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

Tesis de Licenciatura

"Análisis de Modelos Transaccionales Móviles"

Autores

Viviana Hilda Ortiz vivortiz@movi.com.ar
Maximiliano Damián Contieri mcsee@ciudad.com.ar
Jorge Alejandro Handl jhandl@yahoo.com

Directores

Dr. Miguel Felder mfelder@pragma.com.ar
Lic. Eduardo Rodríguez eduardo@dc.uba.ar

Colaboradores

Lic. Andrés Coratella acoratella@hotmail.com
Lic. Roberto Hirsch rhirsch@perfil.com.ar

2003

Agradecimientos

Esta tesis no hubiera sido posible sin la fundamental colaboración de los licenciados Andrés Coratella y Roberto Hirsch quienes nos guiaron, apoyaron y ayudaron durante el desarrollo de la misma.

Agradecemos a nuestros directores Eduardo Rodríguez y Miguel Felder por su guía y apoyo.

Viviana agradece a sus padres, familia y amigos por su apoyo y comprensión a lo largo de toda la carrera.

Jorge Handl agradece a su esposa Gisela por su paciencia y comprensión y a su hijo Federico por su invalorable aporte de alegría.

Maximiliano Contieri agradece a su mujer Virginia y a sus padres por el constante apoyo y paciencia durante el transcurso de la presente tesis.

Tabla de Contenido

Índice de Tablas

Índice de Figuras

Resumen

En este trabajo se estudian distintos modelos transaccionales móviles existentes, se los compara y evalúa cuantitativamente para poder identificar claramente sus cualidades bajo diferentes situaciones y para analizar las soluciones que proponen a los problemas típicos de ambientes móviles.

Para ello, se desarrolla un simulador que permite representar los distintos modelos y someterlos a diferentes entornos para evaluar cómo reaccionan ante variaciones en características peculiares de los ambientes móviles tales como frecuencia de desconexión o de salto de celda.

Cada modelo es sometido a las mismas condiciones externas, para asegurar una comparación ecuánime. Los datos obtenidos a partir de esas simulaciones son presentados en forma gráfica y se utilizan para estudiar los modelos, contrastar su comportamiento teórico con el observado y compararlos desde dos puntos de vista: el tecnológico y el de un usuario final de sistemas transaccionales móviles.

En este trabajo se presentan los siguientes resultados: un simulador que puede ser utilizado para implementar nuevos modelos, verificar sus características y compararlos con los ya existentes; una nueva clasificación de niveles de soporte a la desconexión como resultado de completar aspectos subespecificados de los modelos elegidos; y la comparación de cuatro modelos móviles y uno no móvil, obteniendo una nueva clasificación en base a los resultados empíricos.

Abstract

In the present work several existing mobile transaction models are studied, compared and quantitatively evaluated in order to clearly identify their qualities under different situations and to analyze the solutions these models propose to the problems that are typical of mobile environments.

To this end, a simulator is developed that allows representing these models, test them under different situations and evaluate their behavior when specific characteristics of mobile environments are changed (i.e., the frequency of disconnections or cell handoff).

Each model is tested under the same external conditions to ensure a fair comparison. The data gathered from these simulations is presented graphically and is applied to study the models, contrast their observed behavior with the theory and compare them from two points of view: the technological one, and that of a user of a mobile transaction system.

The results of this work include: a simulator useful to implement new models, verify their properties and compare them with existing ones; a new classification concerning the level of disconnection support, due to the completion of sub-specified issues in the selected models; and the comparison of four mobile and one non-mobile models, resulting in a new empirical classification.

Capítulo 1: Introducción

La aparición de dispositivos móviles (teléfonos celulares, computadoras móviles, aplicaciones satelitales, etc.) así como el avance en el desarrollo de redes inalámbricas en los últimos tiempos, permite explorar un nuevo campo que hasta hace pocos años se encontraba limitado por los requerimientos de hardware.

Este nuevo contexto tecnológico presenta dificultades para la implementación de sistemas transaccionales. Si bien éstos han venido cumpliendo satisfactoriamente con muchos de los requerimientos de software en las últimas cuatro décadas, sus restricciones resultan ser muy fuertes para su aplicación en redes móviles, cuya problemática difiere sensiblemente de la de un sistema de terminales fijas permanentemente conectadas a una base de datos.

Entre las características propias y únicas de los ambientes móviles que impactan notablemente sobre una transacción se destacan las desconexiones frecuentes, la movilidad y las limitaciones de capacidad de los dispositivos móviles (batería, memoria, velocidad de transmisión de datos, etc). Esto ha motivado la creación de nuevos modelos transaccionales móviles que extienden el modelo tradicional, relajando, en consecuencia, las propiedades ACID de las transacciones [Pit94].

Diversas publicaciones han presentado modelos transaccionales móviles, que proponen soluciones teóricas a algunos problemas del entorno móvil. Estos modelos fueron estudiados y clasificados en [Cor2000], sin embargo no existen implementaciones ni estudios comparativos experimentales.

Este trabajo presenta un estudio de dichas características. Para desarrollarlo, analizamos distintas técnicas de medición y determinamos que la más apropiada para este problema es la de simulación de eventos discretos. Analizando simuladores libremente disponibles, observamos que no cubren las necesidades de este estudio. Por lo tanto, desarrollamos un simulador específico para este problema.

Mediante el uso del simulador desarrollado, instanciamos y comparamos algunos modelos, lo que nos permitió analizar sus cualidades y debilidades bajo diferentes situaciones, utilizando datos concretos.

Al estudiar los modelos propuestos en la literatura, y para poder compararlos en un mismo contexto, fue necesario completar su especificación debido a que no proponen soluciones a todos los problemas móviles [Cor2000] ni con el mismo nivel de detalle.

Este trabajo presenta tres aportes originales: en primer lugar provee un simulador que permite comparar modelos y brindar soporte en la investigación y desarrollo de nuevos modelos teóricos, validándolos y mostrando cuantitativa y gráficamente en qué aspectos éstos son mejores que los modelos previos o miediendo el impacto de una modificación a un modelo. También propone una nueva clasificación de niveles de soporte a la desconexión como resultado de completar aspectos subespecificados de los modelos elegidos. Finalmente, compara cuatro modelos móviles y uno no móvil en función de la robustez, la eficiencia y el costo desde el punto de vista tecnológico y del usuario obteniendo una nueva clasificación en base a los resultados empíricos.

1.1 Estado del arte actual

La aparición de dispositivos móviles (teléfonos celulares, pagers, computadoras móviles) así como el avance en el desarrollo de redes inalámbricas en los últimos años permite explorar un nuevo campo que, hasta hace una década, estaba limitado por la tecnología existente.

Las computadoras móviles actuales varían considerablemente en cuanto a capacidad de memoria, procesamiento, comunicaciones y batería: desde los teléfonos celulares, con sus restricciones de tamaño y peso, hasta los componentes móviles de un sistema de defensa, ubicados en todo tipo de vehículos. Todos estos componentes enfrentan dos problemas básicos: la reconfiguración dinámica de su conexión a la red al

cambiar de celda y, por ende, de estación base; y las desconexiones imprevistas por diversos problemas en la comunicación inalámbrica. Los componentes de menor porte se caracterizan, además, por la carencia de recursos computacionales, baja autonomía por falta de una fuente de alimentación permanente y lentitud en las comunicaciones por disponer de un ancho de banda reducido.

Todas estas limitaciones impactan sobre los sistemas transaccionales tradicionales, que requieren considerable capacidad de procesamiento y de comunicaciones, dependen de la corta duración de las transacciones para ofrecer una mayor capacidad de concurrencia, y suponen una conexión permanente y fija a la base de datos.

En los últimos años se han propuesto diferentes modelos teóricos que tratan de resolver la problemática transaccional aplicada al ambiente móvil. Sin embargo, según pudimos determinar, aún no se han realizado implementaciones concretas.

1.2 Perspectiva histórica

Las transacciones móviles se basan en dos grandes nichos tecnológicos en constante desarrollo desde hace varias décadas: las comunicaciones inalámbricas y los sistemas transaccionales. Sin embargo, hasta hace poco no se podían combinar para proveer sistemas transaccionales móviles, debido a limitaciones en ambas tecnologías.

Inicialmente, los dispositivos móviles no disponían de vínculos de comunicación inalámbrica (computadoras portátiles) o bien carecían de potencia de cálculo (radios y teléfonos celulares). Los primeros teléfonos móviles aparecieron en 1946 cuando AT&T introdujo el sistema MTS, en el que el usuario debía sintonizar manualmente una frecuencia libre y luego utilizar el aparato a la manera de un walkie-talkie. Desde entonces, la tecnología pasó por numerosos “estándares”, mutuamente incompatibles, para aumentar la capacidad de comunicaciones simultáneas dentro del mismo área de cobertura y agregar servicios más allá de la comunicación de voz, por ejemplo la transmisión de texto y video [Dal2000].

Por su parte, en la historia de las computadoras personales móviles pueden contarse varios hitos significativos: la aparición de la primera computadora portable (la Osborne I en 1981, de 11 Kg, 1 MHz y 64 Kb [Imp99]), la introducción de la primera “laptop” (la IBM PC Convertible en 1986, de 5 Kg, 5 MHz y 256 Kb [Freu99]), el primer *PDA* o Asistente Digital Personal (la Pilot 1000 de U.S.Robotics en 1996, de 160 gramos, 5 MHz y 128 Kb [Me2001]).

Con la aparición de los teléfonos celulares con tecnología CDMA introducidos por Qualcomm en 1995 y la PalmVII en 1999, la computación móvil con capacidad de comunicación inalámbrica ha alcanzado al público general, creando una gran demanda de aplicaciones para impulsar las ventas.

Una de estas aplicaciones es el acceso a Internet. Para proveer la infraestructura necesaria para acomodar las demandas de la computación móvil, se desarrollaron nuevos protocolos o extensiones a protocolos conocidos, como Mobile IP, explicados en [Ioa91], la diseminación de datos en redes con ancho de banda limitado, así como la modelización de la utilización de datos dependientes de la ubicación (búsqueda del hospital más cercano, la distancia más corta, etc.) [Re99]

Paralelamente a estos desarrollos, los sistemas transaccionales también han evolucionado. En 1976 surgieron las primeras teorías de bases de datos transaccionales desarrolladas por Eswaran y por Gray. Las primeras implementaciones proveían los servicios necesarios evitando el sistema operativo (CICS de IBM en 1968) o ampliando el mismo (MVS de DEC en 1978). Más adelante, en respuesta a las limitaciones de los modelos transaccionales tradicionales, se desarrollaron modelos más avanzados, por ejemplo: transacciones anidadas [Grey93] implementadas en el sistema Camelot en 1991, transacciones con savepoints implementadas en System R en 1979, etc.

La confluencia de todos estos desarrollos permite inferir que en un futuro próximo un gran número de computadoras móviles con capacidad operativa relativamente reducida (en comparación con los puestos fijos) podrán acceder a bases de datos centralizadas a través de conexiones inalámbricas [Bar99]. En [Imi92] se

describe un escenario posible del desarrollo de aplicaciones donde el factor más importante son las consultas a bases de datos en ambientes móviles. Es probable que un gran número de nuevas aplicaciones sean desarrolladas como consecuencia de las posibilidades que ofrecen los entornos móviles.

Para hacer posible este escenario, es imprescindible superar los problemas que presentan los entornos móviles a los sistemas transaccionales, como desconexiones o desplazamientos (estos problemas se describen en mayor detalle en <1.4 Características de un entorno móvil>). Sin embargo, no parece existir consenso en la comunidad académica acerca de cómo modelar la movilidad intrínseca de un usuario en una red de comunicación inalámbrica [Roc2000] .

1.3 Transacciones y las propiedades ACID

Las transacciones existen desde hace milenios. Una transacción es un contrato entre personas o entre una persona y una empresa, por el cual se intercambia algo. Los objetos involucrados en el intercambio pueden ser dinero, productos, servicios, información, etc.

Con la aparición de las computadoras nació el concepto de transacción de base de datos para soportar la registración en forma escalable, confiable y a menor costo. Una transacción de base de datos consiste en la ejecución de un programa que realiza una función administrativa por medio del acceso a una base datos compartida [Ber97]. Los programas transaccionales generalmente efectúan tres tareas:

1. Obtener información de entrada o input a través de una pantalla o de algún otro dispositivo de lectura (sensores, lectores de códigos de barra, etc).
2. Realizar el trabajo o funcionalidad que fue requerida.
3. Producir una respuesta, retornándola generalmente a través del mismo dispositivo que proveyó la información de entrada. [Ber97]

Cada invocación a un programa transaccional es una unidad de trabajo que se ejecuta una vez y produce resultados permanentes, almacenados en la base de datos compartida.

La idea fundamental de las transacciones está basada en las propiedades *ACID* (acrónimo de Atomicidad, Consistencia, aIslamiento y Durabilidad). Estas son las reglas que definen un modelo transaccional tradicional. A pesar de ser aceptadas por consenso en el ámbito académico y en la industria, hemos encontrado sutiles diferencias en la definición de los términos por lo que utilizaremos las siguientes definiciones [Grey93]:

1. **Atomicidad:** los cambios de estado de una transacción son atómicos: sin importar lo que suceda, se ejecutarán todos los pasos de la transacción o ninguno. Esto resulta en la propiedad de que ningún observador externo puede ver un resultado intermedio que desaparecerá si la transacción aborta.

2. **Consistencia:** una transacción es una transformación correcta de un estado. Las acciones tomadas como un grupo no deben violar ninguna de las restricciones de integridad. Esto requiere que una transacción sea un programa correcto. Una transacción no debería producir resultados incorrectos (o de lo contrario abortaría). Como un sistema de transacciones no tiene forma de verificar el cumplimiento o violación de todas las reglas de consistencia internas se puede asumir que la llamada al *commit* (hecha por el programador) garantiza la satisfacción de todas las reglas.

3. **Aislamiento:** aunque varias transacciones se ejecuten concurrentemente, el resultado es el mismo que si se ejecutan de a una, es decir que son *serializables*. Un conjunto de transacciones son serializables si los efectos de la ejecución concurrente de las sentencias son los mismos que los de una ejecución serial. Se pueden utilizar varios criterios para determinar si los efectos de dichas ejecuciones son equivalentes. El que más se menciona en la literatura es el de conflictos.

4. **Durabilidad:** Una vez que una transacción se completa con éxito (*commit*) los cambios realizados son permanentes, sobreviven para siempre y ante cualquier falla (o hasta que otra transacción los revierta o modifique).

