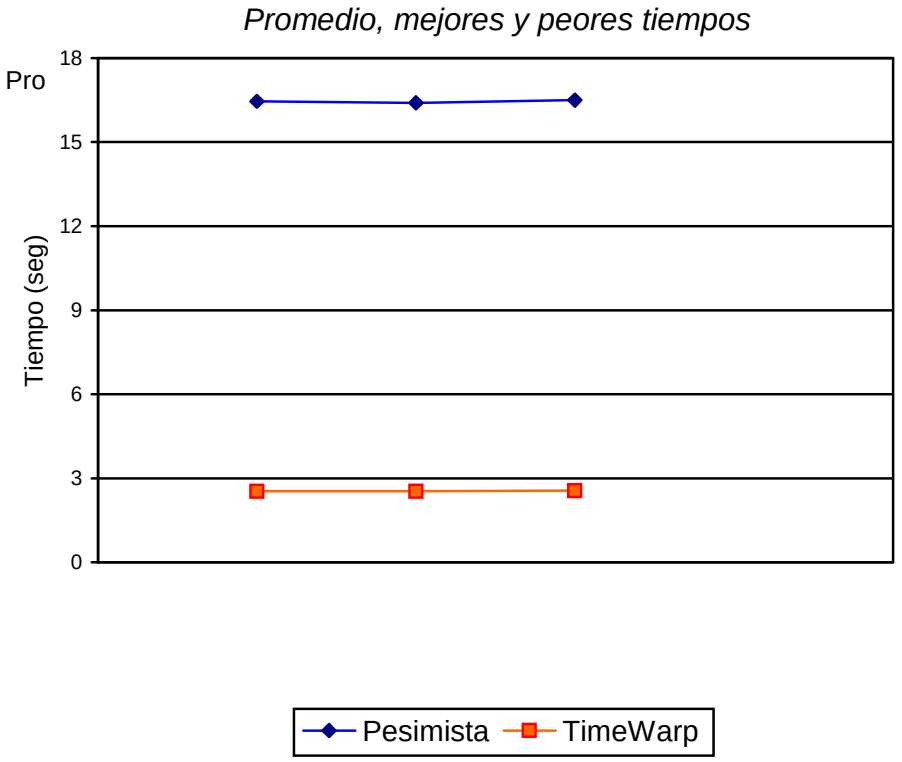
- Equipos Linux, red de 10 Mhz, 20000 eventos

4 Procesos Lógicos	Promedio	Mejor Tiempo	Peor Tiempo
TimeWarp	17.84	17.45	18.47
TimeWarp MFRL	98.35	17.47	159.41
TimeWarp MFRLS	17.77	17.36	18.24
TimeWarp MFRG	19.90	19.08	20.54



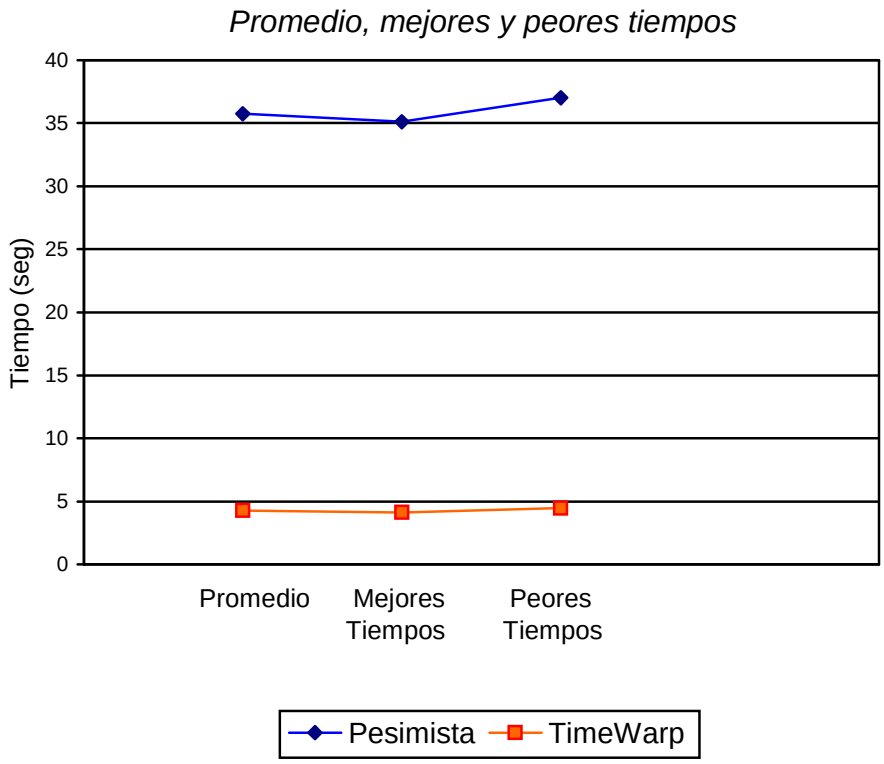
- Equipos Sun netra, red de 100 Mhz, 5000 eventos

4 Procesos Lógicos	Promedio	Mejor Tiempo	Peor Tiempo
Pesimista	16.45	16.40	16.50
TimeWarp	2.54	2.53	2.56



- Equipos Linux, red de 10 Mhz, 5000 eventos

4 Procesos Lógicos	Promedio	Mejor Tiempo	Peor Tiempo
Pesimista	35.75	35.11	37.02
TimeWarp	4.28	4.12	4.48



5.3.3. Observaciones y conclusiones

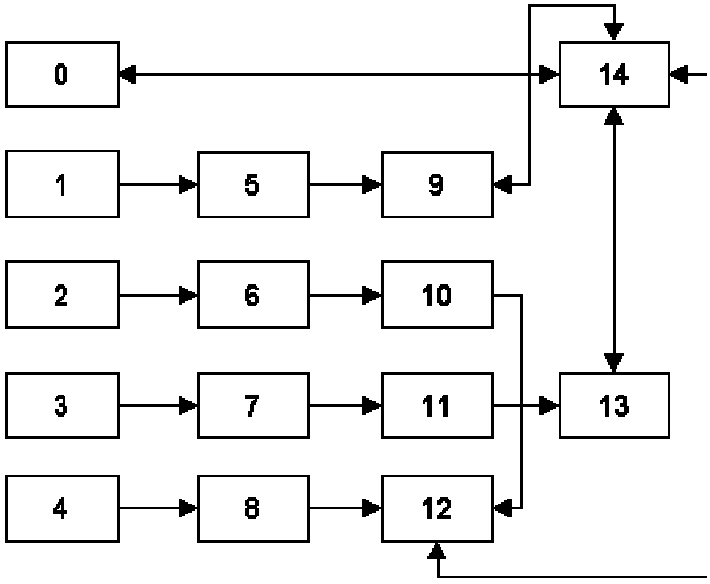
- El protocolo conservador tuvo una muy mala performance. Como luego se comprobará en otros ejemplos, la implementación del protocolo conservador en el kernel de simulación warped es correcta, aunque no es eficiente, ya que sus procesos lógicos avanzan muy lentamente en la simulación.
- En la red Linux, el protocolo TimeWarp Modelo Frecuencia de Rollback Local mostró una enorme inestabilidad. Sus ejecuciones demoraron entre 17.47 y 159.41 segundos. Se observó que tal diferencia se debió a las demoras producidas en el objeto 3. Si éste demoraba su ejecución debido a que había llegado al umbral de rollbacks, lo hacía hasta que finalice el actual ciclo de chequeo de rollbacks. Sin

embargo, debido a la topología de la simulación, la demora de un objeto produce la demora de toda la simulación, ya que el emisor (objeto 0) no puede seguir produciendo hasta tanto sus siguientes objetos no estén listos para la recepción de eventos.

- El protocolo Modelo Frecuencia de Rollbacks por Pasos corrige en forma eficiente los problemas explicados del protocolo Modelo Frecuencia de Rollbacks. Dado que el objeto 3 se suspende una cantidad finita de pasos (y no todo un intervalo de tiempo) se evita que su suspensión demore a toda la simulación, obteniéndose tiempos similares a los de TimeWarp.

5.4. Simulación Cartas2

Nuevamente se desea simular a una empresa que envía cartas con publicidad. Sin embargo, la estructura de distribución y chequeo interna de las cartas es un poco particular, como se muestra en la siguiente figura:



Se buscó que el último objeto de la simulación sufra muchos rollbacks con el protocolo de TimeWarp, para poder así analizar a los protocolos de Modelo Frecuencia de Rollbacks y NoTime.

Inician la simulación los objetos 0, 1, 2, 3 y 4. Envían tantos eventos como haya definido en el archivo de configuración a los objetos 14, 5, 6, 7 y 8 respectivamente, a partir del tiempo 10, y con intervalos de 1 unidad de tiempo. El objeto 14 reenvía un evento al objeto que le acaba de enviar uno, y los demás objetos reenvían el evento recibido a su siguiente objeto, tal como se observa en el esquema.

Para calcular el timestamp de un evento de salida de cualquier objeto se utiliza el siguiente algoritmo:

- Salvo el objeto 14, el timestamp es igual al local Virtual Time del objeto emisor + x *10, donde x es la resta del id del objeto destino y el id del objeto emisor.

- Para el objeto 14, el timestamp es igual al lVT del objeto + (15-d)*10, donde d es el id del objeto destino.

Se realizan estos cálculos ya que se busca que los eventos que deben pasar por más objetos para llegar al objeto 14 vayan llegando con timestamps menores al de los eventos que pasan por menos objetos, para provocar de ese modo muchos rollbacks en la simulación.

Por ejemplo, cuando los objetos 0, 1, 2, 3 y 4 envían eventos con timestamp 10, los objetos 9, 10, 11 y 12 recibirán eventos con tiempos 10 + (4*10) = 50. El objeto 14 recibirá de 9 un evento con tiempo 50 + (14-9)*10 = 100, el objeto 12 uno con tiempo 50 + (12-10)*10=70, y el objeto 13 uno con tiempo 50 + (13-11)*10 = 70. Finalmente, el objeto 14 recibe un evento del objeto 12 de timestamp 70 + (14-12)*10 = 90 y 13 con timestamps de 70 + (14-13)*10 = 80. Aquí se producirá un rollback en el objeto 14, ya que procesó en el ciclo anterior un evento de timestamp 100, pero ahora recibe uno con timestamp 80 y otro con timestamp 90.
Luego, al reenviar el objeto 14 los eventos recibidos a los objetos 0, 9, 12 y 13, lograremos provocarle más rollbacks en pasos sucesivos.

5.4.1. Configuración de la simulación

Se define en el archivo *cartas2.config*, ubicado en el mismo directorio de la simulación, el número de cartas que serán enviadas por el objeto 0 y la cantidad de procesos lógicos en la que se divide la simulación.
Los parámetros de la simulación utilizados son: 1000 cartas, 2, 3, 4 y 5 procesos lógicos.

Las simulaciones TimeWarp Modelo Frecuencia de Rollback Local por Pasos se configuraron del siguiente modo:

- TimeWarp RFMLS 1: max: 10, rfm_interval: 0.05
- TimeWarp RFMLS 2: max: 10, rfm_interval: 0.005
- TimeWarp RFMLS 3: max: 50, rfm_interval: 0.001
- TimeWarp RFMLS 4: max: 10, rfm_interval: 0.2

Las simulaciones TimeWarp Modelo Frecuencia de Rollback Global se configuraron del siguiente modo:

- TimeWarp RFMG rfm_interval: 0.2
- TimeWarp RFMG 2: rfm_interval: 0.02
- TimeWarp RFMG 3: rfm_interval: 0.002
- TimeWarp RFMG 4: rfm_interval: 1

5.4.2. Resultados obtenidos

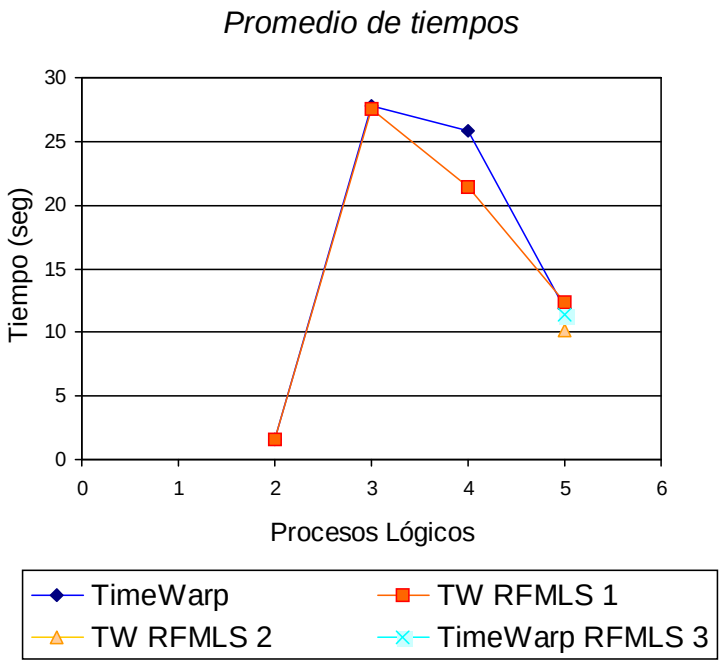
- Equipos Sun netra, red de 10 Mhz

2 Procesos Lógicos	Promedio	Mejor Tiempo	Peor Tiempo
TimeWarp	1.58	1.55	1.63
TimeWarp RFMLS 1	1.56	1.54	1.58

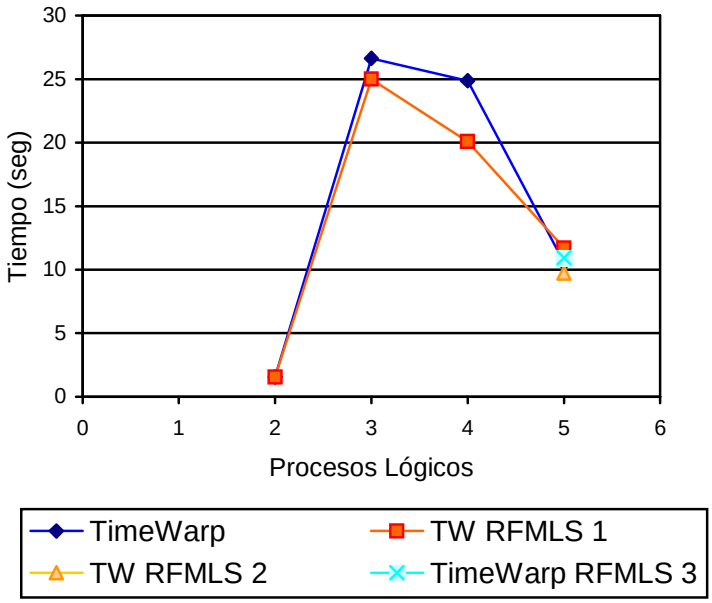
3 Procesos Lógicos	Promedio	Mejor Tiempo	Peor Tiempo
TimeWarp	27.85	26.64	28.97
TimeWarp RFMLS 1	27.61	24.99	29.43

4 Procesos Lógicos	Promedio	Mejor Tiempo	Peor Tiempo
TimeWarp	25.87	24.88	26.39
TimeWarp RFMLS 1	21.48	20.08	22.61

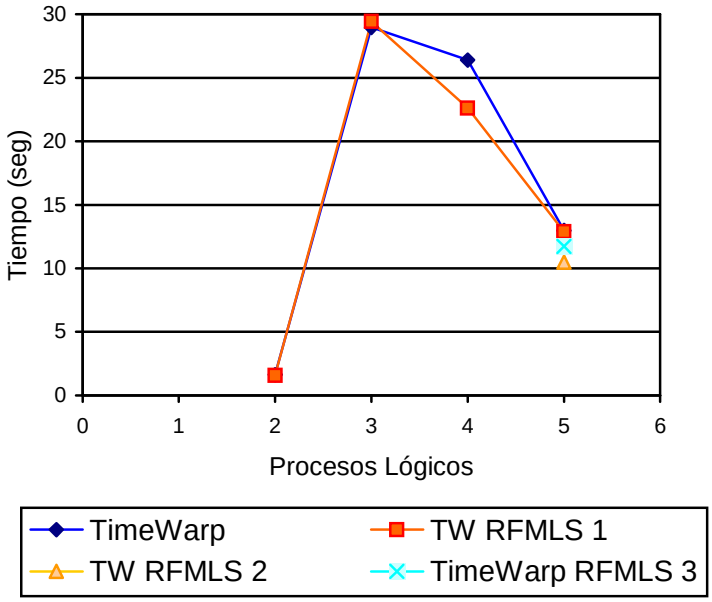
5 Procesos Lógicos	Promedio	Mejor Tiempo	Peor Tiempo
TimeWarp	11.86	10.56	12.97
TimeWarp RFMLS 1	12.40	11.70	12.89
TimeWarp RFMLS 2	10.15	9.73	10.51
TimeWarp RFMLS 3	11.39	10.94	11.72



Mejores tiempos

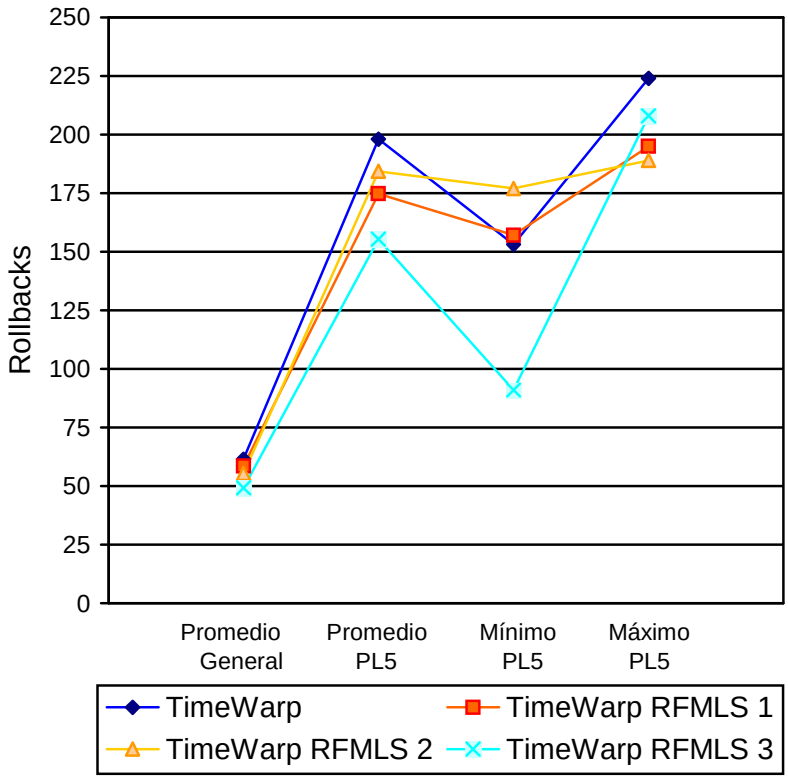


Peores tiempos



Estudio de rollbacks en todos los procesos lógicos y en el proceso lógico 5 en particular:

5 Procesos Lógicos **Rollbacks**	Prom. General	Prom. PL5	Mínimo PL5	Máximo PL5
TimeWarp	55.7	198.0	153.0	224.0
TimeWarp RFMLS 1	54.2	174.7	157.0	195.0
TimeWarp RFMLS 2	48.3	184.3	177.0	189.0
TimeWarp RFMLS 3	45.7	155.3	91.0	208.0



- Equipos Sun netra, red de 100 Mhz

1 Procesos Lógicos	Promedio	Mejor Tiempo	Peor Tiempo
Secuencial	0.15	0.14	0.17

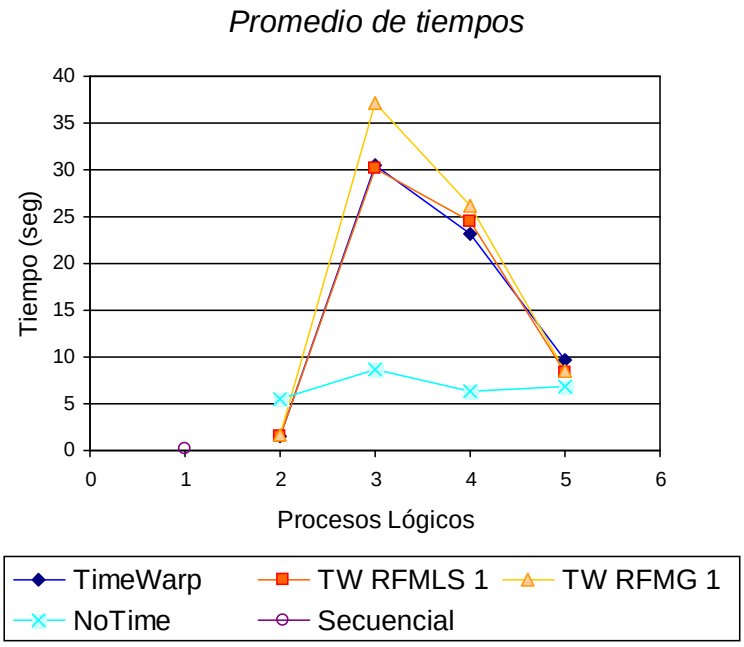
2 Procesos Lógicos	Promedio	Mejor Tiempo	Peor Tiempo
TimeWarp	1.45	1.44	1.46
TimeWarp RFMLS 1	1.50	1.48	1.53
TimeWarp RFMG 1	1.63	1.62	1.65
NoTime	5.52	5.48	5.54

3 Procesos Lógicos	Promedio	Mejor Tiempo	Peor Tiempo
TimeWarp	30.49	28.39	31.62
TimeWarp RFMLS 1	30.15	27.19	33.63
TimeWarp RFMG 1	37.18	35.94	38.73
NoTime	8.73	8.60	8.86

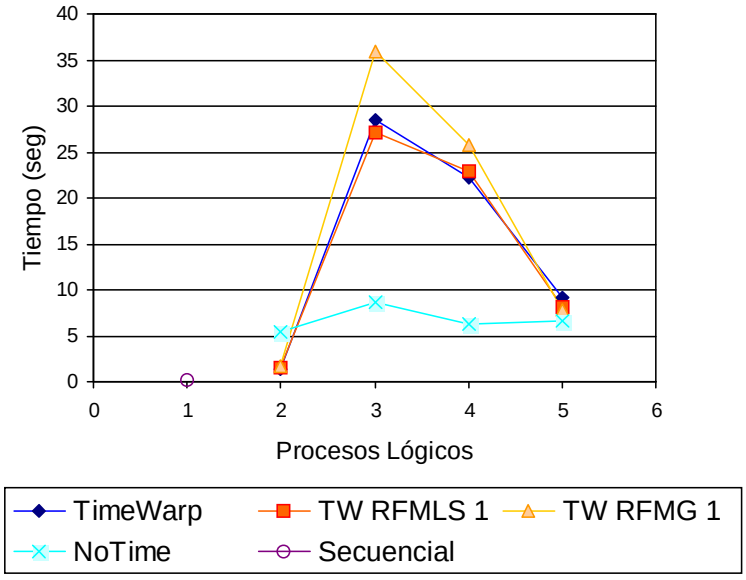
4 Procesos Lógicos	Promedio	Mejor Tiempo	Peor Tiempo
TimeWarp	23.12	22.25	24.05
TimeWarp RFMLS 1	24.57	22.85	26.80
TimeWarp RFMG 1	26.23	25.70	26.91
NoTime	6.32	6.30	6.36

5 Procesos Lógicos	Promedio	Mejor Tiempo	Peor Tiempo
TimeWarp	9.70	9.20	10.41
TimeWarp RFMLS 1	8.41	8.22	8.60
TimeWarp RFMLS 2	9.39	9.25	9.63
TimeWarp RFMLS 3	9.77	9.30	10.10

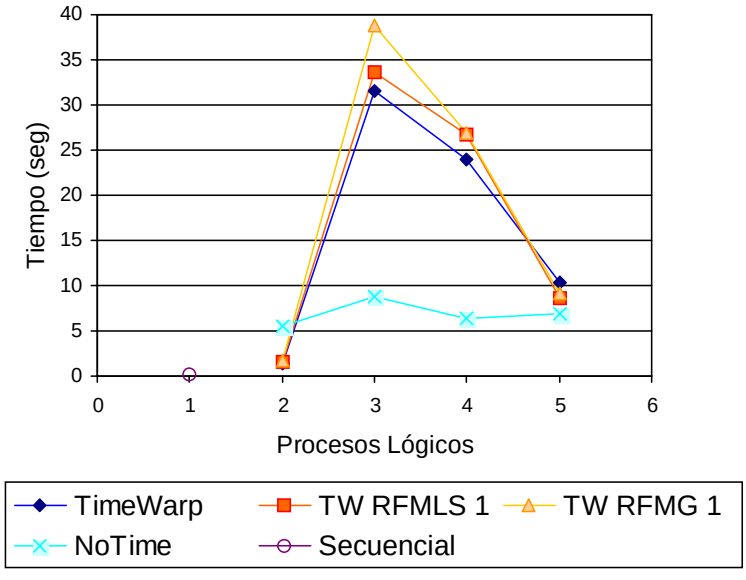
TimeWarp RFMLS 4	8.86	8.40	9.30
TimeWarp RFMG 1	8.46	7.85	9.16
TimeWarp RFMG 2	8.52	8.19	9.08
TimeWarp RFMG 3	9.85	9.65	10.22
TimeWarp RFMG 4	11.83	11.72	11.95
NoTime	6.76	6.63	6.86



Mejores tiempos

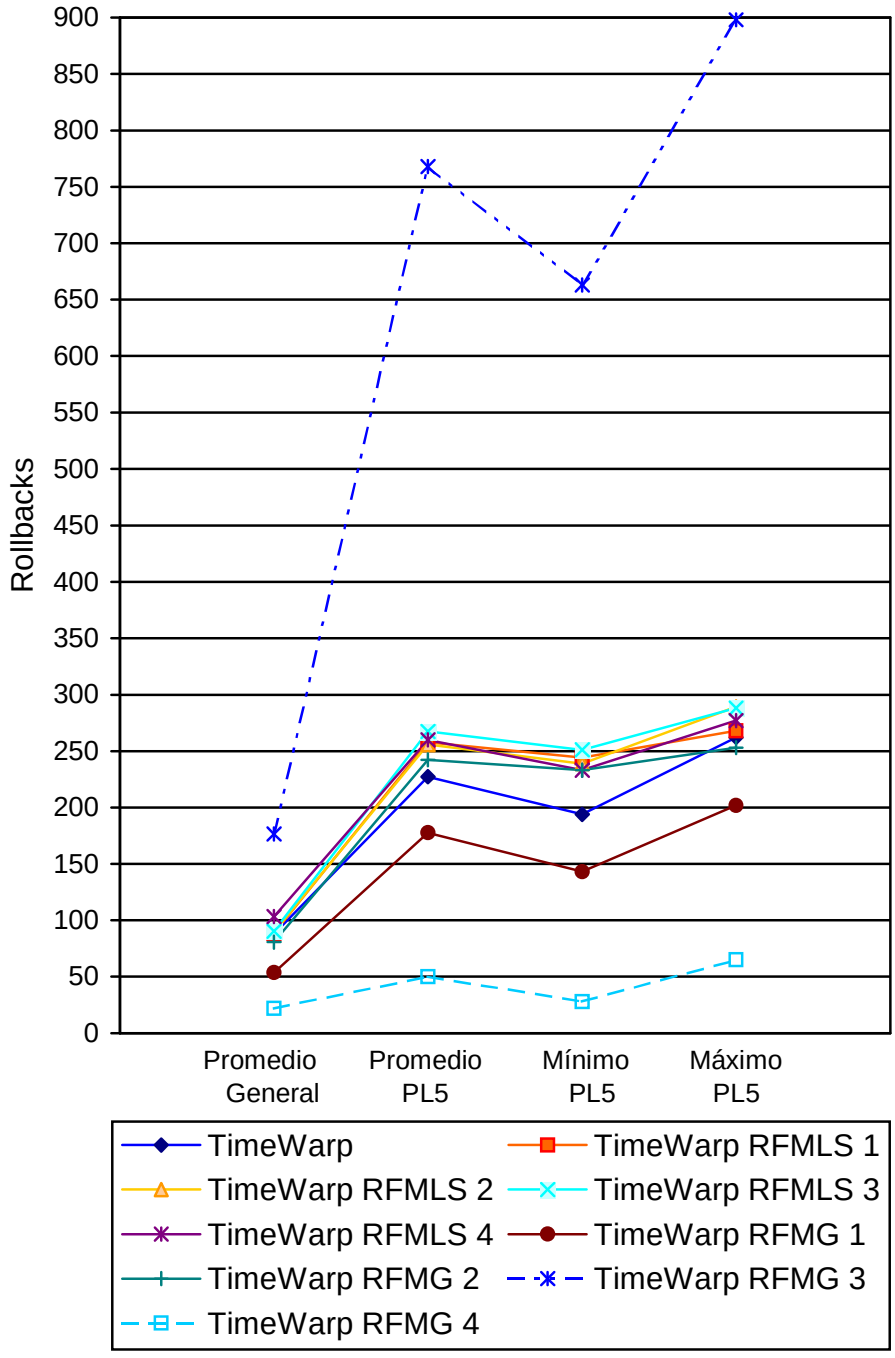


Peores tiempos



Estudio de rollbacks en todos los procesos lógicos y en el proceso lógico 5 en particular:

5 Procesos Lógicos **Rollbacks**	Prom. General	Prom. PL5	Mínimo PL5	Máximo PL5
TimeWarp	87.9	227.3	194.0	262.0
TimeWarp RFMLS 1	88.5	257.7	244.0	268.0
TimeWarp RFMLS 2	89.6	255.7	239.0	289.0
TimeWarp RFMLS 3	90.2	267.3	251.0	288.0
TimeWarp RFMLS 4	103.4	260.0	233.0	277.0
TimeWarp RFMG 1	53.5	177.7	143.0	202.0
TimeWarp RFMG 2	80.6	242.3	233.0	253.0
TimeWarp RFMG 3	176.5	768.0	663.0	898.0
TimeWarp RFMG 4	21.7	50.0	28.0	65.0



5.4.3. Observaciones y conclusiones (red Sun netra)

- En los protocolos de MFR, el uso de intervalos muy grandes no fueron útiles, ya que se corre el riesgo de que todos los objetos sufran suspensiones (tienen más tiempo para llegar a su umbral en el caso de MFR local, y más chances de ser el que más rollbacks acumuló en el caso global), demorando así en demasía la simulación. Tampoco fueron útiles los intervalos muy pequeños, dado que no permiten que los objetos lleguen a su umbral de rollbacks, por lo que éstos no se suspenden, y la simulación termina ejecutándose de igual manera a una simulación de TimeWarp pura. Se observa esto especialmente en el ejemplo MFR Global 3, donde el intervalo es de 0.002 segundos, y casi no se produce suspensión en ningún objeto.
- En las simulaciones MFR Local por pasos, fue útil probar diversas ejecuciones mediante la variación de los valores de max y rfm_interval. Si max es muy alto o rfm_interval demasiado pequeño, el objeto nunca llegará a su umbral de rollbacks y no se suspenderá. Si max es demasiado bajo o rfm_interval demasiado grande, el objeto se suspenderá muchas veces, con lo que se puede demorar en demasía la simulación, incluso más que si se ejecutara con el protocolo TimeWarp puro y se produjeran muchos rollbacks. Ocurre lo mismo para las simulación MFR global y el intervalo definido en mfr_interval. Es dificil decir a priori cuales son los valores óptimos para estas variables, ya que las mismas dependen de la simulación a ejecutar.
- El hecho de que una instancia de la simulación sufra menos rollbacks que otra no es garantía de que la misma ejecute en menos tiempo. Esto se debe a que si una simulación sufre muchas suspensiones, puede tender a disminuirse la cantidad de rollbacks de la misma, pero por otra parte se puede estar demorando su ejecución en demasía, por lo que el tiempo final de ejecución puede no ser el mejor.
- Al producirse muchos rollbacks en la simulación, las ejecuciones de Modelo Frecuencia de Rollback Local y Global arrojaron mejores tiempos que las de TimeWarp. Sin embargo, como quedó dicho en el párrafo anterior, esto dependió de los valores definidos en max y rfm_interval para RFM Local por Pasos, y

rfm_interval para RFM global. En el mejor de los casos se obtuvieron tiempos de ejecución aproximadamente 15% más rápidos que los de TimeWarp.