Como se explica más adelante, las transacciones móviles son de larga duración, propensas a errores y heterogéneas. En consecuencia, imponer las propiedades ACID a las transacciones móviles resulta excesivamente restrictivo. Las transacciones ACID tienen un poder expresivo limitado y no permiten modelar estructuras de control complejas. Adicionalmente, las transacciones ACID no soportan commits, aborts o recuperaciones parciales. Finalmente, no hay manera de suspender una transacción para sobrevivir a una desconexión [Pit94].

Para comenzar a hablar de la relajación de las propiedades ACID en los sistemas transaccionales extendidos es necesario entender exactamente qué es lo que se relaja; además, para validar la satisfacción de una relajación es imperioso contar con una definición formal. Sin embargo, en cuanto a la propiedad de consistencia, en la literatura pueden encontrarse varias definiciones distintas y complementarias. Ante esta multiplicidad de definiciones, optamos por tomar la definición dada en el punto anterior. Sin embargo aprovecharemos esta sección para presentar otras posibles definiciones:

- En términos de base de datos el cumplimiento de la regla de consistencia significa que la base satisface todas sus restricciones de integridad o *constraints*, como por ejemplo: claves primarias, claves foráneas, y algunos predicados como *"la suma de gastos de cada departamento es menor e igual que el presupuesto del departamento."* [Ber97]

- La propiedad de consistencia es responsabilidad compartida entre los programas y el *TP System* que ejecuta esos programas. El *TP System* asegura que un conjunto de transacciones es Atómica, Aislable y Durable, siendo responsabilidad del programador asegurar que un programa transaccional preserva consistencia. [Ber97]

- Una transacción es una transformación correcta del estado de la base de datos. Las acciones tomadas como unidad no violan ninguna de las restricciones de integridad asociadas al estado. Esto requiere que la transacción sea un programa correcto. [Grey93]

- Una base de datos es consistente si sus datos obedecen las reglas de consistencia. Las reglas de consistencia de una transacción pueden ser implementadas como un programa de control que rechaza el *commit* de una transacción cuando no obedece dichas reglas. [Fra99]

1.4 Características de un entorno móvil

De aquí en adelante cuando nos referimos a un entorno móvil de comunicaciones estaremos nombrando a un conjunto de componentes conectados entre sí utilizando diferentes tecnologías, mencionando qué parte de la red estará intercomunicada con conexiones inalámbricas y asumiendo que algunos de dichos componentes se moverán geográficamente.

Para introducir las características de un entorno móvil, haremos una comparación con el tradicional entorno distribuido. Si bien la topología de un entorno distribuido y uno móvil son similares, su comportamiento dista de serlo. Como se menciona en [Lee97], [Bar99] y [Cor2000] las principales diferencias son:

Asimetría en el ancho de Banda: La comunicación desde el servidor al cliente normalmente posee un mayor ancho de banda. Incluso en algunos escenarios los clientes ni siquiera pueden enviar mensajes. Según valores de hoy en día el trafico desde los componentes móviles suele estar en tasas de 10 a 20 *Kbits/seg*, mientras que las redes locales inalámbricas manejan hasta 10 *Mb/seg*. Al tener el cliente móvil menor poder de transmisión**,** las transacciones provenientes desde las estaciones base suelen ser del tipo *broadcast* (modelo *push based*). Según [Pit95] las diferencias en el ancho de banda entre un *MH* y un *MSS* pueden alcanzar cuatro órdenes de magnitud.

Menor cantidad de recursos: En general en un modelo distribuido el cliente cuenta con menor cantidad de recursos que el servidor. En un entorno móvil se agrega un nivel más de diferencia ya que los componentes móviles suelen tener menor disponibilidad de recursos. Entre estos recursos, cabe mencionar las fuentes de alimentación que pueden provocar desconexiones voluntarias de parte del

componente y/o del usuario, como política de ahorro de energía. Los dispositivos móviles tienden a hacerse cada vez más pequeños por lo que el tamaño de sus baterías (en consecuencia su capacidad operativa) va disminuyendo. Del mismo modo, la capacidad de memoria, la velocidad de procesador y las facilidades de comunicación con el usuario (tanto de entrada como de salida de datos), están generalmente limitadas en los dispositivos móviles.

Mayores riesgos físicos: Un componente móvil puede sufrir mas inconvenientes que afecten su normal funcionamiento. Entre estos casos se encuadran los robos, destrucciones o pérdidas debido al tamaño y al movimiento de dichas unidades.

Frecuencia de desconexión: Es esperable que una máquina móvil sufra mayor cantidad de desconexiones durante su movimiento debido a factores externos (o a desconexiones voluntarias) que uno completamente distribuido. Esta salvedad será fundamental a la hora de simular modelos de transacciones. Además en [Bar99] se describen diferentes escenarios por los que un móvil podría comenzar a hacer *roamming*, por ejemplo al desconectarse de una celda y conectarse a otra. En [Pit95] se define un nuevo modo de trabajo llamado modo de operación de desconexión.

Movimiento: Por definición, una unidad móvil se desplaza. Esto implica que debe mantener su comunicación por radio y, dadas las limitaciones de alcance y ancho de banda que impone este medio, necesita encontrarse cerca de una estación base que le sirva de enlace con los servidores de la red fija. Cada una de estas estaciones base define una celda que atiende a un número limitado de dispositivos móviles. Cuando uno de estos dispositivos se desplaza de una celda a otra, el control del dispositivo pasa a la nueva estación base, junto con cualquier información necesaria para continuar el servicio sin que el usuario note el cambio.

Duración de las transacciones: La mayoría de las transacciones móviles son de larga duración por las limitaciones de recursos de los componentes móviles y debido a las desconexiones [San2001], [Lee97], [Bar99], [Sat96], [Maz99], [Pit98].

Tasa de Errores: Según [Pit95] y [For94] la tasa de errores en un ambiente móvil es mucho mayor que el de una red conectada, debido en parte al movimiento y a las características de la red inalámbrica subyacente.

Distinta naturaleza de los bloqueos: Algunas transacciones móviles deben dividirse lógicamente en subtransacciones, algunas de las cuales se pueden ejecutar en los *MSS* y otras en los *MHs* debiendo resolverse los conflictos de acceso además de bloqueos globales. [San2001]. Esto tema es atacado relajando algunas propiedades en los modelos transacciones avanzados

Presentamos a continuación un ejemplo de un entorno móvil. Una computadora personal que viaja en un auto, conectada mediante un celular y ejecutando una transacción puede pensarse como un sistema móvil, siempre y cuando el sistema de procesamiento de transacciones tome en cuenta la ubicación del móvil o el hecho de que se está moviendo. Si el sistema conoce la movilidad del cliente puede estar mejor preparado para lidiar con eventos como desconexiones temporarias (voluntarias e involuntarias), debilidad de la señal, etc.

El movimiento del cliente también puede determinar la invalidez de resultados obtenidos en otra localidad. Por ejemplo si una ambulancia en movimiento necesita saber el centro de atención médica más cercano, es deseable que los datos sean refrescados con respecto a la nueva ubicación del móvil.

En el capítulo de modelos móviles presentamos varios ejemplos de aplicación que se desarrollan en un ambiente móvil.

A continuación haremos una descripción general de los componentes que conforman un ambiente de aplicación móvil. Según [Bar99] un ambiente de comunicaciones móvil consiste básicamente en una red fija compuesta a su vez por máquinas comunes (sin acceso a los componentes móviles de la red), estaciones base a cargo de las comunicaciones (*MSS*s) y las componentes móviles (*MH*s).

Cada *MSS* sirve de nexo entre la red fija y las componentes móviles y cubre una zona geográfica, denominada celda o célula. En un determinado momento, un *MH* se encuentra en una celda controlada por un único *MSS*.

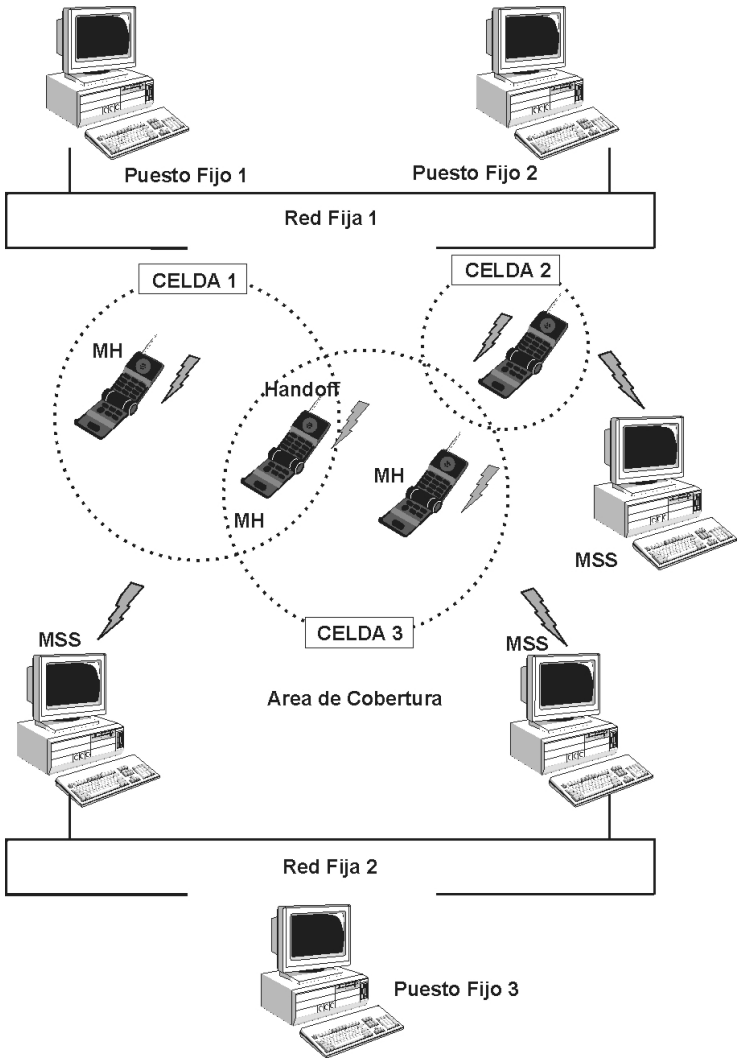


Figura 1: Escenario de un ambiente móvil

Mobile IP

En la actualidad el protocolo de redes por excelencia es *TCP/IP*. Con todas sus limitaciones, esta solución es ampliamente utilizada en ambientes de redes fijas. Desde fines de los 70s se han propuesto extensiones al protocolo para soportar movilidad bajo los conceptos de "computadoras en movimiento" (*mobile computing*). Para brindar computación móvil en una red *TCP/IP* se pueden utilizar dos aproximaciones. La primera consiste en ampliar los servicios básicos de *TCP/IP* con funcionalidad orientada a seguir el movimiento del móvil y ofrecer información de posición. La segunda consiste en usar redes móviles *A.T.M.* (*Asynchronous Transfer Mode*) que en la actualidad son utilizadas principalmente para transporte de datos multimedia. [Var98].

Mobile IP corresponde al primer tipo de solución. Una de las ventajas de Mobile IP es la compatibilidad que tiene este protocolo con *TCP/IP*, lo que permite la interoperatividad de redes. El mecanismo consiste en el reenvío de los paquetes enviados a la dirección IP de un agente (llamado *home agent*) al componente móvil cuya dirección actual sólo conoce el agente. Algunas de las ventajas y desventajas son analizadas en [Pe99],

entre las que se mencionan el problema del ruteo del triángulo y la reasignación dinámica del *MSS* encargado de enviar los paquetes a través de la red inalámbrica.

1.5 Desarrollo y estructura de la tesis

Durante el desarrollo de esta tesis, se llevaron a cabo las siguientes tareas:

1. El primer paso de la tesis consistió en investigar sobre la existencia de modelos transaccionales avanzados. Para ello recopilamos en la literatura extensiones y / o adaptaciones de modelos conocidos.
2. Luego se investigó la existencia de modelos que se adaptaran a los problemas característicos de un entorno móvil, siguiendo la misma metodología aplicada al paso anterior.
3. A continuación se investigaron las técnicas de análisis de rendimiento y se seleccionó la técnica de simulación de eventos discretos. Se relevaron los simuladores disponibles y se decidió desarrollar un simulador específico.
4. El siguiente paso consistió en diseñar y desarrollar un simulador de modelos transaccionales móviles que permitiera efectuar comparaciones cuantitativas.
5. La última etapa consistió en utilizar el simulador para efectuar el estudio comparativo propuesto y presentar las conclusiones resultantes de dicho estudio.

Este documento consta básicamente de dos partes: la primera, dedicada al estudio del modelos y el resto dedicado a efectuar el análisis comparativo de los mismos mediante el desarrollo de un ambiente que nos permite representar estos modelos y otros que se puedan proponer, hacer mediciones y compararlos gráficamente.

- Capítulo 1: Introducción
  Se describen los conceptos fundamentales para entender el marco teórico de base de datos y de ambientes móviles utilizado en este trabajo.
- Capítulo 2: Modelos Transaccionales
  Se presentan las características del modelo transaccional tradicional y los modelos transaccionales avanzados.
- Capítulo 3: Modelos Móviles
  Se presentan varios modelos móviles, preparados para explotar las ventajas y lidiar con los problemas propios de los ambientes móviles.
- Capítulo 4: Técnicas de Análisis de Rendimiento
  Se explican las técnicas estudiadas para la construcción del simulador.
- Capítulo 5: Diseño y Construcción del Simulador
  Se describe la solución propuesta para implementar el simulador y se muestran aspectos singulares concernientes a la implementación de los modelos móviles.
- Capítulo 6: Corridas y Resultados
  Se describen los resultados logrados mediante las corridas.
- Capítulo 7: Conclusiones y Trabajo Futuro
  Se presentan las conclusiones de la tesis y los trabajos futuros.
- Bibliografía Comentada
  Se indican y comentan brevemente las fuentes consultadas para escribir esta tesis.
- Apéndices
  Se presentan ciertos aspectos técnicos referentes en el desarrollo del trabajo.

En este documento se utilizaron las siguientes convenciones:

- Las siglas y los términos técnicos cuya definición puede encontrarse en el Glosario, se muestran en *itálica*.
- Las [Referencias bibliográficas] están encerradas entre corchetes.