- El mejor tiempo obtenido con el protocolo MFR Global fue el del caso 1. Se observa que en este caso los objetos sufrieron menos rollbacks que en los demás, y que se suspendieron en una mayor cantidad de veces.
- Se confirman las diferencias entre las ejecuciones en una red de 10Mhz y una red de 100Mhz.
- El protocolo de NoTime obtuvo un excelente desempeño (excepto en el caso de 2 procesos lógicos, ya que como quedó dicho su arranque es más lento que el de TimeWarp). Aquí se nota se modo notable la performance de NoTime en simulaciones donde se producen muchos rollbacks, y hay muchos almacenamientos y recuperación de estados en cada uno de los objetos (dado que NoTime ignora estas circunstancias).

♦ Equipos Linux, red de 10 Mhz

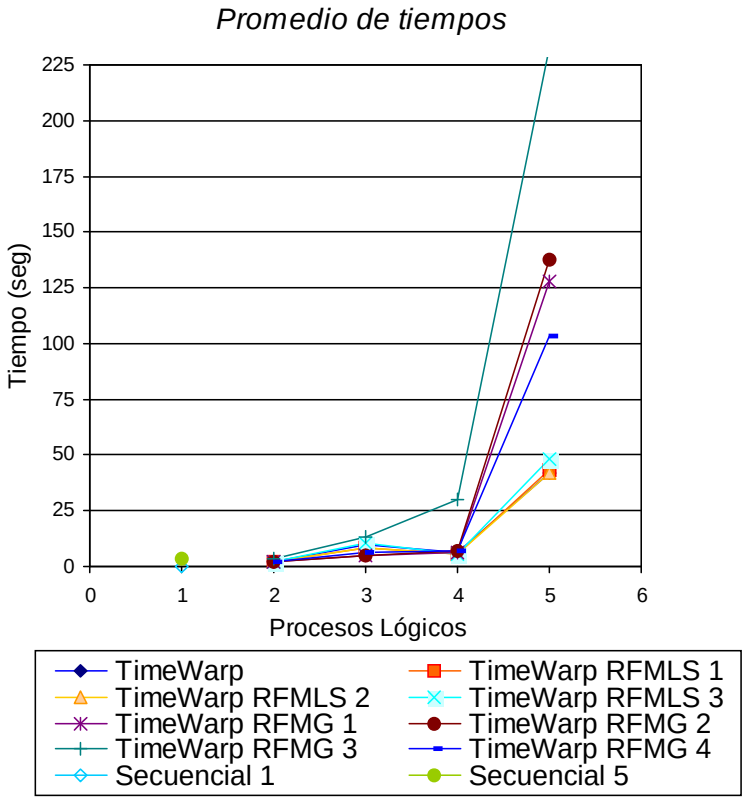
1 Procesos Lógicos	Promedio	Mejor Tiempo	Peor Tiempo
Secuencial 1	0.49	0.48	0.50
Secuencial 5	11.55	11.38	11.65

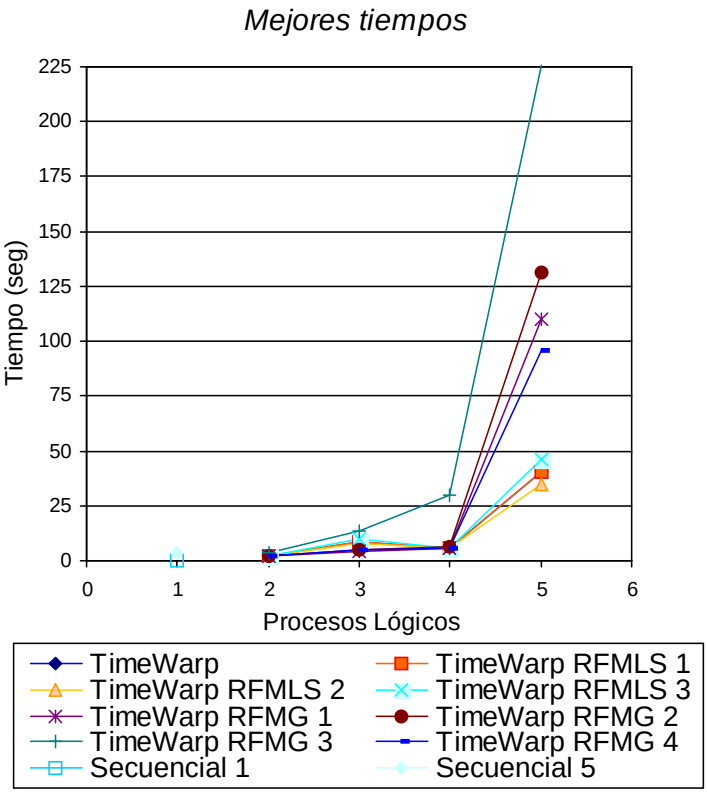
2 Procesos Lógicos	Promedio	Mejor Tiempo	Peor Tiempo
TimeWarp	2.33	2.23	2.41
TimeWarp RFMLS 1	2.39	2.37	2.41
TimeWarp RFMLS 2	2.33	2.27	2.37
TimeWarp RFMLS 3	2.36	2.34	2.38
TimeWarp RFMG 1	2.21	2.11	2.28
TimeWarp RFMG 2	2.19	2.17	2.22
TimeWarp RFMG 3	3.43	3.25	3.76
TimeWarp RFMG 4	2.06	2.03	2.12

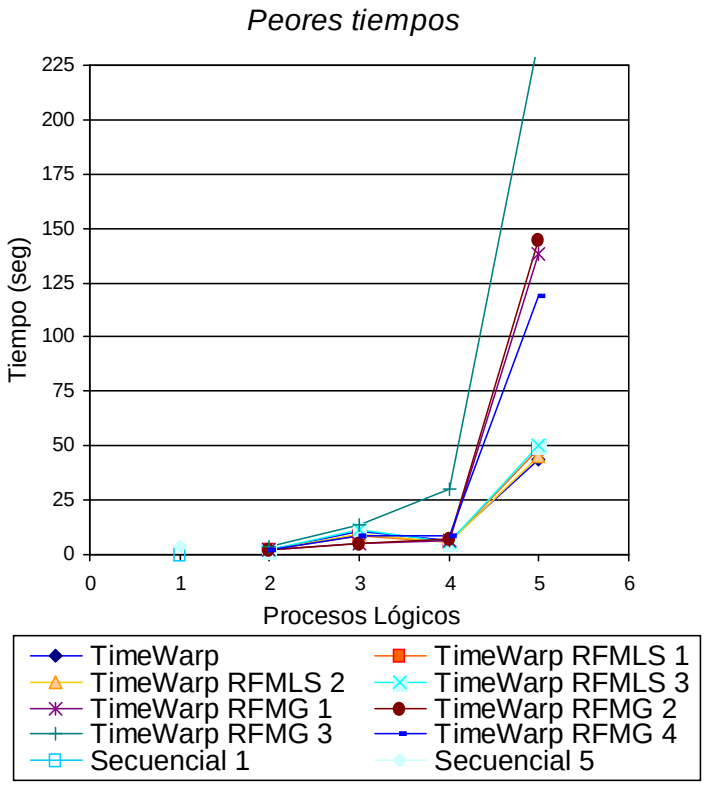
3 Procesos Lógicos	Promedio	Mejor Tiempo	Peor Tiempo
TimeWarp	9.48	8.27	10.49
TimeWarp RFMLS 1	8.36	8.18	8.68
TimeWarp RFMLS 2	8.59	7.47	9.39
TimeWarp RFMLS 3	10.57	9.77	11.20
TimeWarp RFMG 1	4.68	4.34	5.00
TimeWarp RFMG 2	4.75	4.74	4.77
TimeWarp RFMG 3	13.56	13.38	13.72
TimeWarp RFMG 4	6.32	5.22	8.29

4 Procesos Lógicos	Promedio	Mejor Tiempo	Peor Tiempo
TimeWarp	5.98	5.79	6.17
TimeWarp RFMLS 1	5.65	5.37	5.81
TimeWarp RFMLS 2	5.85	5.77	5.92
TimeWarp RFMLS 3	5.84	5.69	6.04
TimeWarp RFMG 1	6.09	5.50	6.42
TimeWarp RFMG 2	6.65	6.48	6.82
TimeWarp RFMG 3	29.84	29.59	30.28
TimeWarp RFMG 4	6.84	5.46	8.71

5 Procesos Lógicos	Promedio	Mejor Tiempo	Peor Tiempo
TimeWarp	42.14	40.38	43.40
TimeWarp RFMLS 1	43.58	40.73	48.65
TimeWarp RFMLS 2	41.73	35.13	45.33
TimeWarp RFMLS 3	48.42	46.44	50.49
TimeWarp RFMG 1	127.89	110.32	138.21
TimeWarp RFMG 2	137.76	131.49	144.40
TimeWarp RFMG 3	232.58	225.66	236.74
TimeWarp RFMG 4	103.62	95.62	118.97


Promedio de tiempos
225
200
175
150
125
100
75
50
25
0
Tiempo (seg)
0
1
2
3
4
5
6
Procesos Lógicos
TimeWarp
TimeWarp RFMLS 1
TimeWarp RFMLS 2
TimeWarp RFMLS 3
TimeWarp RFMG 1
TimeWarp RFMG 2
TimeWarp RFMG 3
TimeWarp RFMG 4
Secuencial 1
Secuencial 5


Mejores tiempos
225
200
175
150
125
100
75
50
25
0
Tiempo (seg)
0
1
2
3
4
5
6
Procesos Lógicos
TimeWarp
TimeWarp RFMLS 1
TimeWarp RFMLS 2
TimeWarp RFMLS 3
TimeWarp RFMG 1
TimeWarp RFMG 2
TimeWarp RFMG 3
TimeWarp RFMG 4
Secuencial 1
Secuencial 5



5.4.4. Observaciones y conclusiones (red Linux)

- Con cinco procesos lógicos, el equipo más lento no sufrió rollbacks (salvo en la última ejecución), dado que es el más lento de la red. Como contrapartida, demoró a la simulación en forma notable, ya que como se explicó en analogías entre TimeWarp y memoria virtual, demorar a un proceso lógico lo suficiente como para que no sufra rollbacks es perjudicial para la simulación, ya que se la termina demorando en su conjunto. Por esta razón la última ejecución con cinco procesadores bajo el protocolo de TimeWarp (ver Apéndice), fue la única en donde hubo rollbacks en el procesador 5, pero a la vez demoró aproximadamente un 7% menos que la primera ejecución, en donde éste procesador no presentó rollbacks.

- Con el uso de cinco procesadores los tiempos obtenidos en simulaciones con un mismo protocolo comenzaron a diferir hasta en un 10%. Parte de esto se debió a la inestabilidad del equipo más lento, el cual además tiene poca memoria, por lo que empeoraba su rendimiento al utilizar memoria secundaria y acceder mucho tiempo a disco, lo que le robaba tiempo en el uso del procesador.
- Hasta con 3 procesadores el protocolo RFMG1 obtuvo resultados apabullantes, llegando a demorar hasta un 45% menos que su equivalente en TimeWarp (se observaron pocos rollbacks pero muchísimas suspensiones en los procesos lógicos, ver Apéndice). Sin embargo, este protocolo agrega un overhead en cada procesador al enviar la cantidad de rollbacks a los demás procesadores y luego analizar su comportamiento, por lo que la inclusión de equipos lentos perjudicó al rendimiento de este protocolo mucho más que a los demás.
- En general se observa una mayor disparidad en los tiempos obtenidos en el entorno heterogéneo de equipos Linux que en el entorno homogéneo de máquinas Sun. Además se observó a lo largo de todo el desarrollo de la tesis muchas irregularidades en la implementación del software mpich en entornos Linux, cosa que no ocurrió con equipos Sun.

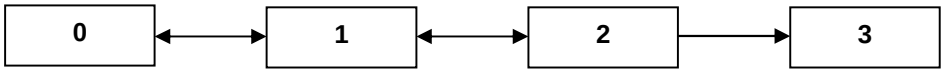
5.5. Simulación de Cartas3

Nuevamente se desea simular a otra empresa que envía cartas con publicidad a sus clientes. Para tal fin existe un generador de cartas (objeto 0), una entidad que corrige las cartas generadas (objeto 1), otra entidad posterior que se encarga del sellado (objeto 2), y finalmente el objeto que se encarga de la distribución final del correo (objeto 3). Las reglas de reenvío de cartas son las siguientes:

“excepto el objeto 3 que es sólo receptor, si el tiempo de recepción del evento es par y menor a 10000, se reenvía el evento al objeto id - 1 con el tiempo de recepción + 3; si no, se envía al objeto id + 1 con el tiempo de recepción + 5”. El objeto 0 envía a partir

del tiempo 15 tantas cartas como se especifica en el archivo de configuración de la simulación. Se ejecutará la simulación hasta que la misma alcance un tiempo de 10000.

Se muestra en la siguiente figura el esquema de la simulación:



Se asignó un objeto a cada proceso lógico de la simulación, por lo que la misma fue ejecutada en cuatro procesadores.

5.5.1. Configuración de la simulación

Objetos: 4
Cantidad inicial de eventos: 2
Cantidad de procesos lógicos: 4

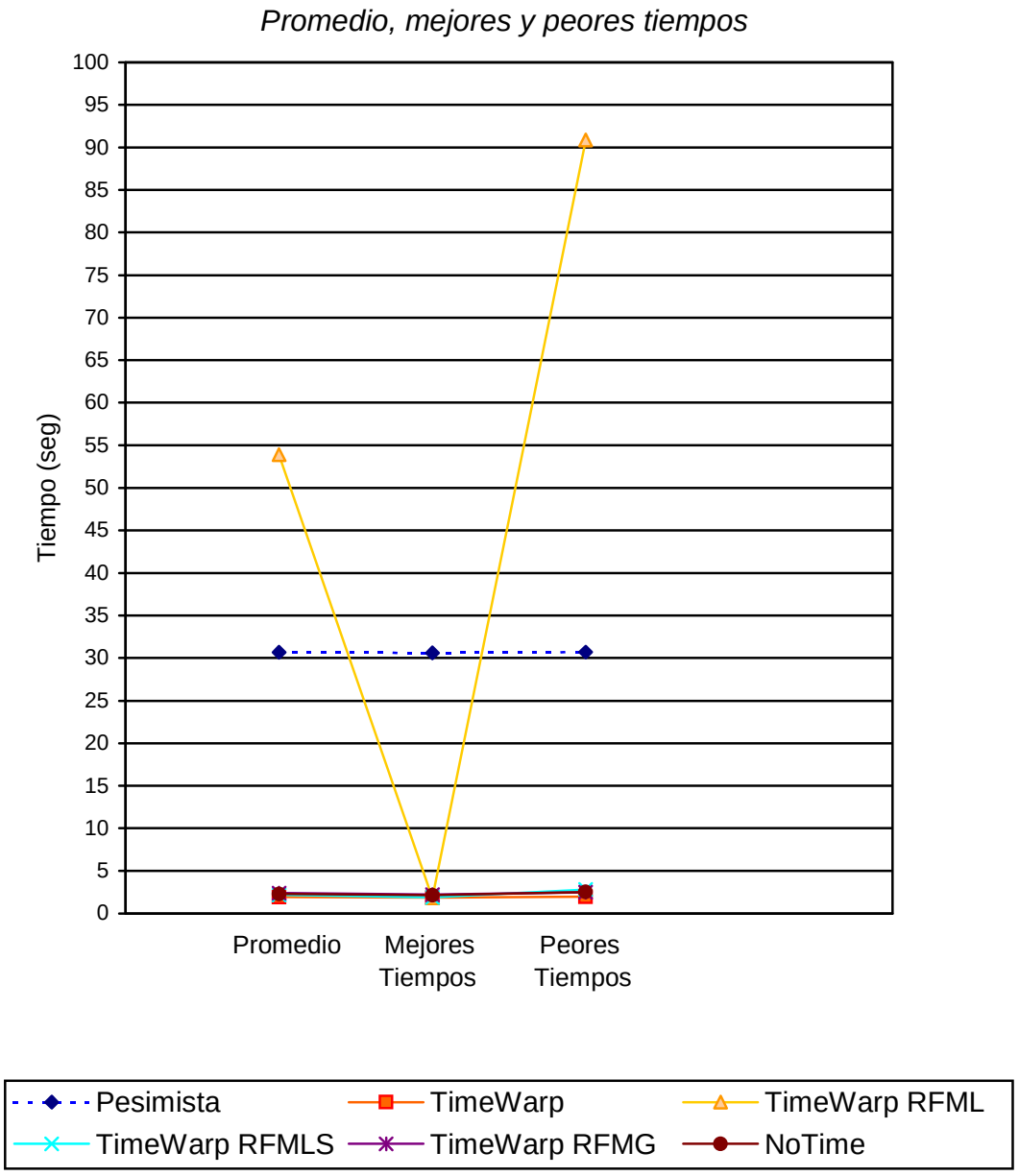
Configuración TimeWarp Modelo Frecuencia de Rollbacks Local y Modelo Frecuencia de Rollbacks Local por Pasos:
max: 10
rfm_interval: 0.05

Configuración TimeWarp Modelo Frecuencia de Rollbacks Global:
rfm_interval: 0.2

5.5.2. Resultados obtenidos

- Equipos Sun netra, red de 100 Mhz

4 Procesos Lógicos	Promedio	Mejor Tiempo	Peor Tiempo
Pesimista	30.67	30.62	30.70
TimeWarp	1.93	1.86	1.96
TimeWarp RFML	53.91	1.84	90.85
TimeWarp RFMLS	2.19	1.85	2.82
TimeWarp RFMG	2.40	2.23	2.50
NoTime	2.30	2.14	2.54

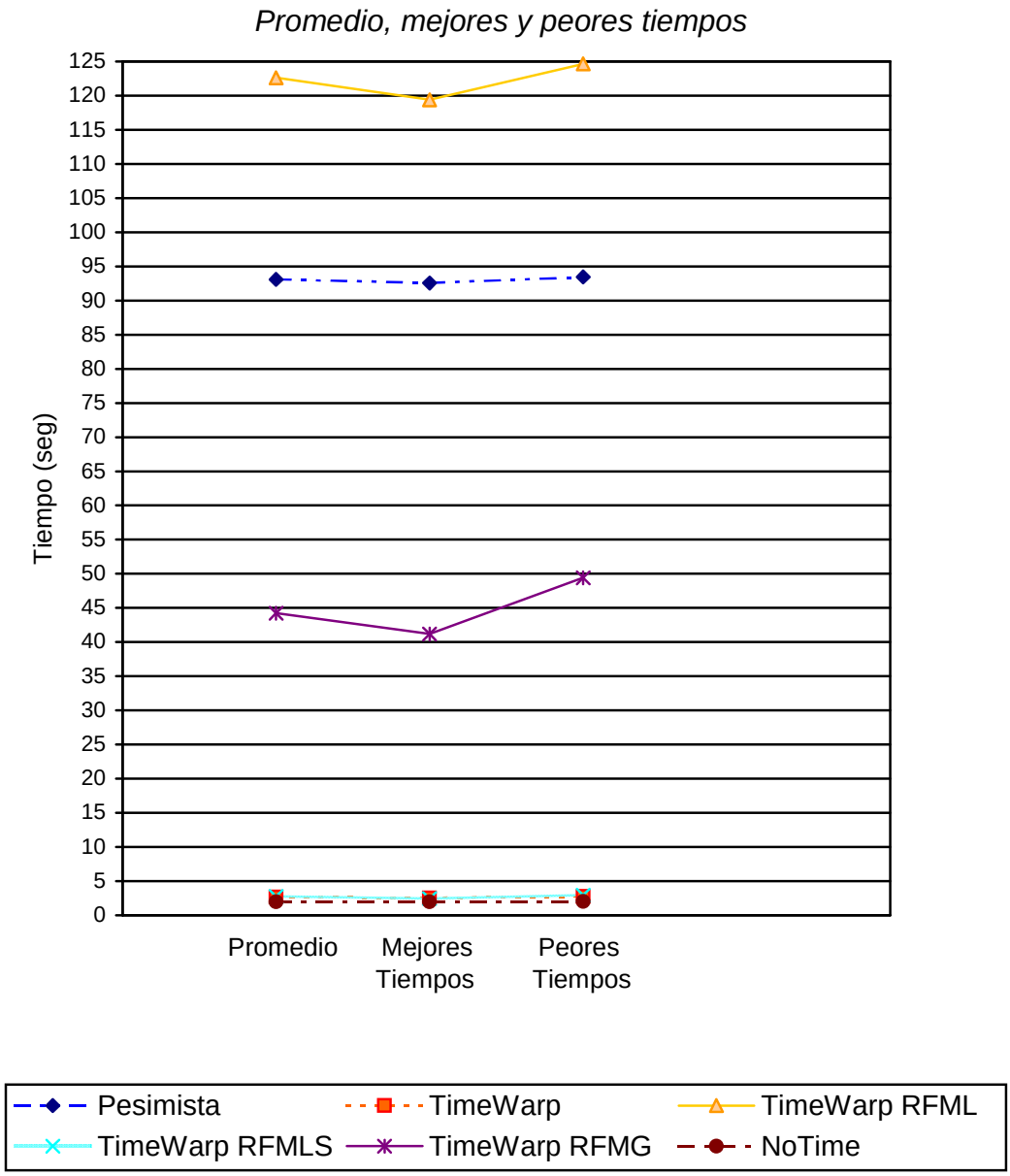

Promedio, mejores y peores tiempos
Tiempo (seg)
100
95
90
85
80
75
70
65
60
55
50
45
40
35
30
25
20
15
10
5
0
Promedio
Mejores Tiempos
Peores Tiempos
Pesimista
TimeWarp
TimeWarp RFML
TimeWarp RFMLS
TimeWarp RFMG
NoTime

5.5.3. Observaciones y conclusiones (red Sun netra)

- El protocolo conservador tuvo una muy mala performance, ya que mientras el protocolo TimeWarp estándar tomó en promedio 1.93 segundos para ejecutar la simulación, el pesimista tomó en promedio 30.67.
- El protocolo TimeWarp Modelo Frecuencia de Rollback Local mostró una enorme inestabilidad. Sus ejecuciones demoraron entre 1.84 y 90.85 segundos. Se observó que tal diferencia se debió a las demoras producidas en el objeto 2. Si éste demoraba su ejecución debido a que había llegado al umbral de rollbacks, lo hacía hasta que finalice el actual ciclo de chequeo de rollbacks. Sin embargo, debido a la topología de la simulación, la demora de un objeto produce la demora de toda la simulación, ya que el emisor (objeto 0) no puede seguir produciendo hasta tanto sus siguientes objetos no estén listos para la recepción de eventos.
- La estructura secuencial de la simulación explica también por qué el protocolo de TimeWarp fue el más rápido en ejecutarse. Esta simulación no se beneficia con la demora o suspensión de un objeto (a pesar de que el mismo haya incurrido en muchos rollbacks), debido a que esto produce la demora total de la simulación, al ser dependientes todos los objetos entre sí para poder avanzar.
- Las simulaciones con el protocolo de NoTime mostraron tener un inicio más lento que aquellas con TimeWarp. Luego, como esta simulación tuvo pocos rollbacks y al ser relativamente pequeña (cada PL tuvo que salvar el estado de sólo un objeto, y la cantidad de eventos a simular fue pequeña), esta técnica no pudo obtener tiempos menores que los de TimeWarp.
- El protocolo Modelo Frecuencia de Rollbacks por Pasos corrige en forma eficiente los problemas explicados del protocolo Modelo Frecuencia de Rollbacks. Dado que el objeto 2 se suspende una cantidad finita de pasos (y no todo un intervalo de tiempo) se evita que su suspensión demore a toda la simulación, obteniéndose tiempos levemente superiores a los de TimeWarp.

- Equipos Linux, red de 10 Mhz

4 Procesos Lógicos	Promedio	Mejor Tiempo	Peor Tiempo
Pesimista	93.08	92.57	93.42
TimeWarp	2.65	2.54	2.73
TimeWarp RFML	122.62	119.38	124.63
TimeWarp RFMLS	2.76	2.45	2.92
TimeWarp RFMG	44.21	41.15	49.40
NoTime	1.95	1.94	1.97



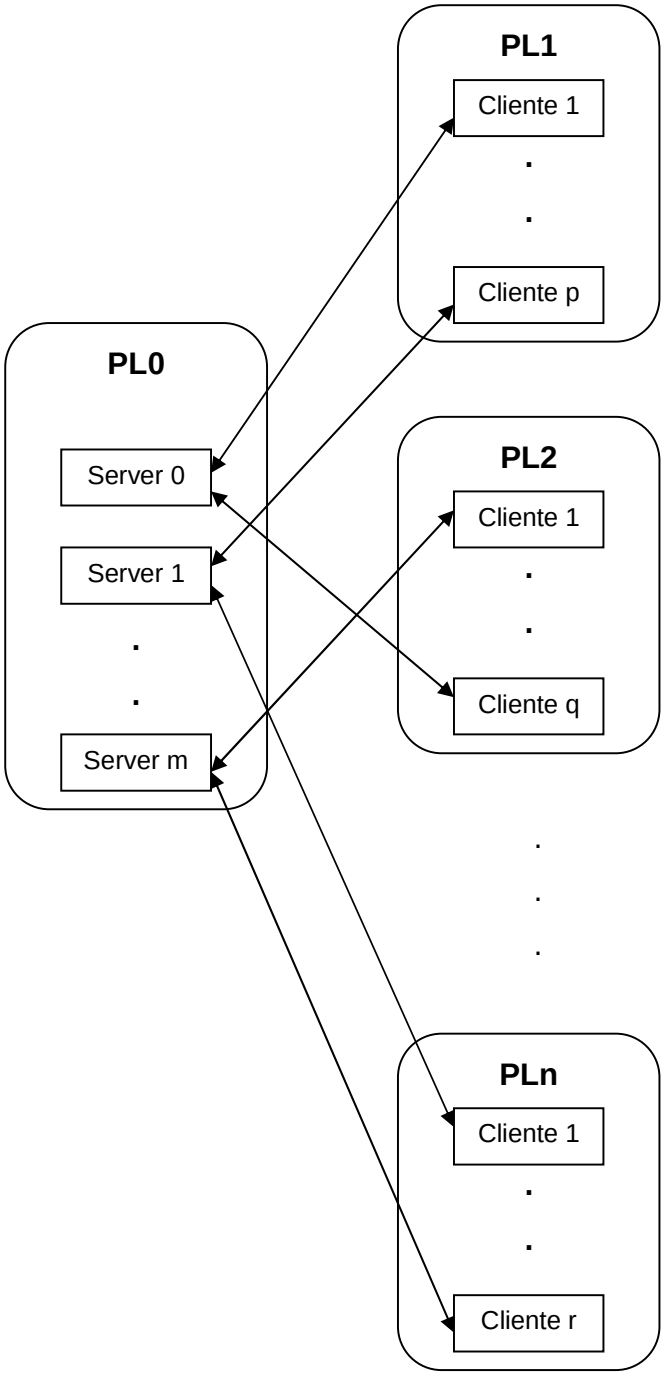
5.5.4. Observaciones y conclusiones (red Linux)

- Al igual que en las simulaciones en entorno Sun, el protocolo conservador tuvo una muy mala performance, como así también el protocolo TimeWarp Modelo Frecuencia de Rollback Local.
- En MRF Local, el PL2 realizó muchas suspensiones, por lo que hubo pocos rollbacks, pero así hizo demorar a las simulaciones en forma desmedida. Con el protocolo MFR Local por pasos ocurren menos suspensiones y muchos más rollbacks, por lo que así no demoró tanto a la simulación y finalizaba en un menor tiempo.

5.6. Simulación de HTTP

Se simula un esquema cliente-servidor, en donde los clientes envían un requerimiento a un servidor, y se quedan a la espera de que éste le responda, para poder enviarle un pedido posterior. Una aplicación típica que responde a este esquema es el de un usuario que utiliza un navegador (cliente HTTP) y requiere páginas, gráficos, etc., de servidores de web.

En nuestro ejemplo cada cliente envía y recibe pedidos de un mismo servidor. Se decidió ubicar a todos los servidores en un mismo proceso lógico, y todos los demás procesos lógicos contienen objetos clientes. Se muestra lo dicho en el siguiente esquema:


PL1
Cliente 1
Cliente p
PL0
Server 0
Server 1
Server m
PL2
Cliente 1
Cliente q
PLn
Cliente 1
Cliente r

5.6.1. Configuración de la simulación

Se define en el archivo *http.config*, ubicado en el mismo directorio de la simulación, la cantidad de objetos servidores, cantidad de objetos clientes, cantidad de requerimientos que hará cada cliente y la cantidad de procesos lógicos en la que se divide la simulación.

Parámetros de la simulación:
Objetos servidores: 5
Objetos clientes: 15
Cantidad inicial de eventos: 10, 100 y 1000
Cantidad de procesos lógicos: 5

5.6.2. Resultados obtenidos

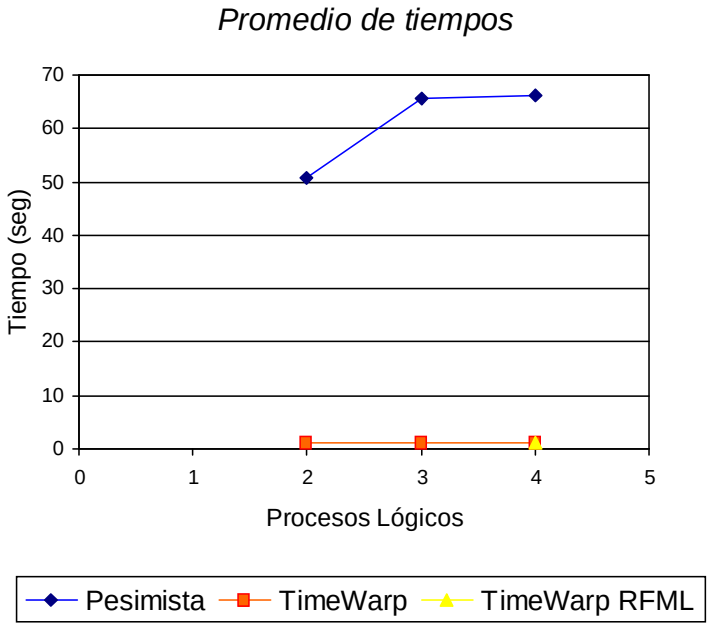
- Equipos Sun netra, red de 100 Mhz, 5 servers - 15 clientes - 100 requerimientos

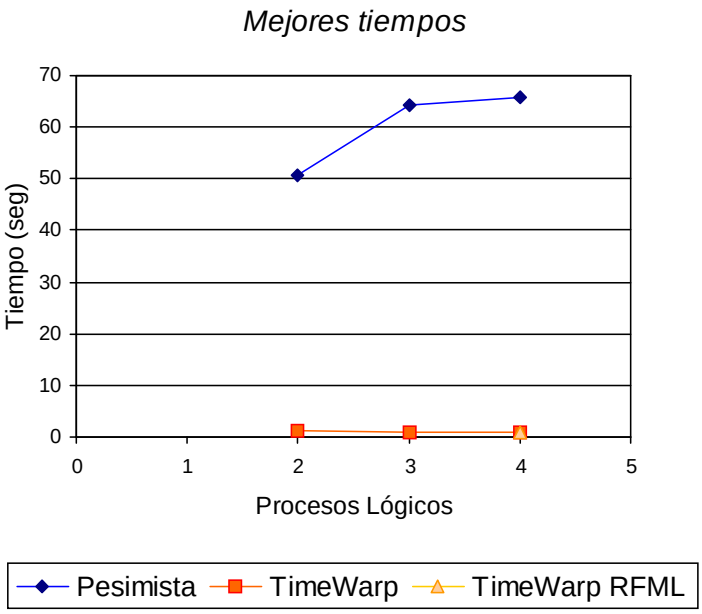
2 Procesos Lógicos	Promedio	Mejor Tiempo	Peor Tiempo
Pesimista	50.84	50.54	51.37
TimeWarp	1.25	1.19	1.28