Capítulo 2: Modelos Transaccionales

En el apartado <1.3 Transacciones y las propiedades ACID> ya describimos los conceptos fundamentales de las transacciones de bases de datos y las propiedades ACID. A continuación profundizamos estos conceptos y presentamos modelos transaccionales tradicionales y avanzados.

2.1 Sistemas de procesamiento de transacciones

Para darle soporte al concepto de transacción en los años 60s nacieron las aplicaciones de procesamiento de transacciones que consisten en una colección de programas diseñados para llevar a cabo las funciones necesarias para automatizar una actividad de un negocio en particular [Ber97]. Dichos sistemas de procesamiento de transacciones deben cumplir una serie de reglas mínimas: [Grey93]

Administradores de recursos: El sistema debe proveer un conjunto de manejadores de recursos transaccionales que provean operaciones *ACID* sobre los objetos que implementan. Los sistemas de bases de datos y lenguajes de programación persistentes son típicos ejemplos.

Estados persistentes: El estado de la aplicación se representa como información duradera almacenada por los administradores de recursos.

TRPC: Llamadas a procedimientos transaccionales remotos permiten a las aplicaciones invocar administradores de recursos locales o remotos como si fueran siempre locales. Esto permite diseñar aplicaciones divididas en clientes y procesos servidores que están en distintas computadoras, interconectadas por medio de *TRPC*.

Propiedades *ACID*: Un sistema de manejo de transacciones correctos debe respetar e imponer las propiedades *ACID*

Programas transaccionales: El programador representa cambios de estado y requerimientos de datos en forma de transacciones, es decir, rodeando ejecuciones exitosas del programa con un par Begin / Commit, y ejecuciones no exitosas con un par Begin / Rollback

2.1.1 Características de un sistema de procesamiento de transacciones

Desarrollo de aplicaciones: El sistema provee un conjunto de herramientas que facilitan el desarrollo de aplicaciones transaccionales: generadores de código, generadores de pantallas y formularios, optimizadores de SQL, depuradores, generadores de datos sintéticos, etc.

Repositorio: El repositorio administra la descripción de los objetos en el sistema y sus interdependencias (diccionario de datos) tanto para tablas internas como para los datos del dominio del problema. También administra los cambios a dicha información y asegura que coincida con la realidad.

TP monitor: Provee un entorno de ejecución para las aplicaciones y el manejo de recursos. Si llega un requerimiento, el monitor controla permisos, inicia una transacción si es que no está ya dentro de otra transacción y evalúa la prioridad. Según la prioridad el requerimiento puede ser postergado. Por último el monitor es el encargado de llamar al servidor.

Comunicación de datos: Los servicios de comunicación de datos varían según la arquitectura de la aplicación, que puede variar entre dos modelos extremos: el modelo de terminal “boba” y el modelo cliente-servidor. En el primero, toda la aplicación reside en el servidor y los puestos clientes son dispositivos orientados a formularios. Aquí, la comunicación se reduce a descripciones de objetos de interfaz (campos, botones, etc) y a registros de datos que son presentados al usuario o que éste ingresa en los formularios. En el segundo modelo, todos los servicios de presentación están en el puesto cliente, que tiene un control parcial de

la lógica de la aplicación. El servidor no se ocupa de cómo se obtiene o se muestra la información. La comunicación se produce por TRPCs del cliente al servidor, con datos referidos a consultas o modificaciones a los datos en la base.

Bases de datos: Almacenan y recuperan cantidades masivas de datos estructurados. Proveen un mecanismo para definir información estructurada y un lenguaje de consulta y modificación de la misma (generalmente SQL). Los detalles del almacenamiento no necesitan ser conocidos por el usuario: la base de datos maneja la correspondencia entre los registros u objetos abstractos del usuario y su disposición en el medio físico. Cada base de datos actúa como un manejador de recursos y de ese modo provee operaciones *ACID* sobre sus datos. También provee un conjunto de herramientas para visualizar los datos y generar reportes.

Operaciones: El sistema provee soporte para las tareas administrativas de mantenimiento, monitoreo del estado y de el rendimiento, reorganización de las estructuras, recuperación, copias de respaldo, etc.

2.1.2 Transacciones planas

Una transacción plana (Flat) es aquella que utiliza los conceptos del modelo transaccional tradicional. Para transacciones simples como el débito o crédito de una cuenta bancaria, una unidad atómica de trabajo es apropiada y éste es el tipo de aplicaciones para las cuales las transacciones planas fueron diseñadas.

Pero rápidamente fue extendido su alcance y generalizada su semántica. Esto es porque una estructura de control simple no es capaz de soportar modelos significativamente más complejos. Para soportar sistemas con servicios básicos, podría ser deseable proveer un modelo de transacciones con más poder expresivo.

Veamos un ejemplo donde las transacciones planas pueden no adecuarse: Un banco a fin de mes debe aplicar un interés a todas sus cuentas. Con transacciones planas este problema puede resolverse de dos formas: Se actualizan todas las cuentas dentro de una única transacción, o cada cuenta con una transacción separada. En la primera alternativa, si la actualización de alguna cuenta falla porque cualquier motivo, la transacción completa fallará y se deberá hacer un *rollback* de todas las cuentas que ya fueron correctamente actualizadas. En la segunda alternativa, ante una eventual falla del sistema, es posible de perder el control de qué cuentas fueron actualizadas y cuales no. Vemos entonces que las transacciones planas no son adecuadas para resolver este problema por si mismas.

Otro ejemplo en el cual las transacciones planas no se adecuan es la implementación de cualquier aplicación que requiera un backtracking, tal como la reserva de pasajes por tramos que mencionaremos en la siguiente sección.

2.2 Modelos transaccionales avanzados

Los sistemas transaccionales tradicionales son aquellos que cumplen y hacen cumplir las reglas *ACID* . [Grey93]. Dichas reglas fueron explicadas en la introducción de esta tesis.

Sin embargo, no todas las transacciones deben ser construidas como operaciones que respeten y refuercen las propiedades *ACID* en todos los niveles. Por el contrario, puede ser conveniente que el sistema alcance las propiedades *ACID* sólo cuando ellas sean requeridas, basándose en operaciones de bajo nivel que no tienen esas propiedades. Comprender estos modelos sirve además para entender los modelos móviles, que en general son adaptaciones o combinaciones de éstos. La mayoría de los modelos que presentamos aquí se encuentran definidos y explicados en [Grey93]. A modo de ejemplo supongamos el siguiente escenario:

Una persona desea hacer un viaje por avión desde el punto A al punto Z. Como no existe un vuelo directo entre ambos puntos, deberá trazar un recorrido compuesto por varios tramos. Si el sistema de reservas encuentra que hay dos rutas posibles A-B-C-D-Z y A-B-O-P-Q-Z, podría comenzar por reservar el primer tramo A-B de la primera ruta, más corta, para luego reservar el tramo B-C y luego el C-D. Si no encuentra lugar disponible en para el tramo C-D, un sistema transaccional que respeta las propiedades *ACID* hará *rollback* informando al usuario que debe buscar otra ruta, perdiendo así todas las reservas que ya había

obtenido. Un sistema avanzado podría descartar sólo las reservas para los tramos B-C y C-D y conservar la reserva sobre el tramo A-B para intentar con la segunda ruta.

2.2.1 Transacciones planas con savepoint

Este modelo es muy parecido al descrito anteriormente con el agregado de que permite introducir *savepoints* a los cuales podrá retornar la transacción siempre que lo desee mientras ésta no finalice.

Los *savepoint*s son marcas puestas explícitamente en la aplicación para señalizar el límite de un *rollback*. En la restauración desencadenada por un *rollback*, la transacción recupera el estado que tenía al momento del *savepoint*. Dado que los *savepoint* son numerados (el *begin work* es el primer *savepoint*), el *rollback* puede hacer referencia a cualquiera de ellos y deshacer más o menos cambios según sea necesario. Sin embargo, ante una caída del sistema, toda la transacción es abortada.

La reserva de pasajes por tramos es una típica aplicación de este mecanismo. Cada tramo reservado es marcado con un *savepoint*. Cuando una secuencia de tramos resulta inviable, el sistema puede intentar una ruta alternativa volviendo a un punto anterior de la transacción por medio de un *rollback* a un *savepoint*, sin perder por ello las reservas de los primeros tramos comunes a ambas rutas.

2.2.2 Transacciones encadenadas

Las transacciones encadenadas (chained transactions) son una variante de los *savepoint*. La diferencia radica en que en vez de un *savepoint* se usa un comando *chain*, que es una operación atómica semánticamente equivalente a un *commit* con liberación opcional de bloqueos y sin perder el contexto de ejecución, seguido de un nuevo *begin*. Esto permite asegurar la persistencia de los cambios realizados hasta el momento y continuar con la transacción, pero perdiendo la posibilidad de rollback más allá del último *chain*.

Ejemplo de aplicación:

- Proceso por lotes de movimientos contables.
- Cálculo y registro de atributos derivables en una gran cantidad de registros en forma *batch*, para reducir la sobrecarga en transacciones *on-line.*

2.2.3 Transacciones anidadas

Las transacciones anidadas (nested transactions) son una generalización del *savepoint*: mientras los savepoints permiten organizar una transacción en una secuencia de acciones que pueden ser revertidas individualmente, las transacciones anidadas puede organizarse jerárquicamente. Formalmente se define a una transacción anidada como un árbol jerárquico desencadenado por el nodo raíz y donde cada subtransacción (nodos intermedios y/o hijos) puede ejecutar un *commit* (cuyo resultado permanece aislado hasta el *commit* del ancestro inmediato y así transitivamente) o un *abort*.

- Una transacción anidada es un árbol de subtransacciones que pueden ser, a su vez, transacciones planas o anidadas.
- Las hojas son transacciones planas, y su distancia a la raíz puede variar para distintas partes del árbol.
- Una subtransacción puede hacer un *commit* o *rollback*. Si hace un *commit* sólo tiene efecto si su padre lo hace.
- El rollback de una subtransacción causa el *rollback* de todos sus hijos. Esto, en conjunto con el último punto, hace que las transacciones anidadas tengan las propiedades ACI pero no D: no respetan la durabilidad porque una transacción puede hacer un *commit* y éste no hacerse efectivo porque su padre hizo un *rollback*.

Las transacciones anidadas se rigen por las siguientes reglas:

- **Regla del *commit***: el *commit* de una subtransacción hace su resultado accesible sólo a su padre. El *commit* se cierra y termina cuando todos sus padres (hasta la raíz) hayan hecho su *commit*.
- **Regla del *rollback***: si alguna hace un *rollback*, todos sus hijos lo hacen también. Esto se aplica recursivamente hacia abajo.
- **Regla de visibilidad**: todos los cambios hechos por una subtransacción son visibles para sus hijos, pero únicamente son visibles para su padre luego del *commit*, y no para las hermanas.

Las transacciones anidadas proveen un mecanismo completo para sincronización del *rollback* en aplicaciones de estructuras complejas. Además es muy fácil combinar transacciones anidadas con programación modular utilizando una pila de llamadas. Por último, es fácil emular transacciones anidadas a partir de *savepoint*s por el simple mecanismo de asignar un *savepoint* al comienzo de cada subtransacción, aunque se pierden las propiedades de durabilidad de las transacciones anidadas.

Ejemplos de aplicación (se aplica a los ejemplos del modelo con *savepoint*s):
- Reserva de pasajes encadenados para llegar a un destino, ya explicado anteriormente.
- Transferencia entre cuentas, donde una subtransacción hace el débito y la otra el crédito.

2.2.4 Transacciones anidadas abiertas

Este modelo es la versión anárquica de las Transacciones anidadas: al no haber ningún tipo de restricción todas las subtransacciones pueden abortar o hacer *commit* independientemente de la transacción que la desencadenó. No existen restricciones en cuanto a relaciones entre padres e hijos. Las transacciones padre lanzan a otras transacciones sin tener control posterior de su destino.

Un ejemplo de la vida real es una búsqueda desordenada o alguna actualización periódica masiva de índices (que no sea critica si falla).

2.2.5 Transacciones distribuidas

Una transacción distribuida (distributed transaction) típicamente es una transacción plana corriendo en un ambiente distribuido. Por lo tanto, para poder resolverla tiene que visitar varios nodos dependiendo de donde se encuentre la información.

Por ejemplo, asumamos que la transacción *T* corre en un nodo *A*, el cual necesita dos tablas *X* e *Y*. Pero sólo *X* está disponible localmente, por lo que la tabla *Y* debe accederse a través de la red. Esto causa la división de *T* en *T1* y *T2*. Esta ejecución es similar a una transacción anidada, pero la descomposición no refleja una división jerárquica de la aplicación sino que se ve condicionada por la distribución de los datos en la red. Al igual que en el modelo anidado, las decisiones tomadas por cada subtransacción afectan a la transacción que las originan.

Como ejemplo de aplicación, podemos mencionar la reserva de artículos en un portal de compras donde los *stocks* están distribuidos en diferentes lugares y se quiere simular una centralización para simplificar el punto de acceso de los usuarios.

2.2.6 Transacciones multiniveles

Este modelo es una generalización del modelo anidado con la relajación de que el *commit* de una subtransacción implica que las demás subtransacciones del mismo nivel pueden ver los cambios inmediatamente. Para mitigar los efectos de resultados fantasma se requiere que cada subtransacción tenga

una contraparte compensatoria que pueda deshacer los cambios. Este modelo se encuentra definido y explicado en [Wei91] además de estar comentado en [Grey93].

Ejemplo de aplicación: actualización de un árbol de búsqueda balanceado (*B-Tree*). La inserción de una hoja provoca actualización de punteros en nodos del mismo nivel, que deben manejarse de manera transaccional para garantizar atomicidad de los cambios [Gray93].

2.2.7 SAGAS

Se define una Saga [Gar87] como una transacción de larga vida que puede ser dividida en varias subtransacciones secuenciales, que a su vez, pueden ser intercaladas con otras transacciones. Por cada subtransacción de una saga se define una subtransacción que la compensa, que es invocada si se decide el *abort* general de la saga y la subtransacción ya había hecho el *commit*. Cada subtransacción mantiene las propiedades *ACID* de la base de datos, no así el conjunto, que relaja atomicidad (pues se realizan *commits* parciales), aislamiento (debido a la intercalación y la visibilidad de resultados intermedios) y consistencia (porque cuando una saga aborta y es compensada, el resultado intermedio puede haber sido usado por otras transacciones).

Informalmente, una saga tiene las siguientes propiedades:

- Es un conjunto de transacciones planas S_1, S_2,... S_n. En el caso simple, éstas son ejecutadas secuencialmente. Por cada transacción S_i existe una transacción de compensación C_{Si}, que aplica los cambios necesarios a la base para revertir los efectos de su transacción correspondiente S_i.

- Si la ejecución no tiene problemas, la secuencia será: S_1, S_2,... S_{n-1}, S_n

- Pero en caso de existir un problema en la transacción S_j, la secuencia será: S_1, S_2,... S_j (*abort*), C_{Sj-1},.... C_{S1}

Como en las transacciones encadenadas sólo la transacción interrumpida puede hacer *rollback*, para todas las demás (las ejecutadas anteriormente) debe haber una compensación semántica porque las modificaciones han tenido su *commit*.

Por definición, SAGAS consiste en una secuencia de subtransacciones. En la ejecución de estas subtransacciones pueden aparecer intercaladas otras transacciones o componentes de otras transacciones SAGAS. Entonces SAGAS permite a otras transacciones acceder a los mismos objetos que alguna subtransacción que haya hecho *commit*, con lo cual la ejecución de las transacciones compensatorias puede no restaurar el estado original, a menos que se desencadene un *abort* en cascada.

En otras palabras, las subtransacciones en SAGAS son transacciones externas, donde todos los efectos hechos por las subtransacciones son visibles para todas las otras transacciones, y por lo tanto, cualquier resultado incompleto no puede ser ocultado.

Por lo tanto, SAGAS no soporta aislamiento. Además si se implementa sólo con *triggers*, no soporta la reanudación del cómputo o *rollback* luego de una caída de la base de datos. Es decir, si en el momento de una caída, una serie de subtransacciones compensatorias se estaban ejecutando, ningún *trigger* se ejecutará cuando la base de datos se reestablezca y la transacción quedara inconsistente.

Las sagas no son atómicas, porque el hecho de que se ejecute todo o nada incluye que ninguna otra transacción vea / utilice resultados que después serán vueltos atrás. Es cierto que esto está muy relacionado con el aislamiento, y esa es la razón por la que todos los modelos que vimos o bien relajan atomicidad y aislamiento a la vez, o bien no relajan ninguna de las dos propiedades.

Un ejemplo de aplicación puede ser un cliente que reserva un cierto número de asientos en un determinado vuelo: antes de reservar cada asiento, el sistema solicita información al usuario (fumador / no fumador, ventanilla / pasillo, comidas especiales, etc). Una vez reservado el primer asiento, no hay necesidad

de evitar que otro usuario pueda reservar otros asientos en el mismo vuelo. Si no quedan asientos libres suficientes o si el cliente cambia de opinión, los asientos ya reservados se liberan con transacciones compensatorias.

2.2.8 Transacciones split

El modelo de transacciones partidas (split transactions) se describe en el paper [Pu88] originalmente como una solución para grandes sistemas de CAD donde se relaja la propiedad de atomicidad al hacer *commit* de algunas partes (ya finalizadas) debido a la naturaleza larga (*LLT*) de la transacción que engloba el proyecto general (que en el caso presentado es el desarrollo de una gran componente de *CAD).*

El dominio de estas aplicaciones es abierto, lo que signi*fica* que puede durar mucho en el tiempo y sus *releases*[1] pueden ir sucediendo de a partes. Otra de las características del dominio *CAD* es la constante interacción entre las partes y/o diferentes personas (por ejemplo un gran proyecto de diseño).

La estrategia consiste en dividir la transacción principal en dos o mas transacciones serializables que pueden ejecutarse de manera secuencial o anidada hasta un nivel arbitrario. Cuando se realiza una operación de *split* los objetos tomados son repartidos entre las subtransacciones. Una subtransacción puede finalizar antes que la otra de manera independiente, aunque lo mas común es que se realice una operación de *join*. Si esto no sucede y una de las subtransacciones decide hacer un *commit*, sus resultados pueden ser vistos inmediatamente por el mundo exterior. El caso restante ocurre cuando se decide llamar a un *join_transaction*, momento en que ambas transacciones se ponen de acuerdo y ven si sus resultados son congruentes, en cuyo caso deciden hacer un *commit* conjunto.

2.2.9 Transacciones flexibles

Este modelo definido en [El90] fue creado principalmente para usar en un entorno *multidabase* caracterizado por tener los datos en motores de diferentes tipos, distribuidos en la red. En este caso el trabajo del *TP System* es mayor ya que debe coordinar sistemas autónomos e independientes.

El modelo se define como una extensión al modelo anidado separado en dos niveles. Como siempre, el primer nivel se utiliza para definir a la transacción global mientras que el segundo consiste en las subtransacciones distribuidas en los diferentes motores. Como cada subtransacción puede estar corriendo en un motor diferente, no hay forma de garantizar las propiedades *ACID* de manera global, lo que fuerza a utilizar un *commit* independiente para cada subtransacción. Esto último condiciona a la utilización de transacciones compensables. En caso de necesitar definir subtransacciones cuyos efectos sean permanentes, éstas deberán esperar un *commit* global antes de finalizar.

Ejemplo de aplicación:

- El dominio impone utilizar bases de datos heterogéneas o dispersas. Por ejemplo un sistema centralizado de bancos donde deban manejarse transacciones coordinadas interactuando con diversos tipos de motores de base de datos.
- Ídem con consultas web que recorran motores heterogéneos.

2.2.10 Multitransacciones

Este modelo es definido en [Buch92] como extensión al modelo anterior (Flexible) en donde los datos se encuentran almacenados en bases de datos diferentes y heterogéneas. La unidad de composición de transacciones es el bloque, que está compuesto a su vez por una o más subtransacciones.

[1] Entregas parciales de un producto.

Los subtipos de transacción (definidos por los bloques) pueden ser a su vez:

- **Multitransacciones**: corren en paralelo y sin orden (salvo que se establezca específicamente), posiblemente en bases de datos heterogéneas y que a su vez pueden tomar los tipos definidos mas abajo. Dichas transacciones pueden realizar *commit* de manera independiente según lo explicado en el modelo anterior.
- **Anidadas**: se rigen por las mismas reglas de visibilidad que el modelo anidado.
- **Compensatorias**: permiten liberar la responsabilidad de hacer un *commit* o no, relajando el asilamiento.
- **De contingencia**: estas transacciones se disparan cuando su transacción asociada falla antes de un *commit*. Su función consiste en reparar los resultados (como las compensatorias) o intentar una segunda alternativa.
- **Vitales**: son casos particulares de transacciones anidadas que no pueden abortar (en todo caso si lo hacen, obligan a toda la transacción a hacerlo).

Ejemplo de aplicación: un conjunto de transacciones globales que dependan de la composición de varios modelos y que deban reflejar las propiedades *ACID* de manera coordinada y distribuida.

2.2.11 Transacciones countermeasure

El modelo de transacciones countermeasure se define en [Fra98] y utiliza un enfoque diferente a los vistos anteriormente: se ve al iniciador de la transacción como cliente y al ejecutor (*TP System*) como servidor.

Una transacción global comienza del lado del cliente y tiene su contraparte dentro del servidor o servidores. La comunicación con el cliente se realiza a través de un único punto de entrada desde la transacción global aunque ésta tenga un árbol con subtransacciones. Cada subtransacción puede realizar un *commit* independiente en el motor correspondiente.

La clasificación que se les da a las subtransacciones es la siguiente:

- **Compensables**: son idénticas a los modelos anteriores.
- **Reintentables**: es una transacción que o bien realiza un *commit* exitoso o tiene otros medios para obtener el resultado (por ejemplo otras políticas de acceso como las de contingencia en el modelo anterior.)
- **Pívot**: las que no entran en las categorías anteriores (un resultado de esta clasificación es que sólo puede existir una transacción pívot dentro de la transacción global.)

La idea nueva de este modelo es que la clasificación descrita anteriormente determina un orden parcial de ejecución formado primeramente por las transacciones compensables (liberándola del *commit* local) y luego continua la transacción pívot (hasta aquí un *abort* general puede deshacer los resultados parciales). Una vez que la transacción pívot finalizó se da por concluida la transacción global disparando las transacciones reintentables.

La otra idea nueva que tiene el modelo es la de utilizar loqueos globales (que abarquen más de una base de datos) denominados *countermeasures*. Estos son controlados por el administrador global (o *Global Transaction Manager*) y evitan las anomalías de inserción y actualización como lecturas sucias y fantasmas.

Ejemplo de aplicación: simulación de propiedades *ACID* en ambientes de múltiples bases de datos con un motor centralizado que actúa como servidor y coordina diversos pedidos de parte de subsistemas clientes.

Capítulo 3: Modelos Móviles

Una vez descritos los modelos transaccionales usados habitualmente, presentamos los motivos por los cuales se desarrollaron modelos transaccionales específicos para ambientes móviles y describimos las diferentes propuestas existentes en la literatura que tienen en cuenta las características propias de los ambientes móviles en la ejecución de transacciones.

Muchos de los modelos descriptos a continuación están basados en modelos transaccionales avanzados, como Split y Nested (ver <2.2 Modelos transaccionales avanzados>), cuyas características resultan útiles para resolver ciertos aspectos de un entorno móvil. Por ejemplo, al producirse un salto de celda, algunos modelos móviles efectúan una operación *split* para continuar la transacción en curso en el nuevo *MSS*.

Al basarse en los modelos transaccionales avanzados, los modelos móviles también relajan las propiedades ACID. Esto es inevitable, ya que de cumplirlas en forma estricta, el porcentaje de transacciones abortadas debido a la naturaleza del entorno sería inaceptablemente alto.

Los modelos móviles presentan técnicas para tratar principalmente las desconexiones frecuentes y los saltos de celda, aunque ninguno de ellos presenta una solución que abarque ambos problemas [Cor2000]. Generalmente, se concentran en describir una solución a alguno de estos problemas en particular, quedando fuera del alcance cómo resuelven el resto de los problemas de un entorno móvil.

Cabe aclarar que estos modelos aún no han sido implementados en aplicaciones reales y, como uno de los objetivos de este trabajo es usar un simulador para compararlos en forma ecuánime, al implementar una simulación para cada uno de estos modelos completamos su especificación tratando de interpretar la intención primaria de los autores.

En este capítulo describiremos algunos de estos modelos y presentaremos nuestras propuestas a las subespecificaciones de los mismos.

3.1 El problema de las transacciones en ambientes móviles

A continuación analizamos los problema que causan las características de los ambientes móviles a los sistemas transaccionales. Estos problemas son los que motivaron el desarrollo de los modelos móviles que analizamos en este trabajo.

Asimetría en el ancho de banda: La mayoría de las aplicaciones transaccionales necesita transmitir tanto como recibir información. Cuanto mayores son los requerimientos de envío de información, mayor es la necesidad de ancho de banda de transmisión del *MH* hacia el *MSS*, y mayor es la necesidad de energía para realizar dicha transmisión. Como estos son dos recursos particularmente escasos en los dispositivos móviles, las aplicaciones que tienen este tipo de requerimiento son poco aptas para funcionar en entornos móviles. Los modelos móviles intentan resolver este problema de varias maneras: reduciendo en lo posible la necesidad de mantener permanentemente actualizada la base de datos central, o bien delegando en el MSS parte del trabajo, ya que el MSS no tiene problemas de energía ni de ancho de banda de comunicación con el servidor de bases de datos.

Menor cantidad de recursos: Las aplicaciones transaccionales varían en cuanto a sus requerimientos de procesador, memoria y capacidades de interacción con el usuario. Como los dispositivos móviles generalmente disponen de una capacidad reducida de estos recursos, muchas aplicaciones transaccionales no funcionan bien en este entorno. Al delegar una parte del trabajo en el MSS, algunos modelos móviles resuelven esta limitación.

Frecuencia de desconexión: La propensión de los dispositivos móviles a desconectarse, sea por decisión del usuario, por falla en las comunicaciones o por falta de batería, afecta el correcto funcionamiento de las aplicaciones transaccionales, ya que éstas requieren una interacción permanente con la base de datos. Si esta comunicación se interrumpe, la aplicación no puede seguir funcionando. En algunos casos incluso puede suceder que los recursos de la base que fueron reservados para la aplicación quedan en ese estado a la espera de ser liberados por una aplicación que ya no está operando. Los modelos móviles enfrentan este problema de dos formas: delegando en el MSS (que no se desconecta del DBS frecuentemente), u operando sobre una copia local de los datos en el dispositivo móvil. En estos casos siempre se asume que el dispositivo volverá a conectarse eventualmente.

Mayores riesgos físicos: Cuando un componente móvil se destruye o pierde, evidentemente no puede seguir operando. Este tipo de eventos es percibido por el servidor como una desconexión involuntaria. Sin embargo en este caso el dispositivo no volverá a conectarse y en consecuencia un modelo móvil poco puede hacer al respecto. Sólo se resuelve en los contados casos en que el modelo delega el trabajo en el MSS y este puede terminar sin necesidad de volver a interactuar con el usuario.

Movimiento: Cuando un dispositivo móvil se desplaza de una celda a otra, el control del dispositivo debe pasar a la nueva estación base, junto con toda la información necesaria para continuar el servicio. En el caso de una computadora móvil, a diferencia de un teléfono celular, cada paquete de información contiene la dirección del móvil en la red. En los protocolos de red usuales, esta dirección identifica al dispositivo, por lo que un cambio de dirección tendría el mismo efecto que una desconexión. Algunos modelos móviles resuelven este inconveniente trabajando explícitamente con las estaciones base, por ejemplo distribuyendo la carga de trabajo entre todas las estaciones base con las que toma contacto.

Adicionalmente, una aplicación transaccional puede depender de cierta información que sólo es válida en una determinada zona geográfica. Por ejemplo, el hospital más cercano en condiciones de atender a un paciente en una ambulancia. A medida que el móvil se desplaza, se invalidan las premisas en las que se basa la toma de decisiones realizada por la aplicación transaccional. Los modelos móviles que resuelven este problema utilizan la información geográfica explícitamente y la aprovechan para permitir aplicaciones transaccionales conscientes de su ubicación.