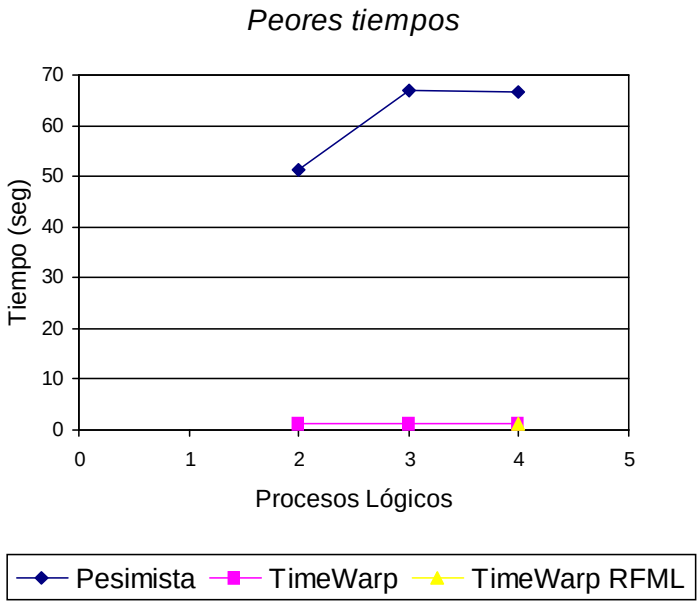
3 Procesos Lógicos	Promedio	Mejor Tiempo	Peor Tiempo
Pesimista	65.58	64.25	66.95
TimeWarp	1.13	1.02	1.27

4 Procesos Lógicos	Promedio	Mejor Tiempo	Peor Tiempo

Pesimista	66.15	65.80	66.70
TimeWarp	1.08	1.03	1.18
TimeWarp RFML	1.10	1.04	1.19







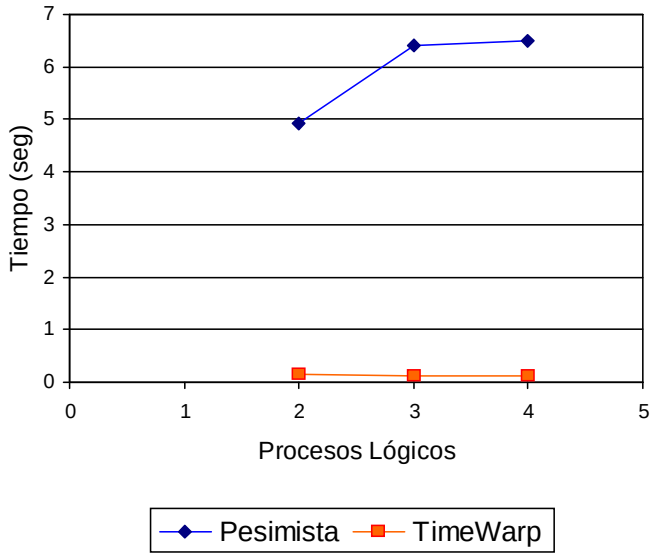
- Equipos Sun netra, red de 100 Mhz, 5 servers - 15 clientes - 10 requerimientos

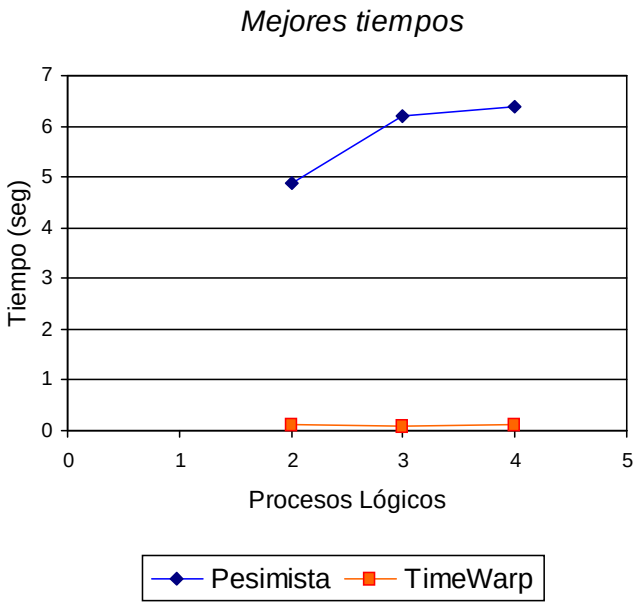
2 Procesos Lógicos	Promedio	Mejor Tiempo	Peor Tiempo
Pesimista	4.91	4.89	4.95
TimeWarp	0.14	0.13	0.15

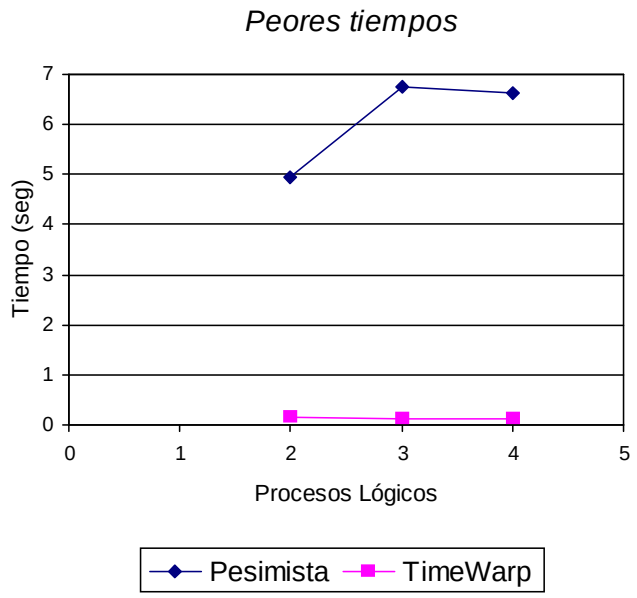
3 Procesos Lógicos	Promedio	Mejor Tiempo	Peor Tiempo
Pesimista	6.41	6.21	6.74
TimeWarp	0.11	0.10	0.11

4 Procesos Lógicos	Promedio	Mejor Tiempo	Peor Tiempo
Pesimista	6.50	6.38	6.64
TimeWarp	0.11	0.11	0.11

Promedio de tiempos







- Equipos Sun netra, red de 100 Mhz, 5 servers - 15 clientes - 10 requerimientos

4 Procesos Lógicos	Promedio	Mejor Tiempo	Peor Tiempo
Pesimista	658.51	654.72	665.97
TimeWarp	10.65	10.44	10.78

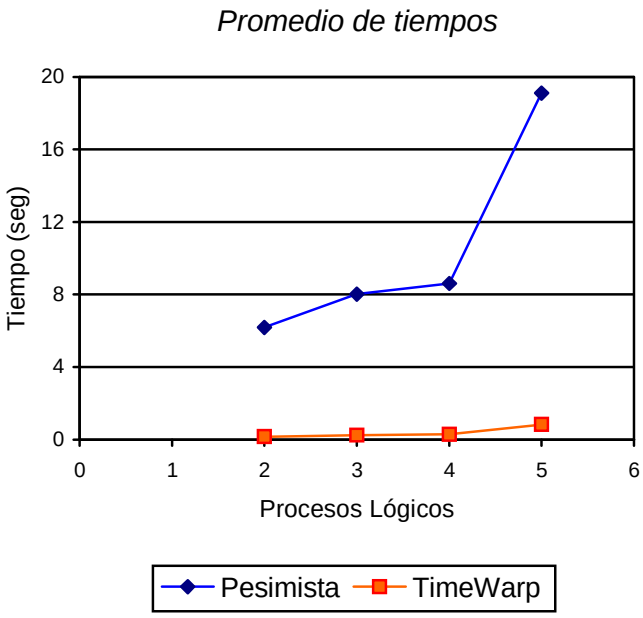
- Equipos Linux, red de 10 Mhz; 5 servers, 15 clientes y 100 requerimientos

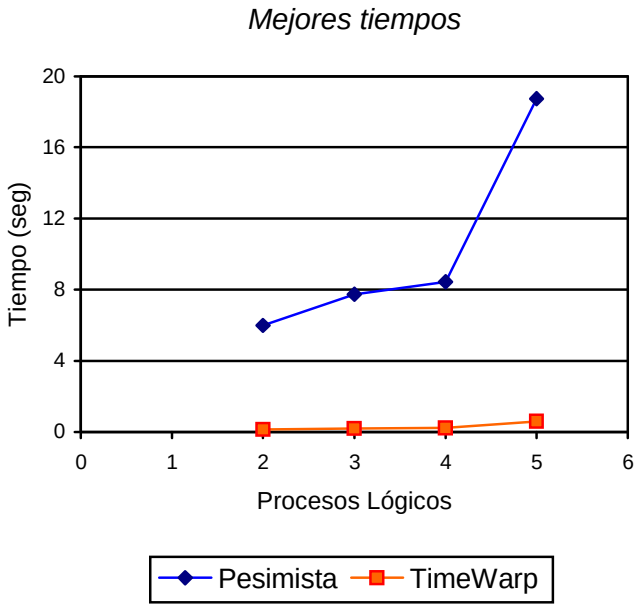
2 Procesos Lógicos	Promedio	Mejor Tiempo	Peor Tiempo
Pesimista	6.18	5.99	6.39
TimeWarp	0.15	0.14	0.17

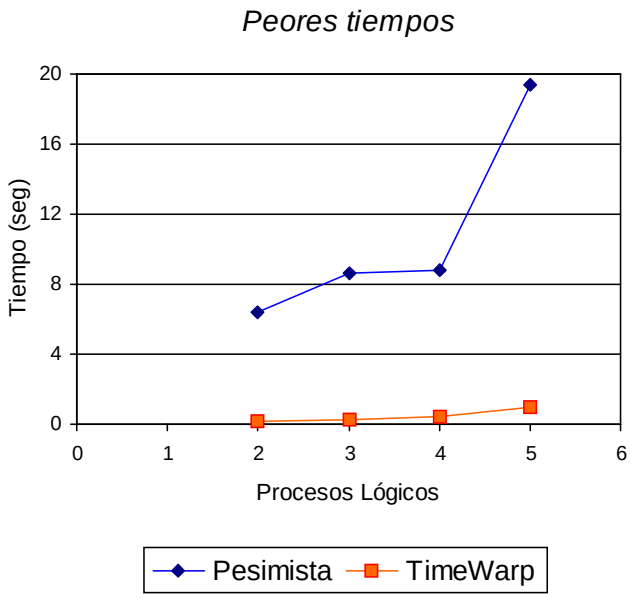
3 Procesos Lógicos	Promedio	Mejor Tiempo	Peor Tiempo
TimeWarp	8.02	7.74	8.58
TimeWarp RFML	0.24	0.19	0.28

4 Procesos Lógicos	Promedio	Mejor Tiempo	Peor Tiempo
Pesimista	8.60	8.45	8.81
TimeWarp	0.30	0.23	0.42

5 Procesos Lógicos	Promedio	Mejor Tiempo	Peor Tiempo
Pesimista	19.10	18.74	19.42
TimeWarp	0.82	0.59	1.01


Promedio de tiempos
20
16
12
8
4
0
Tiempo (seg)
0
1
2
3
4
5
6
Procesos Lógicos
Pesimista
TimeWarp


Mejores tiempos
20
16
12
8
4
0
Tiempo (seg)
0
1
2
3
4
5
6
Procesos Lógicos
Pesimista
TimeWarp



- Equipos Linux, red de 10 Mhz; 5 servers, 15 clientes y 100 requerimientos

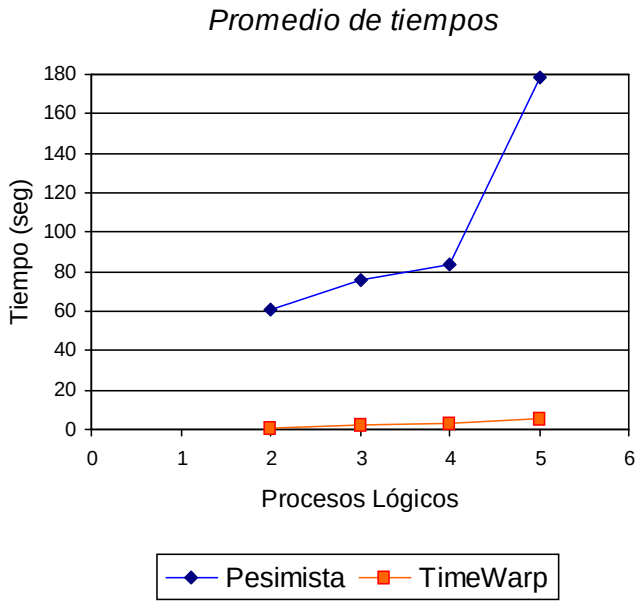
2 Procesos Lógicos	Promedio	Mejor Tiempo	Peor Tiempo
Pesimista	60.89	60.67	61.06
TimeWarp	1.13	1.12	1.14

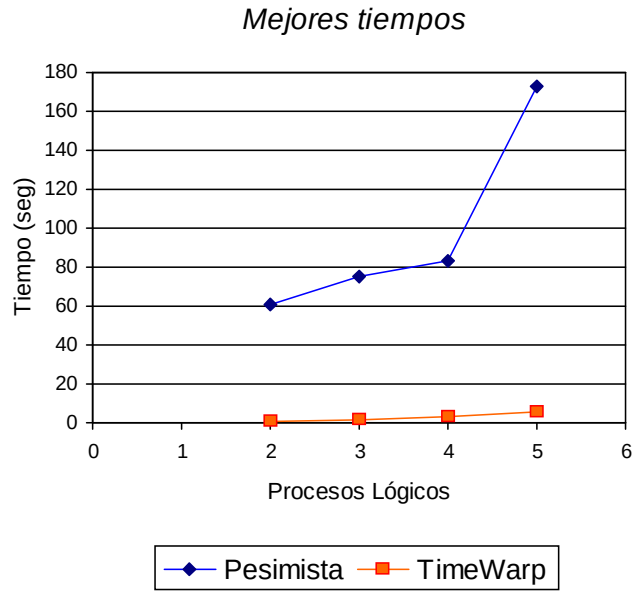
3 Procesos Lógicos	Promedio	Mejor Tiempo	Peor Tiempo
TimeWarp	75.55	75.02	75.87
TimeWarp RFML	2.16	1.95	2.35

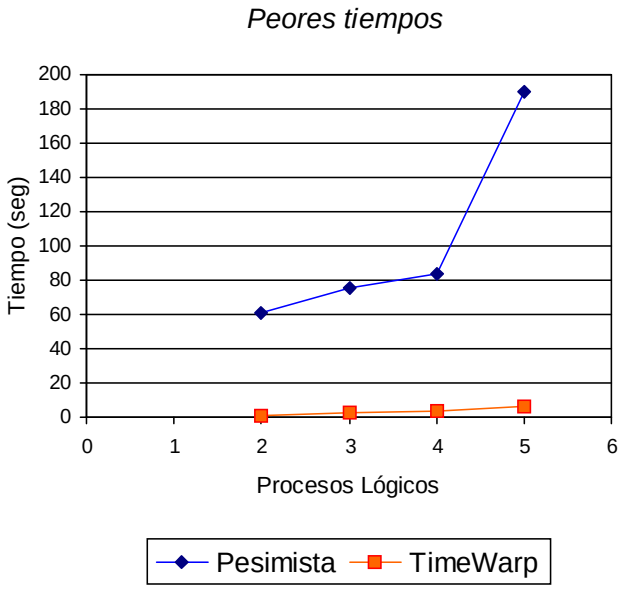
4 Procesos Lógicos	Promedio	Mejor Tiempo	Peor Tiempo
Pesimista	83.39	83.01	83.84
TimeWarp	3.22	3.00	3.59

5 Procesos Lógicos	Promedio	Mejor Tiempo	Peor Tiempo

Pesimista	178.79	172.68	189.95
TimeWarp	5.70	5.32	5.94







5.6.3. Observaciones y conclusiones

- El protocolo conservador tuvo una paupérrima performance respecto a las simulaciones de TimeWarp. Se observó sin embargo que el tiempo que tardaba la simulación fue proporcional a la cantidad de eventos enviados: si se envían 10 veces más eventos, la simulación tardaba 10 veces más, por lo que el crecimiento en los tiempos de ejecución se comportó totalmente lineal. Lo anterior se cumplió tanto para las simulaciones de TimeWarp como para las simulaciones conservadoras.
- Esta observación puede ser útil en el caso de grandes simulaciones, en donde para ver si la misma es correcta se puede probar de ejecutar con pocos eventos, y luego deducir cuánto demorará la misma con la cantidad real de eventos a simular.