Duración de las transacciones: Como consecuencia de las frecuentes desconexiones, que pueden extenderse por períodos prolongados, y de la baja velocidad de transmisión de los dispositivos móviles, las transacciones suelen ser de mayor duración que las de un sistema tradicional que funciona en una estación fija. Esto produce un problema en aplicaciones con gran cantidad de usuarios, ya que los recursos reservados por las transacciones son retenidos por largos períodos de tiempo, causando bloqueos y, posiblemente, abortando las transacciones. Una estrategia para resolver este problema, utilizada por algunos modelos móviles, es la de trabajar en lo posible sobre copias locales de los datos, de modo que no es necesario reservar el recurso en el servidor central hasta el último momento.

3.2 Requerimientos de sistemas transaccionales para ambientes móviles

Como consecuencia de los problemas recién mencionados, para aplicar sistemas transaccionales en ambiente móviles es necesario tener en cuenta los siguientes requerimientos [Cor2000]:

- Habilidad para distribuir la carga del trabajo entre el *MH* y el *MSS*. Frecuentemente es necesario ejecutar ciertas partes de las transacciones en el *MSS* ya que cuenta con mayor poder de procesamiento, alimentación permanente, etc.
- Posibilidad de compartir información entre el *MH* y el *MSS*, ya que la transacción se divide en partes que se ejecutan en distintas plataformas.
- Capturar el movimiento de la transacción; como el *MH* donde se ejecuta la misma se mueve físicamente entre celdas, muchas veces es necesario transferir el control de la transacción entre *MSS*s.

- Posibilidad de manejar transacciones de larga duración, casuadas por desconexiones o por bajo ancho de banda.
- Soportar desconexiones largas; ya sean voluntarias por decisión del usuario o involuntarias por problemas físicos de la red, es importante que el usuario pueda seguir trabajando con el *MH*.
- Soportar caídas debidas a los problemas mencionados en Introducción a las características de un entorno móvil (ver <1.4 Características de un entorno móvil>).

3.3 Modelo Kangaroo

Las Transacciones Kangaroo [Du97] son una generalización de las transacciones multidatabase con el siguiente funcionamiento: cuando la transacción principal se inicia en un *MH,* es llamada Kangaroo Transaction (KT) y genera una subtransacción Joey Transaction (JT) en cada *MSS* que lo controla. A su vez estas últimas pueden desencadenar subtransacciones locales o globales. Durante el salto de celda, la transacción principal (KT) se "mueve" hacia el nuevo *MSS* donde se crea la JT local correspondiente; esto se realiza utilizando un split. La transacción JT que quedó en el *MSS* que controlaba la transacción global antes del salto de celda, efectúa un commit independientemente de la nueva JT. Sin embargo, en caso de una falla global de la KT todas las JTs precedentes son compensadas.

Esto deriva en dos modos de ejecución: el modo de compensación y el modo split. En el modo compensación, una falla en cualquier JT implica un abort en la KT que la controla, y por ende en todas las JTs precedentes. Por el contrario, en el modo Split la decisión de abortar o no las JTs que ya hicieron un commit la toma el motor de la base de datos involucrada.

En cuanto a su impacto sobre las propiedades ACID, ninguno de los dos modos garantiza seriabilidad de la transacción principal o KT, ya que el aislamiento puede ser violado al manejarse los bloqueos a nivel local. En el modo compensación, se asegura seriabilidad de las JTs ya que el inicio de la ejecución de una de ellas, depende del éxito (commit) de la anterior.

Si bien este modelo no está implementado, [Du97] presenta, como ejemplo de aplicación, el caso de un agente de seguros. Éste utiliza su dispositivo móvil para obtener la lista de casos a examinar en el día y elige uno para comenzar. Mientras se dirige al lugar, varias subtransacciones interrogan las bases de datos para obtener toda la información necesaria. Una vez en el lugar el agente de seguros ingresa la información relevante en el sistema, que actualiza las bases de datos centrales y genera los reportes necesarios.

Observamos que en [Du97] se afirma que nunca hay más de una subtransacción corriendo al mismo tiempo. Sin embargo, un párrafo anterior de la misma referencia afirma que cuando una subtransacción aborta, las otras que estaban corriendo no lo hacen. Entonces, no queda claro si puede haber más de una subtransacción corriendo al mismo tiempo. Para instanciar este modelo en el simulador que presentamos en el capítulo 5, permitimos configurar si será de una o de otra forma.

También [Du97] menciona que las *Joey Transactions* pueden ser compensables y que por lo tanto el usuario tiene que decidir si son compensables y en ese caso proveer información para que puedan ser compensadas. Sin embargo, las *JT* se crean por demanda en cada salto de celda. No parece razonable interrogar al usuario cada vez que cambia de celda para que provea información que permita compensar las *JT*

En [Du97] no se especifica el comportamiento ante las desconexiones, lo que también puede verse en [Cor2000]. Dado que las subtransacciones se ejecutan en el MSS, y la desconexión no afecta su comunicación con el servidor de base de datos, consideramos que las subtransacciones no se ven afectadas. Si el usuario desea interactuar con la aplicación para iniciar nuevas subtransacciones, deberá esperar a que se reestablezca la conexión.

3.4 Modelo Moflex

El modelo Moflex [Ku98] es una adaptación al modelo transaccional flexible (ver <2.2.9 Transacciones flexibles>) al que se le agregaron características propias de los ambientes móviles. La novedad de este modelo consiste en poder parametrizar condiciones que regulen el comportamiento de las transacciones.

Una transacción Moflex es un conjunto de subtransacciones ordenadas parcialmente según criterios de abort o commit (este ordenamiento genera un árbol de dependencias) y donde la ejecución de una subtransacción puede estar supeditada a condiciones 'ambientales' como la ubicación geográfica del *MH*, costos de comunicación, distancias, etc.

Durante el salto de celda, una transacción Moflex debe decidir qué hacer a partir de reglas de control determinadas por el usuario. El modelo soporta las siguientes reglas:

- ***Continue***: la subtransacción se sigue ejecutando en el nuevo *MSS* (al que se le transfiere todo el estado interno.)
- ***Restart***: la subtransacción se reinicia en el nuevo *MSS* con los mismos parámetros iniciales, mientras que todas las operaciones efectuadas anteriormente se abortan.
- ***Split_resume****:* la subtransacción ejecuta un *commit* en *MSS* previo y continua con la ejecución de la porción faltante en el nuevo *MSS.*
- ***Split_restart***: Se realiza un *commit* en la celda anterior al igual que *split_resume* pero la ejecución de la transacción se reinicia en el nuevo *MSS.*

En [Re99] se definen criterios de dependencia geográfica y se muestran técnicas de caché semántico.

Este modelo requiere la definición, por parte del usuario, de un conjunto numeroso de reglas. Entre ellas se pueden mencionar los predicados sobre control al momento del salto de celda, predicados sobre dependencia geográfica y temporal, etc. La especificación de estos predicados resulta impracticable para los usuarios, teniendo en cuenta que determinadas reglas dependen fuertemente del momento en que son ejecutadas, punto que frecuentemente no puede preverse.

Un ejemplo de la vida real pasible de ser implementado que [Ku98] presenta es una ambulancia que mientras traslada un paciente, se conecta con hospitales de la zona para conseguir una cama libre para su internación. Esta es la aplicación utilizada con el nombre de Sistema de Emergencias de Ambulancias (ver <6.3.1 Sistema de emergencias de ambulancias>).

En la literatura no aparecen referencias al impacto de este modelo en las propiedades ACID. Sin embargo, por las características del modelo, deducimos que Moflex relaja las propiedades ACI: si la transacción principal aborta luego de que algunas de sus subtransacciones finalizaron con éxito, dado que los resultados de estas últimas son visibles para otras transacciones, no se cumple la propiedad de aislamiento. Esto implica que tampoco cumple con la propiedad de consistencia, ya que los resultados parciales usados en otras transacciones pueden derivar en un estado inconsistente, ni con la propiedad de atomicidad, ya que el estado de la base luego del abort no necesariamente es el mismo que antes del inicio de la transacción.

Como se observa en [Cor2000], el modelo no habla acerca de cómo lidiar con desconexiones. De hecho, los algoritmos que detalla asumen que el *MSS* puede comunicarse con el *MH* siempre que lo desee, por ejemplo para comunicar el *abort* o el *commit* de la Transacción principal, o para pedir al usuario que defina si la suma de los resultados parciales de una subtransacción partida (por efecto de un salto de celda) es equivalente a la transacción no partida.

Decidimos completar la definición de Moflex de modo que una desconexión no resulte en el fracaso de la transacción principal, basándonos en que la actividad principal de este modelo ocurre entre el MSS y la base de datos y este tipo de comunicación no se ve afectado por las desconexiones. Además, el modelo presupone que la lista de subtransacciones y su secuencia de activación están predefinidas, por lo que el usuario no necesita intervenir para iniciar nuevas subtransacciones. Finalmente, consideramos que la probabilidad de que el modelo requiera del usuario que resuelva la equivalencia mencionada en el párrafo anterior, es baja porque esto sólo puede darse si la desconexión sucede entre el evento de salto de celda y la finalización de la subtransacción partida. En todo caso, la consulta al usuario es opcional.

3.5 Modelo Isolation-Only (IOT)

El modelo IOT propuesto en [Lu95] fue diseñado originalmente para extender el sistema de archivos de Unix en un ambiente móvil de manera transparente utilizando técnicas de caché. El núcleo del modelo consiste en permitir el trabajo en modo desconectado. Esta técnica se usa de manera combinada con la de replicación de datos. La clave del modelo radica en detectar y resolver los conflictos de lectura - escritura.

En su propuesta original en [Lu95] se describe este nuevo modelo como una extensión a un sistema de archivos distribuidos de Unix por lo que muchos de sus ejemplos se basan en manejo de archivos. Según [Lu95] los conflictos escritura - escritura son poco frecuentes (bajo la óptica de un modelo basado en desconexiones), por lo que se toma en cuenta principalmente los conflictos lectura - escritura.

El mecanismo de caché funciona de la siguiente manera: como la transacción se ejecuta en el *MH*, ésta utiliza los datos propios (aunque esté conectada). Este hecho también minimiza el tráfico. Las transacciones respetan la propiedades de aislamiento y seriabilidad de manera local por lo que los resultados no son visibles para el entorno. Si durante la fase de commit el *MH* está conectado con el servidor, la transacción (denominada de primera clase) publica sus cambios. Si no está conectado, la transacción (de segunda clase) publica los cambios a nivel local y queda pendiente de validación. Cuando el *MH* se reconecta, se validan los cambios para detectar conflictos de lectura - escritura. Si la validación falla, se resuelven los conflictos siguiendo una de varias estrategias: re-ejecutar la transacción, abortar la transacción, utilizar una subrutina de resolución de conflictos específicamente diseñada para la aplicación, o mostrar los conflictos para que sean resueltos manualmente.

En cuanto a las propiedades ACID, si bien las transacciones respetan ACID en forma local, el modelo relaja la atomicidad, la consistencia y el aislamiento, ya que, por ejemplo, los resultados de una transacción pendiente de validación que aborta al reconectarse pueden haber sido usados por otra que termina exitosamente. De esto se desprende que las propiedades ACID se preservan siempre y cuando el componente móvil permanezca conectado.

El ejemplo de la vida real que podría ser implementado presentado por [Lu95] es el de un desarrollador que se lleva trabajo a casa en una laptop haciendo una copia de las librerías que existen en el sistema central, y realiza una compilación local. Si en ese tiempo la librería en el sistema central fue cambiada, se genera un conflicto durante la reconexión.

IOT está diseñado para soportar desconexiones pero se desentiende de los saltos de celda. En forma análoga a la decisión tomada en WeakStrict, y para mantener la consistencia, completamos la especificación modelando el salto de celda como una desconexión. Como IOT mantiene una copia local actualizada de los datos, la transacción continúa funcionando en forma desconectada y luego deberá someterse a la sincronización al reconectarse con el servidor.

3.6 Modelo Weak/Strict

Este modelo fue definido inicialmente en [Pit95]. Al igual que el modelo anterior está pensado para soportar desconexiones y utiliza técnicas de caché. Se definen clusters de datos según algún criterio de agregación (que queda a cargo del programador). La configuración del criterio de agrupamiento por clusters es dinámica para aprovechar la ventajas de que un móvil puede saber cuando se desconectará (si esta desconexión es voluntaria) y la ubicación física de los datos.

El grado de consistencia (llamado consistencia-m) puede variar adaptándose al ancho de banda existente entre los clusters. Además como este grado queda a criterio de cada aplicación, éstas pueden separar los clusters para maximizar el ancho de banda disponible tomando en cuenta la predictibilidad de las desconexiones y conociendo la ubicación física de los datos.

Estos clusters mantienen las propiedades ACID entre sí pero no necesariamente entre otros clusters (se pueden definir niveles diferentes de consistencia inter-cluster). Para las operaciones intra-cluster se definen nuevas operaciones de lectura y escritura, denominadas débiles (weak read y weak write), en oposición a las operaciones estrictas (strict read y strict write), que son las habituales.

La operación de escritura débil guarda una réplica de los datos, de modo que otras operaciones de lectura débil puedan accederla. En cambio, las operaciones estrictas sólo leen datos escritos por operaciones estrictas, aunque las escrituras estrictas afectan ambos tipos de lectura. Al igual que el modelo anterior (IOT) todas las operaciones de validación de las propiedades ACID globales se realizan en tiempo de reconexión. De este modo, las transacciones en el *MH* pueden subclasificarse en:

- **Transacciones débiles**: aquellas que incluyen por lo menos alguna lectura o escritura débil.
- **Transacciones estrictas**: todas las que tienen escrituras o lecturas estrictas.

En este modelo no pueden definirse reintentos por lo que la única política posible ante un abort es compensar la transacción que hizo commit de manera local (por supuesto únicamente en las operaciones de escritura débil). Notemos además que las lecturas de valores no publicados globalmente pueden causar anomalías como dirty updates, fantasmas, etc.

Un caso especial de las transacciones débiles y estrictas son las consultas (u otras operaciones de lectura) cuyos conflictos suelen estar más acotados. El tratamiento es parecido al del modelo Transacciones de Pre-escritura (ver <3.8 Modelo Pre-escritura>).

Para garantizar la correctitud *intra-cluster* se definen dos criterios:
- **Correctitud Fuerte**: cada transacción débil observa el mismo estado de la base
- **Correctitud Débil**: las transacciones dentro de un mismo *cluster* ven valores similares, sin embargo otras transacciones también débiles pero de otros *clusters* pueden ver valores diferentes.

Además...
- Las transacciones estrictas mantienen consistencia *inter-cluster*, por lo que sólo son posibles estando conectado.
- Las transacciones débiles son obligatorias estando desconectado y opcionales estando conectado.
- Si el *MH* está desconectado y hace *commit* local de una transacción débil, luego cuando éste se conecta se hace el *commit* global, que puede resultar rechazado por el *host*.
- Si ocurre una transacción débil estando conectado, luego del *commit* local inmediatamente se hace el *commit* global (no tiene sentido demorarlo: no se gana ancho de banda y se arriesga el éxito de la transacción).

Un ejemplo de aplicación que podría ser implementado con este modelo, es la edición conjunta de un libro donde cada capítulo es compuesto por una persona que puede ver copias (no tan actualizadas) de las otras partes del libro. Esta es la aplicación utilizada con el nombre de Edición de un libro en colaboración (ver <6.3.3 Edición de un libro en colaboración>).

En [Pit95] no se detalla qué sucede con las transacciones de tipo *strict* ante las desconexiones, por lo que para completar la definición del modelo, decidimos abortar este tipo de transacciones.

Además, [Pit95] no hace referencia a cómo se resuelve el problema del salto de celda. El criterio que utilizamos para completar la especificación fue modelarlo como una desconexión ya que como las transacciones de tipo *weak* trabajan localmente, esto no necesariamente implica el fracaso de la transacción iniciada por el usuario. Sin embargo, como toda transacción de tipo *weak*, al finalizar deberá sincronizarse con el servidor, lo que podría resultar en un abort.

3.7 Modelo Reporting / Co-transactions

Este modelo se define en [Chr93] como extensión a las transacciones Open-Nested (ver <2.2.4 Transacciones anidadas abiertas>). El concepto central es muy sencillo: Ante un salto de celda se debe mover todo el estado interno de la transacción desde el *MSS* controlador hacia el nuevo. Para ésto se debe compartir el estado compuesto por los resultados parciales entre ambas estaciones (independientemente del estado de la transacción).

Formalmente se define una transacción S como un conjunto {t1, t2,..., tn} que a su vez pueden ser descompuestas en otras subtransacciones al igual que en los modelos Nested y Open - Nested. A cada subtransacción se la subclasifica en compensable y no compensable con la semántica habitual. A estos tipos de transacción se agregan las transacciones reportantes, que se caracterizan por relajar el aislamiento permitiendo que sus cambios parciales sean vistos desde el exterior. En caso de que una transacción reportante no sea compensable se dice que dicha transacción espera hasta el tiempo de commit para reportar sus cambios (de ahí el nombre).

El último tipo de transacción es el de Co-transacciones que son un caso particular de las transacciones reportantes que se ejecutan de modo concurrente de manera similar a la definida por el modelo Transacciones Multiniveles.

En este caso, como ejemplo de aplicación pasible de ser implementado, también se aplica el de un agente de seguros que se desplaza hacia sus clientes, ya mencionado en el Modelo Kangaroo (ver <3.3 Modelo Kangaroo>).

3.8 Modelo Pre-escritura

Este modelo se define en [San2001]. La idea novedosa consiste en agregar una nueva operación de aviso de escritura inmediatamente antes de la operación de escritura verdadera. De manera similar se hace un pre-commit antes del commit. Cuando se invoca a una operación de preescritura la transacción hace públicos sus cambios (relajando el aislamiento y la atomicidad) pero sin hacer la actualización de la base de datos. Una vez que se efectuaron todas la operaciones de preescritura se ingresa en la fase de pre-commit. El pre-commit ocurre en el *MH*, que puede estar desconectado. La segunda parte del commit se efectúa cuando se restablece la conexión (o si nunca ocurrió) en el *MSS*, con lo que se evitan los accesos a la base hasta el último instante (además el trafico de datos transcurre en la red fija). Como efecto colateral un valor pre-escrito por una transacción puede ser leído antes del commit, tanto en el *MH* como en el *MSS*. Como una transacción que entró en estado de pre-commit no puede abortar por la regla de *two phase commit*, el modelo no necesita transacciones compensatorias.

Un ejemplo de aplicación que podría ser implementado con este modelo, sería una empresa constructora que ofrece casas mostrando a sus clientes maquetas que pueden ir actualizándose a medida que la construcción avanza (por efecto de las pre-escrituras). Uno de sus clientes decide comprar una de las casas basada en el modelo de la maqueta. La construcción de alguna de las casas será dependiente de la maqueta elegida a la que se agregarán cambios personales. Lo importantes es que exista alguna casa que satisfaga los requerimientos de dicha maqueta que es ‘congelada’ (hace commit) desde el momento en que el cliente hace su opción final.

3.9 Modelo Pre-serialización

Este modelo se define en [Dir2001]. Las transacciones contenidas en el modelo se dividen en dos tipos. Las transacciones locales o “de sitio”, son aquellas cuyas subtransacciones controladas por el manejador de transacciones local, que acceden únicamente a datos locales. Las transacciones “migratorias” son transacciones que fueron locales pero, debido a un salto de celda, parte de estas comenzaron a ejecutarse en nuevos *MSS* (en este contexto el modelo es similar a Kangaroo)

Las transacciones globales del modelo utilizan un esquema multinivel compuesto por transacciones de sitio con la restricción adicional de que no puede existir más de una transacción de sitio por *MSS* (ésto es para

tener aislamiento por sitio). Todas las transacciones de sitio deben ser compensables por lo que pueden hacer commit de manera independiente evitando la duración prolongada de LLT y la consiguiente ocupación de recursos por un tiempo prolongado.

Las transacciones de sitio de subdividen en vitales y no vitales. Al igual que en los modelos precedentes la vitalidad de una transacción indica qué ocurre ante un *abort*: si la transacción es vital se debe abortar la transacción global. El asilamiento y la atomicidad globales están garantizados únicamente para las transacciones vitales.

El trabajo de administrar las transacciones recae en un coordinador global, que maneja una estructura de datos con todas las transacciones de sitio además de los estados de conexión de cada *MSS*. Existe un coordinador local por cada sitio. Las responsabilidades del coordinador global son:

- Iniciar al transacción
- Manejar la migración de datos entre *MSS*s
- Responder ante una desconexión de un sitio

Una transacción global puede estar activa, desconectada voluntariamente, suspendida por una desconexión involuntaria, en commit y abortada. Una transacción de sitio puede estar únicamente en los estados activa, ready to commit, abortada y en commit.

Los mensajes enviados a sitios desconectados son almacenados por el coordinador global y reenviados ante la reconexión. Para controlar la seriabilidad global el modelo ejecuta el algoritmo P.S.P.G. (Partial global serialization graph) descrito en [Dir2001] en cada sitio de manera distribuida. En esto se diferencia con respecto a modelos similares como Kangaroo.

3.10 Clasificación de los modelos móviles

En [Cor2000] se define un framework de comparación de modelos, que presenta varios criterios de clasificación. El primero distingue las siguientes categorías:

- **Sólo soporte de movilidad**: el modelo detecta el cambio de celda y define un comportamiento específico, sin embargo no responde ante una desconexión.
- **Sólo soporte de desconexión**: es el inverso al anterior. El modelo utiliza técnicas de replicación para trabajar de manera desconectada pero no detecta un cambio de celda.
- **Movilidad total**: en este modelo ideal, el *MH* puede trabajar en modo desconectado además de tratar adecuadamente el salto de celda.

Este criterio es usado para comparar, desde el plano teórico, los modelos Kangaroo, Moflex, IOT, WeakStrict y Reporting/Co-Transactions. Los resultados que se presentan en la Tabla 1 son en relación al soporte de movilidad y al soporte de desconexión.

Modelo Transaccional	Soporte de Movilidad	Soporte de Desconexión
Kangaroo	√	×
Moflex	√	×
IOT	×	√
Weak/Strict	×	√
Reporting / Co-Transactions	√	×

Tabla 1: Resultados presentados en [Cor2000].

Otro criterio de clasificación definido en [Cor2000] distingue entre los modelos adaptables y los no adaptables. Mientras que los primeros pueden predicar sobre condiciones de entorno (como Moflex) en los segundos no pueden referenciarse valores ajenos a la transacción.

El último criterio distingue el lugar físico de ejecución de la transacción que puede ser el *MH* el *MSS* o, en algunos modelos, ambos lugares.

Capítulo 4: Técnicas de Análisis de Rendimiento

4.1 Elección de la técnica

Cuando un nuevo modelo transaccional móvil es presentado, es habitual que su autor describa las ventajas del nuevo modelo con respecto a los modelos existentes en términos de un mayor soporte a la movilidad, mayor eficiencia en el uso de recursos o menor costo operativo. Como no existen datos reales que soporten dichas afirmaciones, el lector debe realizar una evaluación cualitativa y teórica.

Para poder realizar un análisis comparativo y cuantitativo de los distintos modelos transaccionales móviles es necesario analizar el rendimiento de dichos modelos. Existen distintas técnicas para realizar este tipo de evaluación, cada una de las cuales tiene ventajas y desventajas que deben ser sopesadas en función de las características particulares del sistema que se desea analizar y a los objetivos de dicho análisis.

Los modelos transaccionales móviles tienen ciertas características que deben ser tenidas en cuenta para seleccionar la técnica de análisis de rendimiento:

- Los modelos están presentados en forma de descripciones funcionales o algorítmicas.
- La interacción entre los modelos y los componentes de un entorno móvil constituyen un sistema complejo.
- No existen implementaciones de los modelos estudiados sobre un entorno móvil.

Por otra parte, también son de interés los objetivos que nos planteamos para este trabajo:

- Estudiar el comportamiento de los algoritmos de los modelos transaccionales móviles.
- Someter a dichos modelos a situaciones de características muy diferenciadas.
- Analizar la evolución y las tendencias de cada modelo a medida que variamos distintos parámetros del entorno en un amplio rango.
- Mantener la flexibilidad de poder cambiar los aspectos medidos, ya que muchas veces los resultados obtenidos sugieren nuevas líneas de investigación..

Con estos puntos en mente, se describen y evalúan a continuación las tres técnicas de análisis de rendimiento presentadas en [Raj91]: medición, modelado analítico y simulación.

4.1.1 Medición

La técnica de medición supone la existencia de un sistema igual o similar al propuesto que se encuentre implementado y en funcionamiento, ya sea el sistema final o un prototipo. A partir la medición de las variables de interés durante la experimentación con el sistema, se obtienen los datos necesarios para efectuar el análisis de rendimiento. Esta técnica puede lograr una gran exactitud en los datos, concediendo mayor confianza a los resultados y a las conclusiones derivadas de ellos.

Sin embargo, esta exactitud puede verse comprometida por la dificultad para medir los parámetros de interés debido a las complejas interacciones de un sistema real con su entorno y a que los datos obtenidos pueden reflejar las características particulares del experimento. Por ejemplo, no siempre es posible someter un prototipo a la misma carga que sufrirá el sistema real. La técnica de medición es, además, la menos apropiada en cuanto a flexibilidad para investigar distintas alternativas, tanto por el costo asociado a la reconfiguración de cada experimento, como por la dificultad para distinguir si los cambios en los resultados provienen del cambio de un parámetro o de un cambio fortuito en el entorno.

Por estas razones y por el simple hecho de que no hay implementaciones de los modelos estudiados, descartamos esta técnica.

4.1.2 Modelado analítico

El modelado analítico requiere abstraer el sistema analizado hasta el nivel de fórmulas matemáticas, evaluarlas bajo diferentes condiciones y aplicar técnicas estadísticas para derivar los resultados. Los modelos analíticos ofrecen la ventaja de examinar un amplio espectro de parámetros para determinar una configuración óptima. Sin embargo para sistemas complejos como el que nos ocupa, los modelos analíticos pueden resultar poco precisos, llevar a conclusiones erróneas o ser computacionalmente intratables.

Esta técnica presupone también que los aspectos que se quieren medir se conocen con exactitud desde el principio, lo que no es necesariamente cierto en nuestro caso. Cambiarlos una vez avanzado el trabajo puede invalidar el análisis previo, ya que las fórmulas se diseñan en función de aquello que se quiere medir. Además, los modelos están generalmente descritos en forma de algoritmos; la traducción de un algoritmo a una fórmula que lo describa con algún grado de exactitud puede resultar compleja y es difícil de verificar. Por todas estas razones, decidimos no utilizar esta técnica.

4.1.3 Simulación

La técnica de simulación implica programar un modelo simplificado del sistema que se desea evaluar, y luego utilizarlo como un prototipo del cual se obtienen los datos necesarios para realizar el análisis de rendimiento. Esta técnica permite evaluar un sistema evitando los costos de utilizar prueba y error con implementaciones reales y permite la evaluación de condiciones iniciales arbitrarias, al igual que la técnica de modelado analítico, pero con mayor exactitud en los resultados.

La exactitud de los resultados puede aumentarse con el agregado de un mayor nivel de detalle al modelo, si bien esto lleva tiempo y aumenta la complejidad del proyecto, con lo que se incrementa la probabilidad de errores que pueden invalidar los resultados y dificulta la tarea de descubrir y corregir dichos errores. Sin embargo, es la técnica adecuada para evaluar sistemas complejos que no cuentan con implementaciones reales y que deben someterse a una gran variedad de condiciones. Un simulador bien diseñado permite agregar y modificar variables a medir, incluso en etapas avanzadas del proyecto.

Por estas razones y porque la traducción de los algoritmos de los modelos transaccionales móviles a programas que los simulan resulta natural, la simulación es la técnica seleccionada para este trabajo.

Dentro de la técnica de simulación se distinguen distintos tipos: emulación, simulación de montecarlo, simulación dirigida por trazas y simulación de eventos continuos o discretos.

- **Emulación**: La emulación es una simulación al máximo nivel de detalle, que actúa igual que el sistema emulado. Es común utilizar dispositivos físicos similares al emulado, por ejemplo, al usar un microprocesador para emular a otro, o una terminal para emular a otra. La emulación suele ser sumamente fiel al original y permite interactuar con el sistema emulado como su fuese el real: emular el microcódigo de un microprocesador permite desarrollar y probar compiladores optimizadores para dicho procesador aún antes de su fabricación. Esta técnica requiere que el sistema a emular esté especificado completamente y en todos sus detalles. Sin embargo, los modelos móviles que queremos comparar no cuentan con una especificación detallada. Aún si estuvieran completos, realizar una emulación resulta poco práctico y no es necesario para nuestros objetivos.