Luego de obtener semejante diferencia de tiempos entre los protocolos conservador y de TimeWarp, se modificó la simulación, ejecutándose dos instancias distintas, que

obedecen al siguiente criterio de ejecución. Supongamos que los clientes i, j le envían requerimientos al servidor k (que tarda 2 unidades de tiempo en procesar cada pedido), y que cada vez que reciben una respuesta, envían el siguiente requerimiento con el tiempo de la respuesta (que pasa a ser el Local Virtual Time del objeto) más el id del objeto (o sea, el lookahead del objeto *i* es de *i* unidades de tiempo).

Ejemplo:
i=10
j=20

1) Si los objetos i y j reciben una respuesta de k en tiempo 100, el objeto i enviará el siguiente requerimiento en tiempo 100+10=120, y el objeto j en tiempo 100+20=120. Luego, k procesará de modo seguro el requerimiento de i.

2) Si en cambio los objetos i,j enviaran el próximo requerimiento con el timestamp lVT + id * 3, i enviaría su siguiente pedido en tiempo 130, y el objeto j en tiempo 160. Luego, el server k atenderá el pedido de i, y lo responderá con tiempo 132. Si el objeto i no tuviera mas mensajes para enviar, la simulación avanzaría en base a la circulación de mensajes nulos (ya que si no el server k nunca procesaría el pedido del objeto j). Así, como el lookahead de i es 10, enviará mensajes nulos con timestamp de 142, 152, 162, y aquí el server k podrá procesar el requerimiento con timestamp 160 del objeto j.
Si la diferencia entre los tiempos de reenvío de pedidos entre los objetos i y j se acrecentaran, se precisara que el objeto i envíe cada vez mas objetos nulos para que se procese el pedido del objeto j.
Estas pruebas demostraron que en base al lookahead de los objetos clientes, las simulaciones pesimistas podían demorarse en su ejecución tanto como se deseara. En cambio, como el protocolo de TimeWarp simula en forma agresiva sin importarle el timestamp de los eventos, no sufre este tipo de problemas.

6. Conclusiones y trabajo futuro

6.1. Conclusiones generales

Es mucho más difícil de ejecutar una simulación pesimista que optimista (TimeWarp).
En primer lugar, debido a la naturaleza conservadora del protocolo, todos los objetos deben saber qué objetos de la simulación pueden enviarle eventos, como así también deben saber a qué otros objetos pueden enviar eventos. Para indicarle estas restricciones al kernel de la simulación es preciso escribir un archivo de configuración (lo llamamos *obj_queues*) con todo este detalle, y además el lookahead de cada objeto, para el envío de mensajes nulos. En una simulación con muchos objetos, esta tarea es muy ardua, y si el archivo no detalla con exactitud al esquema de la simulación, no se podrá ejecutar la simulación del sistema en forma exitosa.
Si se deseara ejecutar la misma simulación pero con otra cantidad de objetos, en TimeWarp bastaría con modificar este número en el archivo de configuración de la simulación; con el protocolo pesimista además se debe modificar el archivo obj_queues, con los riesgos de cometer errores que esto
genera.
Por otra parte, puede ser necesario modificar el código de una simulación para que se la pueda compilar con el protocolo pesimista. La simulación de Cartas es un ejemplo de ello.

La implementación en particular del protocolo conservador en Warped no es eficiente. Esto se debe a que Warped está preparado para que cada proceso lógico simule de modo agresivo, sin importar el estado de sus vecinos. En cambio, con el protocolo conservador, los objetos deben tener todas sus colas de entrada completas para estar seguros de que procesarán un evento en forma correcta. Recomendamos que si se

desea un kernel de simulación pesimista eficiente, se reescriba al mismo, al igual que el grupo que implementó Warped reescribió este kernel para implementar el protocolo de NoTime.

Se observó en muchos casos que el agregar procesadores a una simulación no la aceleraba. Quedó claramente demostrado que lo dicho se debe principalmente a problemas de colisión en la red de comunicación. Creemos firmemente que de contar con una red más rápida y sin problemas de colisión de paquetes, las simulaciones podrían haber explotado mucho mejor el paralelismo en la ejecución de las mismas.

Cuando se simula un sistema lineal, en donde hay una dependencia total de un objeto con su anterior, no es útil ejecutar un protocolo de Modelo Frecuencia de Rollback, ya que la demora de un objeto hace que toda la simulación termine demorándose, no pudiéndose lograr una mejora efectiva respecto al TimeWarp original. Algo similar ocurrió con el protocolo de NoTime, ya que se observó que el inicio de estas simulaciones demoraban más que las de TimeWarp; luego, si la simulación no generaba muchos rollbacks o no tenía un overhead grande cuando cada objeto almacenaba su estado, el protocolo de NoTime demoraba más que TimeWarp.

El Modelo Frecuencia de Rollbacks Local por Pasos demostró ser de gran utilidad en aquellas simulaciones en donde había una ocurrencia media o alta de rollbacks. Este protocolo agrega restricciones al protocolo optimista (por lo que le agrega las mejoras de los protocolos conservadores), obteniendo de este modo mejores resultados que el protocolo de TimeWarp original. Si asumimos que en general las simulaciones tienen una gran cantidad de objetos y están formadas por estructuras complejas (no son una simple estructura lineal), este método resultó ser óptimo para la ejecución de simulaciones paralelas y distribuidas.

Con el protocolo Modelo Frecuencia de Rollbacks Global también se lograron resultados promisorios. Creemos que si contáramos con una mejor red de

comunicación, éste método tendría grandes posibilidades de mejorar notablemente al protocolo de TimeWarp, ya que permite tener un alto conocimiento del estado de los objetos de la simulación, y actuar adecuadamente en base a ello. Tiene como desventaja que cada un intervalo de tiempo todos los procesos lógicos deben comunicarse entre sí para enviarse la información de los objetos que contienen, por lo que la red de comunicación toma mucha importancia.
En general, los protocolos globales no son eficaces cuando hay pocos rollbacks en la simulación, ya que siempre un proceso lógico debe demorarse (el que tiene el objeto que sufrió más rollbacks). En cambio, los protocolos locales se demoran sólo cuando un proceso lógico supera un umbral de rollbacks, por lo que en una simulación con pocos rollbacks no sufrirán demora alguna.
Es importante destacar que el desempeño de los protocolos Modelo Frecuencia de Rollbacks depende de los valores definidos en las variables max y rfm_interval en el caso local, y rfm_interval en el caso global. Dado que encontrar los valores óptimos puede depender del tipo de simulación, en ocasiones es aconsejable ejecutar en primera instancia la simulación en menor escala y con menor cantidad de eventos, y probar diferentes combinaciones de variables para hallar los valores óptimos para la simulación tratada. Luego sí se puede ejecutar la simulación original con los protocolos citados.

El protocolo de NoTime puede ser de mucha utilidad en el caso de simulaciones que sean complejas, y en donde no importe en demasía la exactitud de los resultados obtenidos.

Tal como se explicó en toda la introducción teórica (sobre todo en Modelo Frecuencia de Rollbacks y Analogías entre tiempo y memoria virtual), no siempre es bueno que no haya rollbacks en los procesos lógicos. La ausencia de rollbacks puede indicar que una simulación no está avanzando en lugar de indicar una eficiente ejecución.

6.2. Trabajo futuro

Creemos que se cumplió ampliamente el objetivo de esta tesis. La idea de este trabajo no fue mostrar que más paralelismo de ejecución implicaba mejores tiempos de simulación (de hecho, por lo explicado respecto a la red de comunicación esto no pudo ser demostrado). Nuestra misión fue la de estudiar los protocolos existentes de simulación, estudiar el kernel de simulación warped, enriquecerlo para que pueda simular con otras técnicas de simulación, y luego mejorarlo para tratar de obtener mejores tiempos de ejecución de acuerdo al modelo a simular. Creemos que hemos cumplido con este fin, ya que ahora Warped puede simular con varios protocolos más (conservador y TimeWarp con Modelo Frecuencia de Rollbacks, locales y globales).

Sin embargo, sabemos que queda mucho por hacer. Sentimos que este trabajo es la base de un desarrollo mucho más amplio sobre el área de simulaciones, ya que se dejan asentadas las bases para un crecimiento armónico en el tema.

Entre las tareas que creemos se pueden continuar encontramos a:

- Probar distintas opciones del archivo config.hh de Warped. De este modo es posible ejecutar simulaciones con distintas estrategias de cancelación (agresiva, ociosa y dinámica), de envío de antimensajes por ocurrencia rollbacks, etc., y evaluar las diferencias en los tiempos de ejecución obtenidos.
- Implementar el protocolo Time Warp working set, el cual especificamos tras una comparación entre tiempo y memoria virtual, pero no hemos implementado.
- Modificar el protocolo de Modelo Frecuencia de Rollbacks Global para que cada proceso lógico envíe por la red sólo el mayor numero de rollbacks de sus objetos, o los n mayores. Entre las políticas de suspensión se podría demorar a los n mayores objetos generadores de rollbacks, en lugar de a uno solo.

- Ejecutar las simulaciones en diversas arquitecturas de computadoras, como pueden ser equipos multiprocesadores, o equipos de la talla de HP con sistema operativo HP-UX, IBM con sistema operativo AIX, Alpha con sistema operativo OSF1, etc.
- Ejecutar las simulaciones en redes que no sean broadcast como Ethernet (por ejemplo token-ring o ATM) para evitar así que la red sea el cuello de botella de las simulaciones.
- Implementar y experimentar con un protocolo Modelo Frecuencia de Rollbacks Global por pasos. El mismo se comportaría como el MFR Local por pasos, en donde un proceso lógico se demora hasta una cantidad fija de pasos, en lugar de todo el período de chequeo de rollbacks.
- Implementar un protocolo de Modelo Frecuencia de Rollbacks Global con mezcla del protocolo Local. El mismo se comportaría de igual modo al primero, pero no demoraría al proceso lógico que más rollbacks sufrió a menos que supere un umbral de rollbacks preestablecido (como lo especifica el Modelo Frecuencia de Rollbacks Local).
- Utilizar diferentes valores de N para las simulaciones de Modelo Frecuencia de Rollback Local por Pasos. En este trabajo se usó un N de 10 ciclos de simulación, por lo que variar este número de suspensiones puede redundar en mejores o peores tiempos de simulación.
- Especificar y programar simulaciones “grandes”. Todas las probadas en este trabajo son de pequeña escala, pero puede ser un trabajo arduo e interesante simular sistemas voluminosos y de mucha interacción, como, por ejemplo, el tránsito en calles de una ciudad.

Apéndice

Cuadros de tiempos y rollbacks de las simulaciones

Ping-pong

- Ping-pong best partition, equipos Sun netra, red de 10 Mhz

2 Procesos Lógicos	Tiempo 1	Tiempo 2	Tiempo 3	Promedio
TimeWarp	2.83	2.95	2.83	2.87
TimeWarp RFML	3.26	3.20	3.23	3.23
NoTime	2.69	2.59	2.71	2.66

3 Procesos Lógicos	Tiempo 1	Tiempo 2	Tiempo 3	Promedio
TimeWarp	4.25	4.36	4.28	4.30
TimeWarp RFML	4.60	4.63	4.64	4.62
NoTime	7.08	7.21	7.34	7.21

4 Procesos Lógicos	Tiempo 1	Tiempo 2	Tiempo 3	Promedio
TimeWarp	5.10	5.06	5.13	5.10
TimeWarp RFML	5.49	5.47	5.67	5.54
NoTime	10.16	10.33	10.41	10.30

5 Procesos Lógicos	Tiempo 1	Tiempo 2	Tiempo 3	Promedio
TimeWarp	6.03	6.05	6.12	6.07
TimeWarp RFML	6.44	6.54	6.41	6.46
NoTime	13.07	12.96	12.79	12.94

Ninguna simulación de ping-pong sufrió rollbacks

- Ping-pong worst partition, equipos Sun netra, red de 10 Mhz

2 Procesos Lógicos	Tiempo 1	Tiempo 2	Tiempo 3	Promedio
TimeWarp	46.42	44.29	44.39	45.03
TimeWarp RFML	48.70	49.02	48.41	48.71
NoTime	33.91	33.78	34.10	33.93

3 Procesos Lógicos	Tiempo 1	Tiempo 2	Tiempo 3	Promedio
TimeWarp	45.15	44.94	44.74	44.94
TimeWarp RFML	44.00	45.11	45.33	44.81
NoTime	41.69	41.63	41.43	41.58

4 Procesos Lógicos	Tiempo 1	Tiempo 2	Tiempo 3	Promedio
TimeWarp	48.43	48.00	48.18	48.20
TimeWarp RFML	46.95	48.47	48.27	47.90
NoTime	47.30	46.79	47.17	47.09

5 Procesos Lógicos	Tiempo 1	Tiempo 2	Tiempo 3	Promedio
TimeWarp	50.48	51.86	50.85	51.06
TimeWarp RFML	51.47	52.52	51.70	51.90
NoTime	52.60	52.45	52.19	52.41

- Ping-pong best partition, equipos Sun netra, red de 100 Mhz

1 Proceso Lógico	Tiempo 1	Tiempo 2	Tiempo 3	Promedio
Secuencial	0.39	0.37	0.38	0.38

2 Procesos Lógicos	Tiempo 1	Tiempo 2	Tiempo 3	Promedio
TimeWarp	2.67	2.66	2.65	2.66
TimeWarp RFML	2.80	2.90	2.81	2.84
NoTime	2.56	2.63	2.57	2.59

3 Procesos Lógicos	Tiempo 1	Tiempo 2	Tiempo 3	Promedio
TimeWarp	4.12	4.13	4.08	4.11
TimeWarp RFML	4.37	4.37	4.37	4.37
NoTime	6.94	6.98	6.98	6.97

4 Procesos Lógicos	Tiempo 1	Tiempo 2	Tiempo 3	Promedio
TimeWarp	4.95	4.98	5.02	4.98
TimeWarp RFML	5.40	5.43	5.39	5.41
NoTime	9.71	9.83	9.93	9.82

5 Procesos Lógicos	Tiempo 1	Tiempo 2	Tiempo 3	Promedio
TimeWarp	5.86	5.95	5.88	5.90
TimeWarp RFML	6.47	6.29	6.42	6.39
NoTime	12.62	12.78	12.35	12.58

- Ping-pong worst partition, equipos Sun netra, red de 100 Mhz

2 Procesos Lógicos	Tiempo 1	Tiempo 2	Tiempo 3	Promedio
TimeWarp	32.68	30.65	33.37	32.23
TimeWarp RFML	32.98	33.07	32.84	32.96
NoTime	25.40	25.68	25.32	25.47

3 Procesos Lógicos	Tiempo 1	Tiempo 2	Tiempo 3	Promedio
TimeWarp	34.66	35.19	35.36	35.07

TimeWarp RFML	34.79	35.36	35.15	35.10
NoTime	33.26	33.34	33.07	33.22

4 Procesos Lógicos	Tiempo 1	Tiempo 2	Tiempo 3	Promedio
TimeWarp	37.76	37.70	37.23	37.56
TimeWarp RFML	37.85	38.10	37.90	37.95
NoTime	38.13	37.64	38.65	38.14

5 Procesos Lógicos	Tiempo 1	Tiempo 2	Tiempo 3	Promedio
TimeWarp	39.93	39.83	39.65	39.80
TimeWarp RFML	40.59	40.69	40.87	40.72
NoTime	42.91	43.01	42.56	42.83

- Ping-pong best partition, equipos Linux, red de 10 Mhz

1 Proceso Lógico	Tiempo 1	Tiempo 2	Tiempo 3	Promedio
Secuencial 1	0.23	0.29	0.18	0.23
Secuencial 5	3.88	3.82	3.65	3.78

2 Procesos Lógicos	Tiempo 1	Tiempo 2	Tiempo 3	Promedio
TimeWarp 1-2	2.22	1.98	1.88	2.03
TimeWarp 1-3	2.94	3.17	2.90	3.00
TimeWarp 1-4	5.12	5.15	5.58	5.28
TimeWarp 1-5	25.39	25.70	25.09	25.39
TimeWarp 4-5	26.22	24.48	25.45	25.38
NoTime 1-2	2.32	2.76	2.18	2.42
NoTime 1-5	20.94	20.49	20.92	20.78
NoTime 4-5	20.65	20.18	20.64	20.49