- **Simulación de Montecarlo**: La simulación de montecarlo es utilizada para modelar sistemas estáticos, sin evolución en el tiempo, o para evaluar expresiones no probabilísticas usando métodos probabilísticos. Por ejemplo, para calcular la superficie de una figura puede aplicarse esta técnica de la siguiente manera: se generan puntos ubicados aleatoriamente dentro de un rectángulo que contiene a la figura y para cada punto se determina si está dentro o fuera de la figura. A partir de la superficie del rectángulo y la proporción de puntos que cayeron dentro de la figura, se deduce la superficie de

la figura. Dado que los algoritmos de los modelos móviles tienen fases y estados que evolucionan con el tiempo, esta tipo de simulación no resulta apropiado.

- **Simulación de eventos continuos**: Este tipo de simulación es apropiado para sistemas de naturaleza continua, es decir que el estado del sistema evoluciona de manera continua en función del tiempo. Por ejemplo, en la simulación de un sistema químico, el estado del sistema se describe en términos de concentraciones de sustancias, que están en constante fluctuación a causa de las reacciones químicas. Sin embargo, como se verá en el siguiente punto, los sistemas informáticos no son de naturaleza continua. Si bien el entorno móvil incluye aspectos físicos de naturaleza continua, como el movimiento de un *MH*, el objetivo de este trabajo es estudiar el comportamiento de los modelos móviles, un aspecto netamente informático y no continuo, por lo que no utilizamos este tipo de simulación.

- **Simulación de eventos discretos**: Esta técnica es apropiada para simular sistemas cuyo estado se describe en términos de valores discretos, que son afectados por la ocurrencia de eventos y permanecen inalterados hasta la ocurrencia del siguiente evento. Los eventos ocurren en forma aleatoria, siguiendo distribuciones predefinidas. Un simulador de eventos discretos se diferencia de un emulador en que el primero genera información que indica cómo habría actuado el sistema real, mientras que el segundo actúa efectivamente como el sistema real.

  Los sistemas informáticos suelen ser simulados con esta técnica. Por ejemplo, en la simulación del administrador de procesos de un sistema operativo, el estado del sistema está representado principalmente por la cantidad de procesos en las diversas colas de espera, estado que no se modifica hasta que algún proceso libera un recurso. Este es el tipo más adecuado de simulación para la comparación de modelos transaccionales móviles.

- **Simulación por trazas**: La técnica de simulación por trazas utiliza, como entrada al simulador, un registro de eventos de un sistema real en lugar de generar los eventos en forma aleatoria, para evaluar cómo el sistema simulado responde a dichos eventos. Si bien no disponemos de una traza obtenida de un sistema real, utilizamos esta técnica, como se verá en Conceptos básicos del simulador (ver <5.3 Conceptos básicos del simulador>), para someter a los distintos modelos móviles a la misma secuencia de eventos y, de esta manera, garantizar una comparación justa. En nuestro caso, la traza es obtenida de una simulación previa y no de un sistema real.

El simulador que utilizamos en este trabajo se basa principalmente en la técnica de simulación de eventos discretos. Claramente éste es el tipo de simulación más adecuado para nuestros propósitos, ya que se trata de un sistema informático cuyo estado se describe en términos de valores discretos, como la cantidad de dispositivos móviles o el estado de una transacción, y que evoluciona con la ocurrencia de eventos tales como un salto de celda o el fin de una transacción.

4.2 Aleatoriedad

4.2.1 Pruebas de aleatoriedad

Un punto central de cualquier simulación es la generación de números aleatorios, con los que es posible someter a los modelos a una gran variedad de condiciones iniciales y, además, descartar que los resultados reflejen dependencias ocultas o combinaciones particulares en las condiciones de entrada.

En informática es común el uso de número pseudo-aleatorios. Estos números, sin bien tienen muchas de las características de los verdaderos números aleatorios, son realmente generados de un modo determinístico y suelen tener propiedades que los hacen poco adecuados para los fines de un estudio como el que nos ocupa. Por ejemplo, toda secuencia pseudo-aleatoria se repite cíclicamente luego de una cantidad finita de números. Si la longitud del ciclo no es lo suficientemente larga, la calidad de los resultados puede verse afectada, por ejemplo mostrando dependencias donde no las hay.

Adicionalmente, muchos algoritmos comúnmente utilizados para generar números pseudo-aleatorios carecen de la calidad estadísticas necesaria. Se espera que un buen generador provea números uniformemente distribuidos e independientes entre sí. Sin embargo muchas, sino todas, de las secuencias pseudo-aleatorias contienen patrones que delatan su origen algorítmico. En general es difícil descubrir estos patrones y los mejores generadores pseudo-aleatorios pueden tener patrones no descubiertos aún. Para determinar la calidad de un generador existe una serie de casos de prueba que se aplican a las secuencias generadas y que buscan determinar el grado de uniformidad de su distribución y descubrir dependencias dentro de la secuencia.

Es por eso que, cuando la aleatoriedad de los valores a utilizar es importante para la calidad de los resultados (especialmente en simulación y criptografía), suelen usarse fuentes de números aleatorios verdaderos, por lo general mediciones de fenómenos físicos no determinísticos, como por ejemplo el resultado de arrojar un dado, el momento en el que decae un átomo radioactivo o simplemente ruido atmosférico digitalizado.

En este trabajo utilizamos una fuente de números aleatorios con distribución uniforme libremente disponible en Internet (www.random.org) que utiliza este último método: el sonido de una radio sintonizada en una frecuencia no utilizada por ninguna emisora es digitalizada a ocho bits, de los cuales se conserva sólo el menos significativo. La secuencia resultante se filtra utilizando el algoritmo “mapeo de transiciones” atribuido a *Von Neumann*, que toma pares de bits y sólo utiliza uno de ellos (por ejemplo el primero) si el par es “01” o “10”. De esta manera se obtiene una secuencia con alta entropía y con una distribución uniforme de ceros y unos.

Del sitio mencionado obtuvimos una secuencia de más de 400 millones de bits. Esta secuencia fue sometida a varias pruebas de aleatoriedad para asegurar su calidad. La literatura desaconseja utilizar una única prueba, ya que algunas pruebas pueden dar buenos resultados con secuencias no aleatorias (por ejemplo, la secuencia 0,1,2,... pasa la prueba de *chi-cuadrado* con puntaje perfecto). Las pruebas de aleatoriedad comúnmente utilizadas son:

- Prueba de Chi-cuadrado (la secuencia debería cubrir uniformemente n subintervalos)
- Prueba de los promedios (el promedio de los bits debería ser 0.5)
- Prueba de exactitud del valor de PI calculado por el método de *Monte Carlo*
- Coeficiente de autocorrelación (mide el grado de correlación entre valores sucesivos de una serie temporal; debería ser bajo)
- Prueba de la entropía (la capacidad de la secuencia para ser comprimida debería ser baja)
- Prueba de las corridas (la longitud de las subsecuencias monótonamente crecientes o decrecientes debería ser baja)

Para evaluar la aleatoriedad de los bits obtenidos de www.random.org realizamos las pruebas detalladas en la Tabla 2.

Prueba realizada	Resultado esperado	Resultado obtenido	Pasa la prueba?
Chi-cuadrado [2]	entre 10% y 90%	75%	Si
Entropía	1 bit por bit	1 bit por bit	Si
Promedio	0.5	0.5	Si
Valor de PI	3.14159265358979323...	3.140518461 (0.03% err)	Si
Autocorrelación	0	0.000209	Si

Tabla 2: Evaluación de números aleatorios

De los resultados de estas pruebas se desprende que no se puede rechazar la hipótesis de que los bits utilizados son aleatorios y provienen de una distribución uniforme.

4.2.2 Distribuciones de probabilidad

[2] En <Apéndice 4: Prueba de aleatoriedad> se muestra en detalle cómo se realizó esta prueba

Hasta aquí hemos mostrado cómo obtenemos números aleatorios con distribución uniforme. Sin embargo, la mayoría de las variables aleatorias que intervienen en una simulación tienen distribuciones más complejas. En la Tabla 3 mostramos algunas distribuciones que suelen utilizarse y qué clase de sucesos suelen representar. [Dev87]

Distribución	Descripción	Parámetros	Significado	Fórmula	Técnica utilizada
Bernoulli	Prueba simple con dos resultados exhaustivos y mutuamente excluyentes (éxito / fracaso)	P	P = probabilidad de ocurrencia	P	Método Discreto
Binomial	Cantidad de éxitos en una sucesión de pruebas de Bernoulli	N, P	N = cantidad de repeticiones, P = probabilidad constante	$(n\ y) * p^{y} * (1-p)^{n-y}$	Método Discreto
Geométrica	Repetición de pruebas de Bernoulli hasta el primer éxito	P	P = probabilidad de ocurrencia	$(1-p)^{n-1} * p$	Método Discreto
Poisson	Número de arribos en un intervalo de tiempo	λ	λ = frecuencia por unidad de tiempo / espacio	$(\lambda^{x} * e^{-\lambda}) / x!$	Método Discreto
Exponencial	Tiempo entre arribos	λ	λ = frecuencia por unidad de tiempo / espacio	$\lambda * e^{-\lambda t}$	Método de la transformada inversa
Gauss	Límite para la suma de eventos o distribución de errores de medición	μ, σ	μ = media esperada, σ = varianza	$(1 / \sqrt{(2 * \pi)})\ e\ ^{\wedge}\ (-(x-\mu)^{2} / 2\ \sigma^{2})$	Método basado en el Teorema del Límite Central
Uniforme	Sucesos equiprobables en un intervalo	a,b	a = límite inferior b = límite superior	1 / (b-a) si a ≤ x ≤ b 0 en otro caso	Método de la transformada inversa

Tabla 3: Distribuciones utilizadas en el simulador

4.2.3 Generación de números aleatorios con distribuciones no uniformes

A fin de obtener números aleatorios con las distribuciones descriptas en el apartado <1.4 Distribuciones utilizadas> se investigaron los métodos correspondientes a cada distribución en particular.

Partiendo de un generador de números aleatorios con distribución Uniforme(0, 1) independientes e idénticamente distribuidos, utilizaremos diferentes métodos para obtener muestras de otras variables aleatorias utilizando su función de probabilidad.

Una variable aleatoria *Uniforme*(0, 1) tiene la función de probabilidad dada por:

$$f_U(u) = 1 \text{ si } 0 \leq u < 1 \text{ y } 0 \text{ en otro caso}$$

y la función de probabilidad acumulada:

$$F_U(u) = u \text{ si } 0 \leq u < 1,\ 0 \text{ si } u < 0 \text{ y } 1 \text{ si } u > 1$$

Existen diversos métodos de generación presentes en la bibliografía [Dev87] [Bu2000]. Las características que determinarán la utilización de uno u otro para una distribución concreta serán:

- *Exactitud:* algunos métodos, como el basado en el teorema central del límite, son aproximados.
- *Eficiencia:* tanto en velocidad como en uso de memoria.
- *Robustez:* debe ser eficiente para cualquier conjunto de parámetros.

Debe necesitar siempre el mismo número de muestras de la Uniforme(0,1) para generar una muestra de la variable aleatoria *X*. De este modo nos independizaremos de la forma en que se generan / obtienen los números.

Métodos utilizados para la generación de distribuciones

Método de la Transformada Inversa

Este método utiliza la función de probabilidad de la variable de la que queremos generar una serie de muestras. Dada *U* una variable aleatoria uniforme en (0,1) y dada $F(x)$ la función de probabilidad acumulada de la variable aleatoria que queremos generar, las muestras vendrán dadas por la siguiente relación:

$$X = F^{-1}(U)$$

Demostración:

$$F(x) = P(X \leq x) = P[F^{-1}(U) \leq x] = P[U \leq F(x)] = F_U(F(x)) = F(x)$$

Simulación de una distribución Exponencial

Podemos utilizar este método para la generación de una variable aleatoria *X* cuya distribución es exponencial de media $1/\lambda$.

$$f(x) = \lambda * e^{-\lambda x}$$

y $F(x) = 1 - e^{-\lambda x}$ para $x \geq 0$.

Su transformada inversa es

$$U = 1 - e^{-\lambda x} \Rightarrow x = (-1 / \lambda) * \ln(1-U)^{3} = (-1 / \lambda) * \ln(U_2)$$

Simulación de una distribución Uniforme

En el caso de una distribución uniforme arbitraria $f(x) = 1 / (b-a)$ si $a \leq x \leq b$ y 0 en otro caso.

Tenemos $F(x) = (x-a) / (b-a) = U$ y su inversa despejando $x = a + (b-a) U$

Es importante resaltar que este método sólo es válido para la generación de variables aleatorias cuyas funciones de distribución $F(x)$ son invertibles y con función inversa única.

Método basado en el Teorema del Límite Central

Este método no es exacto, pero su sencillez lo hace apropiado, en determinados casos, para la obtención de secuencias de una variable aleatoria normal.

Simulación de una distribución Normal

La función correspondiente es $f(x) = (1 / \sqrt{(2 * \pi)})\ e\ ^\wedge\ (-(x-\mu)^2 / 2\ \sigma^2)$

Al no poder despejarse su función acumulada o poder utilizarse el método de transformada inversa hay que utilizar otras técnicas indirectas.

[3] Siendo U una variable aleatoria de distribución Uniforme $\Rightarrow$ 1 – U también lo es

El teorema del límite central demuestra que cualquier suma de *n* variables aleatorias independientes e idénticamente distribuidas (X_i) tiende a una variable aleatoria normal (Y) de parámetros: $\mu_y = \mu * n$ y $\sigma_y^2 = \sigma^2 * n$ siendo μ la media de las X_i, σ^2 su varianza y *n* un número entero suficientemente grande.

$Y = a_1x_1 + .. + a_nx_n \Rightarrow Z = (Y - \Sigma_{i=1..n}\, a_i\, \mu_i) / \sqrt{(\Sigma_{i=1..n}\, a_i^2\, \sigma_i^2)}$ tiene una distribución normal si $n \rightarrow \infty$

Si las variables que se están sumando son Uniforme(0,1) $\Rightarrow Z = \Sigma_{i=1..n}\, U_i - (n/2)) / \sqrt{(n/12)}$ tiene una distribución normal estándar.