3 Procesos Lógicos	Tiempo 1	Tiempo 2	Tiempo 3	Promedio

TimeWarp	3.04	2.96	2.93	2.98
NoTime	4.37	4.07	4.40	4.28

4 Procesos Lógicos	Tiempo 1	Tiempo 2	Tiempo 3	Promedio
TimeWarp	10.56	10.37	10.07	10.33
NoTime	6.41	6.31	6.48	6.40

5 Procesos Lógicos	Tiempo 1	Tiempo 2	Tiempo 3	Promedio
TimeWarp	17.44	17.31	16.51	17.09
NoTime	22.02	22.23	22.30	22.18

- Ping-pong worst partition, equipos Linux, red de 10 Mhz

2 Procesos Lógicos	Tiempo 1	Tiempo 2	Tiempo 3	Promedio
TimeWarp 1-2	42.67	42.83	43.52	43.01
TimeWarp 1-5	194.94	187.34	194.24	192.17
TimeWarp 4-5	248.71	251.20	248.74	249.55
NoTime 1-2	32.48	31.37	30.21	31.35
NoTime 1-5	181.77	193.39	195.74	190.30
NoTime 4-5	263.72	202.21	211.27	225.73

3 Procesos Lógicos	Tiempo 1	Tiempo 2	Tiempo 3	Promedio
TimeWarp	50.45	51.90	50.57	50.97
NoTime	41.89	42.35	44.07	42.77

4 Procesos Lógicos	Tiempo 1	Tiempo 2	Tiempo 3	Promedio
TimeWarp	57.31	55.22	54.75	55.76
NoTime	45.21	44.63	44.65	44.83

5 Procesos Lógicos	Tiempo 1	Tiempo 2	Tiempo 3	Promedio
TimeWarp	120.65	116.46	118.89	118.67
NoTime	124.87	125.55	118.69	123.04

Ninguna simulación de ping-pong sufrió rollbacks

Cartas

- Equipos Sun netra, red de 100 Mhz, 20000 eventos

4 Procesos Lógicos	Tiempo 1	Tiempo 2	Tiempo 3	Promedio
TimeWarp	9.81	9.96	9.95	9.91
TimeWarp RFMLS	10.35	10.07	10.12	10.18
TimeWarp RFMG	10.73	10.60	10.53	10.62

- Equipos Sun netra, red de 100 Mhz, 5000 eventos

4 Procesos Lógicos	Tiempo 1	Tiempo 2	Tiempo 3	Promedio
Pesimista	16.40	16.50	16.46	16.45
TimeWarp	2.56	2.53	2.53	2.54

- Equipos Linux, red de 10 Mhz, 20000 eventos

4 Procesos Lógicos	Tiempo 1	Tiempo 2	Tiempo 3	Promedio
TimeWarp	17.59	17.45	18.47	17.84
TimeWarp MFRL	118.18	159.41	17.47	98.35
TimeWarp MFRLS	17.36	17.71	18.24	17.77

TimeWarp MFRG	19.08	20.08	20.54	19.90

- Equipos Linux, red de 10 Mhz, 5000 eventos

4 Procesos Lógicos	Tiempo 1	Tiempo 2	Tiempo 3	Promedio
Pesimista	35.13	35.11	37.02	35.75
TimeWarp	4.48	4.24	4.12	4.28

Rollbacks: Los números en rojo y negrita indican que el proceso lógico sufrió al menos una demora debido al protocolo Modelo Frecuencia de Rollbacks utilizado

- Equipos Sun netra, red de 100 Mhz, 20000 eventos

4 Procesos Lógicos – **Rollbacks**	PL0	PL1	PL2	PL3	Total
TimeWarp	0	0	0	2	2
	0	0	0	2	2
	0	0	0	2	2
TimeWarp RFML	0	0	0	1	1
	0	0	0	2	2
	0	0	0	2	2
TimeWarp RFMG	0	0	0	3	3
	0	0	0	1	1
	0	0	0	2	2

- Equipos Sun netra, red de 100 Mhz, 5000 eventos

4 Procesos Lógicos – **Rollbacks**	PL0	PL1	PL2	PL3	Total
TimeWarp	0	0	0	2	2
	0	0	0	1	1
	0	0	0	1	1

- Equipos Linux, red de 10 Mhz, 20000 eventos

4 Procesos Lógicos – **Rollbacks**	PL0	PL1	PL2	PL3	Total
TimeWarp	0	0	0	6	6
	0	0	0	19	19
	0	0	0	86	86
TimeWarp MFRL	0	0	0	**10**	10
	0	0	0	**10**	10
	0	0	0	2	2
TimeWarp MFRLS	0	0	0	8	8
	0	0	0	**64**	64
	0	0	0	**92**	92
TimeWarp MFRG	0	0	0	2	2
	0	0	0	1	1
	0	0	0	2	2

- Equipos Linux, red de 10 Mhz, 5000 eventos

4 Procesos Lógicos – **Rollbacks**	PL0	PL1	PL2	PL3	Total
TimeWarp	0	0	0	4	4
	0	0	0	7	7

	0	0	0	26	26

Cartas2

- Equipos Sun netra, red de 10 Mhz

2 Procesos Lógicos	Tiempo 1	Tiempo 2	Tiempo 3	Promedio
TimeWarp	1.55	1.63	1.55	1.58
TimeWarp RFMLS 1	1.58	1.54	1.57	1.56

3 Procesos Lógicos	Tiempo 1	Tiempo 2	Tiempo 3	Promedio
TimeWarp	26.64	28.97	27.95	27.85
TimeWarp RFMLS 1	29.43	28.40	24.99	27.61

4 Procesos Lógicos	Tiempo 1	Tiempo 2	Tiempo 3	Promedio
TimeWarp	26.34	26.39	24.88	25.87
TimeWarp RFMLS 1	22.61	21.74	20.08	21.48

5 Procesos Lógicos	Tiempo 1	Tiempo 2	Tiempo 3	Promedio
TimeWarp	10.56	12.06	12.97	11.86
TimeWarp RFMLS 1	12.62	12.89	11.70	12.40
TimeWarp RFMLS 2	9.73	10.51	10.21	10.15
TimeWarp RFMLS 3	11.72	11.51	10.94	11.39

Rollbacks: Los números en rojo y negrita indican que el proceso lógico sufrió al menos una demora debido al protocolo Modelo Frecuencia de Rollbacks utilizado

2 Procesos Lógicos – **Rollbacks**	PL0	PL1	Total
TimeWarp	0	0	0
	0	0	0
	0	0	0
TimeWarp RFMLS 1	0	0	0
	0	0	0
	0	0	0

3 Procesos Lógicos – **Rollbacks**	PL0	PL1	PL2	Total
TimeWarp	14	41	10	65
	18	50	13	81
	10	52	4	66
TimeWarp RFMLS 1	12	**70**	6	88
	8	**73**	3	84
	12	**49**	4	65

4 Procesos Lógicos – **Rollbacks**	PL0	PL1	PL2	PL3	Total
TimeWarp	10	4	27	16	57
	7	2	22	6	37
	8	3	20	8	39
TimeWarp RFMLS 1	14	7	**17**	**33**	71
	22	**12**	**15**	28	77
	9	6	**13**	**39**	67

5 Procesos Lógicos – **Rollbacks**	PL0	PL1	PL2	PL3	PL4	Total

TimeWarp	30	12	11	23	153	229
	30	15	17	12	224	298
	37	11	16	27	217	308
TimeWarp RFMLS 1	51	**26**	**23**	**27**	**157**	284
	18	9	**15**	**16**	**195**	253
	42	**19**	**21**	**22**	**172**	276
TimeWarp RFMLS 2	23	**12**	**11**	**14**	**189**	249
	16	**11**	**13**	**17**	**177**	234
	22	**8**	**13**	**11**	**187**	241
TimeWarp RFMLS 3	34	15	14	14	**208**	285
	32	11	9	18	**91**	161
	30	17	11	15	**167**	240

- Equipos Sun netra, red de 100 Mhz

1 Proceso Lógico	Tiempo 1	Tiempo 2	Tiempo 3	Promedio
Secuencial	0.15	0.17	0.14	0.15

2 Procesos Lógicos	Tiempo 1	Tiempo 2	Tiempo 3	Promedio
TimeWarp	1.44	1.46	1.45	1.45
TimeWarp RFMLS 1	1.49	1.48	1.53	1.50
TimeWarp RFMG 1	1.63	1.65	1.62	1.63
NoTime	5.54	5.48	5.54	5.52

3 Procesos Lógicos	Tiempo 1	Tiempo 2	Tiempo 3	Promedio
TimeWarp	31.62	28.39	31.46	30.49
TimeWarp RFMLS 1	27.19	33.63	29.63	30.15
TimeWarp RFMG 1	38.73	35.94	36.87	37.18
NoTime	8.60	8.72	8.86	8.73

4 Procesos Lógicos	Tiempo 1	Tiempo 2	Tiempo 3	Promedio
TimeWarp	23.05	22.25	24.05	23.12
TimeWarp RFMLS 1	24.06	26.80	22.85	24.57
TimeWarp RFMG 1	26.09	26.91	25.70	26.23
NoTime	6.36	6.30	6.31	6.32

5 Procesos Lógicos	Tiempo 1	Tiempo 2	Tiempo 3	Promedio
TimeWarp	9.49	9.20	10.41	9.70
TimeWarp RFMLS 1	8.41	8.60	8.22	8.41
TimeWarp RFMLS 2	9.63	9.30	9.25	9.39
TimeWarp RFMLS 3	10.10	9.90	9.30	9.77
TimeWarp RFMLS 4	8.88	8.40	9.30	8.86
TimeWarp RFMG 1	7.85	9.16	8.36	8.46
TimeWarp RFMG 2	8.19	8.29	9.08	8.52
TimeWarp RFMG 3	10.22	9.68	9.65	9.85
TimeWarp RFMG 4	11.95	11.72	11.82	11.83
NoTime	6.63	6.86	6.80	6.76

Rollbacks: Los números en rojo y negrita indican que el proceso lógico sufrió al menos una demora debido al protocolo Modelo Frecuencia de Rollbacks utilizado

2 Procesos Lógicos – **Rollbacks**	PL0	PL1	Total
TimeWarp	0	0	0
	0	0	0
	0	0	0
TimeWarp RFMLS 1	0	0	0
	0	0	0

	0	0	0
TimeWarp RFMG 1	0	0	0
	0	0	0
	0	0	0

3 Procesos Lógicos – **Rollbacks**	PL0	PL1	PL2	Total
TimeWarp	3	73	2	78
	3	68	2	73
	3	67	7	77
TimeWarp RFMLS 1	3	**59**	2	64
	12	**66**	7	85
	3	**72**	2	77
TimeWarp RFMG 1	**3**	82	2	87
	2	81	2	85
	2	74	**3**	79

4 Procesos Lógicos – **Rollbacks**	PL0	PL1	PL2	PL3	Total
TimeWarp	84	20	17	246	367
	81	21	16	260	378
	78	22	19	268	387
TimeWarp RFMLS 1	**73**	**16**	**19**	**216**	324
	83	**24**	**19**	**252**	378
	83	**23**	**17**	**241**	364
TimeWarp RFMG 1	**16**	1	24	**78**	119
	11	0	27	**67**	105
	14	0	20	**73**	107

5 Procesos Lógicos – **Rollbacks**	PL0	PL1	PL2	PL3	PL4	Total
TimeWarp	72	19	26	97	226	440
	76	22	27	88	262	475
	77	28	39	66	194	404
TimeWarp RFMLS 1	61	6	**15**	**96**	**244**	422
	70	**15**	**19**	**99**	**261**	464
	53	7	**14**	**99**	**268**	441
TimeWarp RFMLS 2	68	12	**27**	**93**	**289**	489
	84	**20**	**18**	**100**	**239**	461
	49	**10**	**16**	**80**	**239**	394
TimeWarp RFMLS 3	64	12	16	**85**	**251**	428
	71	19	32	51	**288**	461
	71	11	16	**103**	**263**	464
TimeWarp RFMLS 4	**95**	**38**	**54**	**81**	**277**	545
	83	**23**	**44**	**97**	**270**	517
	92	**38**	**71**	**55**	**233**	489
TimeWarp RFMLG 1	**19**	**5**	**12**	**34**	**143**	213
	27	9	**23**	**65**	**188**	312
	19	**10**	**16**	**30**	**202**	277
TimeWarp RFMLG 2	**57**	**8**	**16**	**93**	**233**	407
	36	**2**	**18**	**74**	**253**	383
	61	**9**	**13**	**95**	**241**	419
TimeWarp RFMLG 3	12	1	30	74	**663**	780
	23	0	40	57	898	1018
	13	4	33	56	743	849
TimeWarp RFMLG 4	**15**	**11**	**12**	**14**	**65**	117
	36	**14**	**13**	**17**	**57**	137

	9	**9**	**14**	**12**	**28**	72

- Equipos Linux, red de 10 Mhz

1 Proceso Lógico	Tiempo 1	Tiempo 2	Tiempo 3	Promedio
Secuencial 1	0.5	0.48	0.48	0.49
Secuencial 5	11.65	11.38	11.62	11.55

2 Procesos Lógicos	Tiempo 1	Tiempo 2	Tiempo 3	Promedio
TimeWarp	2.23	2.41	2.35	2.33
TimeWarp RFMLS 1	2.39	2.41	2.37	2.39
TimeWarp RFMLS 2	2.34	2.27	2.37	2.33
TimeWarp RFMLS 3	2.35	2.38	2.34	2.36
TimeWarp RFMG 1	2.24	2.28	2.11	2.21
TimeWarp RFMG 2	2.17	2.17	2.22	2.19
TimeWarp RFMG 3	3.76	3.29	3.25	3.43
TimeWarp RFMG 4	2.03	2.03	2.12	2.06

3 Procesos Lógicos	Tiempo 1	Tiempo 2	Tiempo 3	Promedio
TimeWarp	10.49	8.27	9.69	9.48
TimeWarp RFMLS 1	8.23	8.68	8.18	8.36
TimeWarp RFMLS 2	8.91	7.47	9.39	8.59
TimeWarp RFMLS 3	9.77	10.73	11.20	10.57
TimeWarp RFMG 1	4.71	5.00	4.34	4.68
TimeWarp RFMG 2	4.74	4.75	4.77	4.75
TimeWarp RFMG 3	13.57	13.38	13.72	13.56
TimeWarp RFMG 4	8.29	5.45	5.22	6.32

4 Procesos Lógicos	Tiempo 1	Tiempo 2	Tiempo 3	Promedio
TimeWarp	6.17	5.97	5.79	5.98

TimeWarp RFMLS 1	5.37	5.77	5.81	5.65
TimeWarp RFMLS 2	5.92	5.77	5.85	5.85
TimeWarp RFMLS 3	6.04	5.80	5.69	5.84
TimeWarp RFMG 1	6.42	5.50	6.35	6.09
TimeWarp RFMG 2	6.82	6.66	6.48	6.65
TimeWarp RFMG 3	29.59	29.64	30.28	29.84
TimeWarp RFMG 4	8.71	5.46	6.36	6.84

5 Procesos Lógicos	Tiempo 1	Tiempo 2	Tiempo 3	Promedio
TimeWarp	43.40	42.64	40.38	42.14
TimeWarp RFMLS 1	41.36	40.73	48.65	43.58
TimeWarp RFMLS 2	44.73	35.13	45.33	41.73
TimeWarp RFMLS 3	46.44	48.33	50.49	48.42
TimeWarp RFMG 1	138.21	110.32	135.15	127.89
TimeWarp RFMG 2	144.40	137.39	131.49	137.76
TimeWarp RFMG 3	225.66	235.35	236.74	232.58
TimeWarp RFMG 4	118.97	95.62	96.28	103.62

Rollbacks: Los números en rojo y negrita indican que el proceso lógico sufrió al menos una demora debido al protocolo Modelo Frecuencia de Rollbacks utilizado

2 Procesos Lógicos – **Rollbacks**	PL0	PL1	Total
TimeWarp	0	111	111
	0	128	128
	0	119	119
TimeWarp RFMLS 1	0	100	100
	0	124	124
	0	136	136
TimeWarp RFMLS 2	0	123	123

	0	112	112
	0	162	162
TimeWarp RFMLS 3	0	149	149
	0	128	128
	0	119	119
TimeWarp RFMG 1	0	7	7
	0	7	7
	0	7	7
TimeWarp RFMG 2	0	7	7
	0	7	7
	0	7	7
TimeWarp RFMG 3	0	24	24
	0	4	4
	0	2	2
TimeWarp RFMG 4	0	7	7
	0	7	7
	0	7	7

3 Procesos Lógicos – **Rollbacks**	PL0	PL1	PL2	Total
TimeWarp	34	19	155	208
	31	14	94	139
	70	22	88	180
TimeWarp RFMLS 1	41	10	**98**	149
	38	**24**	**62**	124
	62	**20**	16	98
TimeWarp RFMLS 2	30	**20**	**104**	154
	44	**19**	**19**	82
	48	**20**	**78**	146
TimeWarp RFMLS 3	26	19	**125**	170

	43	13	**180**	236
	43	17	**155**	215
TimeWarp RFMG 1	**29**	**19**	**25**	73
	12	**22**	**12**	46
	23	**16**	**19**	58
TimeWarp RFMG 2	5	67	8	80
	4	64	2	70
	12	50	9	71
TimeWarp RFMG 3	0	**83**	1	84
	6	82	3	91
	0	88	0	88
TimeWarp RFMG 4	**43**	**43**	**6**	92
	4	52	2	58
	4	67	**3**	74

4 Procesos Lógicos – **Rollbacks**	PL0	PL1	PL2	PL3	Total
TimeWarp	10	4	0	351	365
	6	2	0	349	357
	7	5	0	360	372
TimeWarp RFMLS 1	8	0	0	**316**	324
	4	2	0	**351**	357
	7	7	0	**340**	354
TimeWarp RFMLS 2	7	5	0	**365**	377
	7	4	0	**355**	366
	3	0	0	**331**	334
TimeWarp RFMLS 3	17	7	0	**390**	414
	2	2	0	**322**	326
	3	1	0	**341**	345
TimeWarp RFMG 1	2	1	3	**198**	204

	4	4	9	**37**	54
	4	1	7	**275**	287
TimeWarp RFMG 2	2	2	15	4	23
	2	0	10	**81**	93
	0	0	6	103	109
TimeWarp RFMG 3	1	**1**	45	1	48
	0	0	50	0	50
	2	2	40	6	50
TimeWarp RFMG 4	8	**6**	**17**	**10**	41
	3	1	16	**258**	278
	3	2	7	**122**	134

5 Procesos Lógicos – **Rollbacks**	PL0	PL1	PL2	PL3	PL4	Total
TimeWarp	0	0	36	74	0	110
	0	0	42	89	0	131
	0	0	41	73	2	114
TimeWarp RFMLS 1	0	0	**46**	**71**	0	117
	0	0	**42**	**80**	1	122
	0	0	**45**	**78**	0	123
TimeWarp RFMLS 2	0	0	**42**	**75**	0	117
	0	0	**43**	**75**	0	118
	0	0	**41**	**85**	0	126
TimeWarp RFMLS 3	0	0	45	**77**	0	122
	0	0	43	**89**	0	132
	0	0	42	74	0	116
TimeWarp RFMG 1	0	0	35	**68**	0	103
	0	0	**33**	**74**	0	107
	0	0	**36**	72	0	108
TimeWarp RFMG 2	0	0	28	73	0	101

	0	0	33	67	0	100
	0	0	38	79	0	117
TimeWarp RFMG 3	0	0	34	88	0	122
	0	0	36	85	0	121
	0	0	37	97	0	134
TimeWarp RFMG 4	0	0	**33**	**75**	0	108
	0	0	**34**	**58**	0	92
	0	0	**34**	**66**	0	100

Cartas3

- Equipos Sun netra, red de 100 Mhz

4 Procesos Lógicos	Tiempo 1	Tiempo 2	Tiempo 3	Promedio
Pesimista	30.68	30.62	30.70	30.67
TimeWarp	1.96	1.96	1.86	1.93
TimeWarp RFML	90.85	69.04	1.84	53.91
TimeWarp RFMLS	1,85	2.82	1.91	2.19
TimeWarp RFMG	2.50	2.23	2.46	2.40
NoTime	2.23	2.14	2.54	2.30

Rollbacks: Los números en rojo y negrita indican que el proceso lógico sufrió al menos una demora debido al protocolo Modelo Frecuencia de Rollbacks utilizado

4 Procesos Lógicos – **Rollbacks**	PL0	PL1	PL2	PL3	Total
TimeWarp	0	11	10	0	21
	1	4	0	0	5
	0	8	3	0	11

TimeWarp RFML	4	**15**	1	0	16
	3	**13**	6	0	22
	0	3	1	0	4
TimeWarp RFMLS	1	9	3	0	12
	36	**47**	3	0	86
	1	6	3	0	10
TimeWarp RFMG	2	**8**	**3**	0	13
	1	11	4	0	16
	2	11	**5**	1	19

- Equipos Linux, red de 10 Mhz

4 Procesos Lógicos	Tiempo 1	Tiempo 2	Tiempo 3	Promedio
Pesimista	92.57	93.42	93.24	93.08
TimeWarp	2.73	2.54	2.67	2.65
TimeWarp RFML	124.63	123.85	119.38	122.62
TimeWarp RFMLS	2.45	2.91	2.92	2.76
TimeWarp RFMG	42.09	49.40	41.15	44.21
NoTime	1.95	1.97	1.94	1.95

Rollbacks: Los números en rojo y negrita indican que el proceso lógico sufrió al menos una demora debido al protocolo Modelo Frecuencia de Rollbacks utilizado

4 Procesos Lógicos – **Rollbacks**	PL0	PL1	PL2	PL3	Total
TimeWarp	31	210	99	0	340
	24	216	103	0	343

	42	219	96	0	357
TimeWarp RFML	4	**15**	6	0	25
	0	**12**	7	0	19
	4	**21**	**13**	0	38
TimeWarp RFMLS	23	**168**	**85**	0	276
	21	**206**	**90**	0	317
	15	**185**	**93**	0	293
TimeWarp RFMG	**186**	**516**	**189**	**1**	892
	184	**536**	**205**	**0**	925
	151	**489**	**200**	**3**	843

HTTP

- Equipos Sun netra, red de 100 Mhz; 5 servers, 15 clientes y 100 requerimientos

2 Procesos Lógicos	Tiempo 1	Tiempo 2	Tiempo 3	Promedio
Pesimista	50.62	51.37	50.54	50.84
TimeWarp	1.28	1.28	1.19	1.25

3 Procesos Lógicos	Tiempo 1	Tiempo 2	Tiempo 3	Promedio
TimeWarp	66.95	65.53	64.25	65.58
TimeWarp RFML	1.11	1.02	1.27	1.13

4 Procesos Lógicos	Tiempo 1	Tiempo 2	Tiempo 3	Promedio
Pesimista	65.94	66.70	65.80	66.15
TimeWarp	1.18	1.03	1.04	1.08
TimeWarp RFML	1.19	1.04	1.06	1.10

Rollbacks: Los números en rojo y negrita indican que el proceso lógico sufrió al menos una demora debido al protocolo Modelo Frecuencia de Rollbacks utilizado

2 Procesos Lógicos – **Rollbacks**	PL0	PL1	Total
TimeWarp	263	41	304
	281	27	308
	241	13	254

3 Procesos Lógicos – **Rollbacks**	PL0	PL1	PL2	Total
TimeWarp	25	9	6	40
	10	3	4	17
	113	98	2	213

4 Procesos Lógicos – **Rollbacks**	PL0	PL1	PL2	PL3	Total
TimeWarp	8	0	15	9	32
	2	0	1	1	4
	4	0	3	1	8
TimeWarp RFML	8	19	0	13	40
	0	0	0	0	0
	1	0	1	0	2

- Equipos Sun netra, red de 100 Mhz; 5 servers, 15 clientes y 10 requerimientos

2 Procesos Lógicos	Tiempo 1	Tiempo 2	Tiempo 3	Promedio
Pesimista	4.95	4.9	4.89	4.91
TimeWarp	0.13	0.13	0.15	0.14

3 Procesos Lógicos	Tiempo 1	Tiempo 2	Tiempo 3	Promedio
TimeWarp	6.21	6.27	6.74	6.41
TimeWarp RFML	0.11	0.11	0.1	0.11

4 Procesos Lógicos	Tiempo 1	Tiempo 2	Tiempo 3	Promedio
Pesimista	6.64	6.38	6.48	6.50
TimeWarp	0.11	0.11	0.11	0.11

Rollbacks: Los números en rojo y negrita indican que el proceso lógico sufrió al menos una demora debido al protocolo Modelo Frecuencia de Rollbacks utilizado

2 Procesos Lógicos – **Rollbacks**	PL0	PL1	Total
TimeWarp	25	1	26
	26	2	28
	29	4	33

3 Procesos Lógicos – **Rollbacks**	PL0	PL1	PL2	Total
TimeWarp	4	1	2	7
	4	0	2	6
	3	0	1	4

4 Procesos Lógicos – **Rollbacks**	PL0	PL1	PL2	PL3	Total
TimeWarp	2	0	2	0	4
	2	0	2	0	4
	2	0	2	0	4

- Equipos Sun netra, red de 100 Mhz; 5 servers, 15 clientes y 1000 requerimientos

4 Procesos Lógicos	Tiempo 1	Tiempo 2	Tiempo 3	Promedio
Pesimista	665.97	654.84	654.72	658.51
TimeWarp	10.44	10.72	10.78	10.65

Rollbacks: Los números en rojo y negrita indican que el proceso lógico sufrió al menos una demora debido al protocolo Modelo Frecuencia de Rollbacks utilizado

4 Procesos Lógicos – **Rollbacks**	PL0	PL1	PL2	PL3	Total
TimeWarp	27	17	37	23	104
	80	58	47	60	245
	104	44	45	1	194

- Equipos Linux, red de 10 Mhz; 5 servers, 15 clientes y 100 requerimientos

2 Procesos Lógicos	Tiempo 1	Tiempo 2	Tiempo 3	Promedio
Pesimista	6.16	6.39	5.99	6.18
TimeWarp	0.17	0.15	0.14	0.15

3 Procesos Lógicos	Tiempo 1	Tiempo 2	Tiempo 3	Promedio
TimeWarp	7.75	8.58	7.74	8.02
TimeWarp RFML	0.28	0.24	0.19	0.24

4 Procesos Lógicos	Tiempo 1	Tiempo 2	Tiempo 3	Promedio
Pesimista	8.45	8.81	8.54	8.60

TimeWarp	0.23	0.24	0.42	0.30

5 Procesos Lógicos	Tiempo 1	Tiempo 2	Tiempo 3	Promedio
Pesimista	19.42	18.74	19.14	19.10
TimeWarp	1.01	0.59	0.85	0.82

Rollbacks: Los números en rojo y negrita indican que el proceso lógico sufrió al menos una demora debido al protocolo Modelo Frecuencia de Rollbacks utilizado

2 Procesos Lógicos – **Rollbacks**	PL0	PL1	Total
TimeWarp	5	5	10
	8	4	12
	5	1	6

3 Procesos Lógicos – **Rollbacks**	PL0	PL1	PL2	Total
TimeWarp	18	16	4	38
	19	15	8	42
	13	6	5	24

4 Procesos Lógicos – **Rollbacks**	PL0	PL1	PL2	PL3	Total
TimeWarp	8	0	7	4	19
	23	3	11	9	46
	29	14	32	4	79

5 Procesos Lógicos – **Rollbacks**	PL0	PL1	PL2	PL3	PL4	Total
TimeWarp	53	16	21	20	0	110
	57	11	21	26	0	115
	77	20	32	32	1	161

- Equipos Linux, red de 10 Mhz; 5 servers, 15 clientes y 10 requerimientos

2 Procesos Lógicos	Tiempo 1	Tiempo 2	Tiempo 3	Promedio
Pesimista	60.67	60.93	61.06	60.89
TimeWarp	1.12	1.14	1.13	1.13

3 Procesos Lógicos	Tiempo 1	Tiempo 2	Tiempo 3	Promedio
TimeWarp	75.77	75.02	75.87	75.55
TimeWarp RFML	2.17	1.95	2.35	2.16

4 Procesos Lógicos	Tiempo 1	Tiempo 2	Tiempo 3	Promedio
Pesimista	83.01	83.84	83.33	83.39
TimeWarp	3.06	3.00	3.59	3.22

5 Procesos Lógicos	Tiempo 1	Tiempo 2	Tiempo 3	Promedio
Pesimista	189.95	172.68	173.74	178.79
TimeWarp	5.32	5.84	5.94	5.70

Rollbacks: Los números en rojo y negrita indican que el proceso lógico sufrió al menos una demora debido al protocolo Modelo Frecuencia de Rollbacks utilizado

2 Procesos Lógicos – **Rollbacks**	PL0	PL1	Total
TimeWarp	5	5	10
	8	4	12
	5	1	6

3 Procesos Lógicos – **Rollbacks**	PL0	PL1	PL2	Total
TimeWarp	18	16	4	38
	19	15	8	42
	13	6	5	24

4 Procesos Lógicos – **Rollbacks**	PL0	PL1	PL2	PL3	Total
TimeWarp	8	0	7	4	19
	23	3	11	9	46
	29	14	32	4	79

5 Procesos Lógicos – **Rollbacks**	PL0	PL1	PL2	PL3	PL4	Total
TimeWarp	53	16	21	20	0	110
	57	11	21	26	0	115
	77	20	32	32	1	161

Bibliografía

[Cha79] CHANDY, K; MISRA, J. "Distributed simulation: a case study in design and verification of distributed programs", IEEE TOSE, September 1979.

[Fuj90] FUJIMOTO, R. M. Parallel discrete event simulation. Commun. ACM 33, 30-53. October 1990.

[Gia96] GIAMBIASI, N. "Introduction à la modélisation et à la simulation". Materiales del curso de D.E.A.; Université d'Aix-Marseille III. 1996.

[Ho89] HO, Y. "Special issue on discrete event dynamic systems", Proceedings of the IEEE, 77 (1), 1989.

[Jef85] JEFFERSON, D. "Virtual Time". ACM TOPLS, 7(3): 404-425. July 1985.

[Pan93] PANCERELLA, C. M., REYNOLDS, P. F., Jr. Disseminating critical target-specific synchronization information in parallel discrte event simulations. En *Proceedings of the Seventh Workshop on Parallel and Distributed Simulation* (May), 52-59. 1993.

[Mis86] MISRA, J. "Distributed discrete-event simulations". ACM Computing surveys. Vol. 18, No. 1, 39-65. 1986.

[MMRW97] MARTIN, D.; McBrayer, T.; Radhakrishnan, R.; Wilsey, P. "TimeWarp Parallel Discrete Event Simulator". Technical Report. Computer Architecture Design Laboratory, University of Cincinnati. Diciembre de 1997.

[Nic90] NICOL, D. M., REYNOLDS, P. F., Jr. Optima dynamin remapping of parallel computations. IEEE Trans. Comput. 39, 206-219. February 1990.

[Rey88] REYNOLDS, P. F., Jr. A spectrum of options for parallel simulation . En *Proceedings of the 1988 Winter Simulation Conference*, 325-332. December 1988.

[Rey et al 93] REYNOLDS, P. F., Jr., PANCERELLA, C. M., SRINIVASAN S. Design and Performance analysis of hardware support for parallel simulations. J. Parallel Distrib. Comput. 18, 435-453. 1993.

[RTRW98] Rao D. M.; Thondugulam N.; Radhakrishnan, R.; Wilsey, P. Unsynchronized parallel discrete event simulation.

[Sil98] Silberchatz, Sistemas Operativos (Completar) Operating Systems Concepts, Silberchatz, Galvin, Ed. Adisson-Wesley, 5a. edición, 1998

[Sri98] SRINIVASAN S.; REYNOLDS, J. "Elastic time". ACM Transactions on Modeling and Computer Simulation. Vol. 8, No. 2. 103-139. April 1998.

[Wai96] WAINER, G. "Introducción a la simulación de sistemas de eventos discretos". Informe técnico N° 96-005. Departament o de Computación. FCEN/UBA. 1996.

[Wai 98] WAINER, G. "Discrete event cellular models with explicit delays". Ph.D. Thesis. DIAM/IUSPIM. Université d'Aix-Marseille III. 1998.

[Wul69] WULF, W. Performance monitors for multiprogramming systems. Proceedings of the 2nd. ACM symposium on Operating Systems Principles. 1969. pp. 175-181.