Como la distribución normal estándar de una variable aleatoria x distribuida normalmente se obtiene como $Z = (x-\mu)/\sigma \Rightarrow$ para simular la variable aleatoria x basta con despejar

$x = m + s * (\Sigma_{i=1..\infty}\, U_i - (n/2)) / \sqrt{(n/12)}$

Se ha comprobado que para la generación de secuencias de una variable aleatoria $N(0, 1)$ es suficiente con utilizar 12 variables aleatorias *Uniforme*(0, 1) [Bu2000]. Por lo que si además descartamos los valores en el rango $(x < \mu - 6\sigma) U (\mu + 6\sigma < x)$ la expresión se simplifica a $x = \mu + \sigma * (\Sigma_{i=1..12}\, U_i - 6)$

Método de la Composición

Este método es útil para aquellas variables aleatorias que tienen una función de distribución $F(x)$ que no se puede expresar analíticamente, aunque sí como combinación de otras funciones de distribución conocidas $F_1, F_2, ... F_n$. No es necesario para generar ninguna de las distribuciones estudiadas por lo que no explicaremos este método.

Método de la Convolución

Este método se aplica cuando la variable aleatoria X a generar se puede descomponer como la suma de *n* variables aleatorias independientes e idénticamente distribuidas: $X = Y_1 + ... + Y_n$ Al igual que en el caso anterior no utilizaremos ninguna distribución obtenida con este método.

Método de Aceptación-Rechazo

Fue propuesto por *von Neumann* en el año 1951 y suele utilizarse cuando no es posible emplear ninguno de los anteriores. Es un método iterativo, por lo tanto aproximado, y es necesario especificar una función $t(x)$ que maximice la función de densidad de probabilidad $f(x)$ de la variable aleatoria de la que queremos las muestras.

Métodos discretos

Simulación de una distribución Bernoulli

Dada una probabilidad p con $0 \le p \le 1$ se elige un número aleatorio x de distribución Uniforme(0,1)
Si $x \le p$ entonces hubo éxito
Si $x > p$ entonces ocurrió un fracaso

Simulación de una distribución Binomial

Si deseamos generar números aleatorios con la distribución de probabilidad $(n\ y) * p^y * (1-p)^{n-y}$ para y = 0..n el resultado se obtiene repitiendo n veces la prueba de Bernoulli del siguiente modo

1. Se generan n números aleatorios con distribución Uniforme(0,1)
2. Se cuenta cuantos de estos son menores que la probabilidad p
3. La cantidad encontrada en el paso anterior es el valor simulado de la variable aleatoria

Simulación de una distribución Geométrica

Si deseamos generar números al azar con la distribución de probabilidad $(1-p)^{n-1} * p$ para y = 0..n el resultado se obtendrá repitiendo n veces la prueba de Bernoulli del siguiente modo

1. Se genera un número aleatorio con distribución Uniforme(0,1)
2. Si este número es menor que p se retorna la cantidad de pruebas realizadas
3. Se incrementa en uno la cantidad de pruebas realizadas y se retorna al paso 1

Simulación de una distribución Poisson

Se puede demostrar [Bu2000] que bajo las siguientes condiciones:

- ✓ El número total de eventos que ocurren durante un intervalo de tiempo dado es independiente del número de eventos que ya han ocurrido previamente al inicio del intervalo.
- ✓ La probabilidad de que un evento ocurra en el intervalo $[t,t + \Delta t]$ es aproximadamente $\lambda\Delta t$ para todos los valores de *t*.

Entonces se dan los siguientes resultados:

- ✓ La distribución de probabilidad del tiempo entre eventos es $f(t) = \lambda * e^{-\lambda\tau}$
- ✓ La probabilidad de que ocurran *x* eventos durante el tiempo *T* es $(\lambda^x * e^{-\lambda}) / x!$ (donde λ es el número de eventos promedio que ocurre durante el tiempo *T*).

Utilizando los resultados enunciados mas arriba se puede derivar el siguiente algoritmo [Bu2000]:

1. Definir el tiempo T
2. Simular mediante el método de la transformada inversa, números al azar que sigan una distribución exponencial con media $1/\lambda$
3. Sumar los tiempos entre eventos simulados en el paso anterior de modo que esta suma no sea mayor que T
4. Contar el número de eventos ocurrido durante el tiempo T

4.3 Análisis de resultados

Presentamos a continuación las técnicas utilizadas para obtener, a partir de un simulador de eventos discretos, resultados confiables que puedan ser usados para derivar conclusiones válidas y correctas. Estas técnicas son importantes especialmente en casos en los que no se cuenta con datos reales del sistema simulado. En estos casos es difícil asegurar la calidad de las simulaciones ya que cualquier resultado obtenido puede parecer correcto.

4.3.1 Validación y verificación

La validación implica responder a la pregunta "¿hemos construido el producto correcto?" En otras palabras, implica asegurar que el producto cumple con la especificación. La verificación, en cambio, implica responder a la pregunta "¿hemos construido el producto correctamente?", que equivale a asegurar que el producto cumple con el diseño.

En la mayoría de los proyectos de desarrollo de software, el usuario suele ser quien valida la correctitud del producto en la prueba de aceptación. En el caso de las simulaciones, la validación consiste en determinar si la simulación se asemeja al sistema real. Las técnicas presentadas por [Raj91] son apropiadas para validar simuladores:

Intuición experta: Esta es la técnica más efectiva y comúnmente usada para validar un modelo. Un experto puede validar un simulador por la simple observación de los resultados, o comparando los resultados de la simulación con mediciones de un sistema real.

Mediciones reales: Si se dispone de datos obtenidos de un sistema real, es posible validar el simulador por medio de la técnica estadística de intervalo de confianza.

Resultados teóricos: En algunos casos es posible construir un modelo teórico del sistema para comparar sus resultados con los obtenidos del simulador. El problema de esta técnica es que el modelo teórico también puede ser inválido, por lo que se recomienda validar como mínimo una parte del modelo teórico por medio de expertos o de datos reales, para luego proceder a comparar ambos modelos en situaciones más complejas.

La implementación de un simulador se puede verificar usando muchas de las técnicas comunes de aseguramiento y control de calidad de software. Las técnicas de verificación propuestas en [Raj91] son especialmente aplicables a simuladores de eventos discretos:

Diseño modular: El diseño debe estructurarse en módulos que se comunican entre sí por medio de interfaces bien definidas. Estos módulos pueden ser desarrollados, depurados y mantenidos en forma independiente, lo que facilita estas tareas y hace al sistema más confiable.

Verificación de invariantes: Las invariantes son condiciones que se comprueban durante la simulación y que verifican condiciones que deberían cumplirse si el simulador funciona correctamente. Por ejemplo, un simulador de tráfico de red puede comprobar al final de cada simulación si el número de paquetes enviados coincide con el número de paquetes recibidos más el número de paquetes perdidos. Cualquier diferencia es señal de un error de programación.

Inspección de código: Esta técnica consiste en explicar el código a otra persona o grupo de personas. Aún si estas personas no entienden el código, el simple acto de analizar el código línea por línea para explicarlo permite descubrir errores.

Modelos determinísticos: El principal problema para depurar un simulador es la aleatoriedad de sus variables. Evidentemente es más simple depurar un sistema determinístico. Una forma de reducir este problema es permitir la modificación de las distribuciones de las variables aleatorias y asignarles una distribución constante.

Casos simplificados: Debido a la gran cantidad de parámetros que suelen utilizar las simulaciones y a la dificultad inherente de analizar el resultado de una simulación para validarlo, resulta impráctico realizar pruebas funcionales sobre configuraciones complejas. Es conveniente configurar la simulación para correr casos simples, cuyos resultados se puedan analizar y comparar con los resultados esperados calculados en forma independiente.

Trazas: Una traza consiste en una lista ordenada por tiempo de los eventos ocurridos durante una simulación y sus variables asociadas. El usuario debe tener la opción de seleccionar el nivel de detalle de la traza o de filtrar cierto tipo de eventos o los eventos generados por cierta entidad de la simulación para fines de depuración.

Resultados gráficos: Dada la longitud de las trazas es difícil seguirlas para encontrar un error. La misma información puede mostrarse en forma gráfica, de manera que se puedan detectar fácilmente anomalías en los resultados. Una anomalía como la presencia de discontinuidades en las curvas puede indicar la posible presencia de un error.

Pruebas de consistencia: Estas pruebas consisten en verificar si el simulador produce resultados similares para casos distintos pero equivalentes. Por ejemplo, una red debería sufrir la misma carga con un nodos que genera 100 paquetes o con dos nodos que generan 50 paquetes cada uno. Cualquier diferencia en los resultados debería ser explicable o podría deberse a errores de programación.

4.3.2 Calentamiento (warm-up)

En la mayoría de las simulaciones sólo interesan los datos obtenidos cuando el sistema alcanzó un estado estable. En esos casos, los resultados de la parte inicial de la simulación, llamados “transitorios”, deben ser descartados. Por ejemplo, en una simulación de tráfico de red que comienza sin terminales conectadas y éstas se van agregando a medida que pasa el tiempo, deben descartarse las mediciones hasta que la cantidad de

terminales conectadas se estabiliza y por lo tanto el tráfico es representativo del estado típico de una red en plena operación.

Determinar exactamente qué porción de datos debe descartarse no es un problema simple. [Raj91] presenta diversas heurísticas:

- Realizar corridas largas, para minimizar la influencia de los datos transitorios. El problema de este método es que malgasta recursos y es difícil determinar si las corridas son suficientemente largas.
- Inicializar la simulación en un estado próximo al estable. En nuestro caso esto no es práctico por la gran cantidad de variables que definen el estado de una simulación y las complejas interacciones entre ellas.
- Truncar la primera porción de los datos hasta que el valor del primer dato no sea ni el mínimo ni el máximo de los datos restantes. Este método falla cuando los valores iniciales no crecen o decrecen en forma monótona, por lo que no es recomendable.
- Particionar los datos en porciones cada vez más grandes, calcular la varianza de los promedios de los datos de las porciones y determinar a partir de qué tamaño de porción la varianza comienza a decrecer en forma definitiva. Ése es el tamaño de la porción a eliminar.
- Estudiar el promedio de los datos luego de eliminar los resultados iniciales. Cuando se eliminan datos correspondientes al estado estable, el promedio de los restantes no debe cambiar significativamente. Por lo tanto, debe aumentarse la porción de datos eliminados hasta que el cambio relativo en el promedio de los datos restantes se estabilice. Sin embargo, por la aleatoriedad de los datos, siempre habrá una pequeña variación, por lo que el método requiere promediar varias simulaciones ejecutadas bajo las mismas condiciones de entrada.
- Aplicar el método anterior pero sin calcular el promedio total de los datos de la simulación sino únicamente el promedio de una ventana de datos móvil. Se descartan los datos iniciales hasta que el promedio de esta ventana se estabiliza.

Capítulo 5: Diseño y Construcción del Simulador

5.1 Elección del simulador

Una vez seleccionada la técnica de simulación de eventos discretos, se evaluaron dos alternativas: la adopción de un simulador libremente disponible, y la construcción de un simulador específico para la comparación de modelos transaccionales móviles. Para evaluar los simuladores disponibles, elaboramos la siguiente lista de requerimientos:

- Debe permitir la simulación de redes fijas e inalámbricas con componentes móviles
- Debe permitir la simulación de bases de datos y transacciones
- Debe permitir la programación de modelos transaccionales móviles
- Debe permitir la medición de variables arbitrarias

A continuación se detalla la lista de simuladores que fueron considerados:

Cnet: Este simulador de redes permite experimentar con varios protocolos de distintas capas del modelo OSI en redes formadas por combinaciones de vínculos punto a punto y segmentos de Ethernet. Cnet ha sido desarrollado específicamente para su uso en cursos de grado en redes de computadoras y ha sido usado en todo el mundo por miles de estudiantes desde 1991. (http://www.cs.uwa.edu.au/cnet/)

Berkeley NS: NS es un simulador de eventos discretos destinado a la investigación de redes. Provee soporte para simulación de protocolos TCP, ruteo y multicast sobre redes fijas e inalámbricas (pero no móviles). Según sus autores, NS no está completamente terminado y muchos errores pueden aún aparecer. (http://www.isi.edu/nsnam/ns/)

REAL Network Simulator: Es una versión anterior de NS, desarrollado en la Universidad Cornell, para estudiar el comportamiento dinámico de esquemas de control de flujo y congestionamiento en redes de conmutación de paquetes. Provee cerca de 30 módulos escritos en el lenguaje C que emulan las acciones de varios protocolos de control de flujo conocidos, por ejemplo TCP. (http://www.cs.cornell.edu/skeshav/real/overview.html)

OMNet++: Es un simulador de eventos discretos diseñado para simular redes de computadoras, multiprocesadores y otros sistemas distribuidos. (http://whale.hit.bme.hu/omnetpp/)

Parsec: Es un lenguaje de simulación basado en C, desarrollado por el Laboratorio de Computación Paralela de la UCLA, para la ejecución secuencial y paralela de modelos de simulación de eventos discretos. Es una versión modernizada de MAISIE. También puede ser usado como un lenguaje de programación paralela. (http://pcl.cs.ucla.edu/projects/parsec/)

Swarm: Simulador basado en agentes que interactúan en un ambiente dinámico. Provee un conjunto de bibliotecas disponibles en Java que implementan mensajes, objetos, grupos y *schedules*. (http://www.swarm.org/)

Estos simuladores se agrupan en dos categorías: los que están específicamente diseñados para la simulación de redes y los que proveen soporte para simulaciones generales. Descartamos los simuladores del primer grupo (Cnet, NS, REAL y OMNet++) porque se concentran en detalles de bajo nivel de la red, que no es un punto de interés en esta tesis, y ninguno soporta dispositivos móviles.

Descartamos también los simuladores del segundo grupo (Parsec y Swarm) porque, al tener un dominio muy amplio de aplicación, resultan demasiado generales y complejos para nuestras necesidades. Dado que estos simuladores no proveen soporte explícito para ninguno de los requerimientos planteados más arriba, dicho soporte debe ser programado especialmente. Esto implica que el uso de un simulador ya desarrollado no reduce significativamente el esfuerzo de programación para este proyecto. En otras palabras, la ventaja de

tener total control sobre el simulador supera el ahorro de usar un producto ya desarrollado para resolver un pequeño porcentaje del proyecto.

Por estos motivos, decidimos desarrollar un simulador de eventos discretos específico para resolver el problema de comparar modelos transaccionales móviles.

5.2 Alcances de la simulación

En este apartado describiremos las principales funcionalidades del simulador y aquellos aspectos que quedan fuera del alcance del mismo. Las características del simulador que permiten comparar modelos móviles son las siguientes

- El simulador modela los componentes de un entorno móvil: *MH*, *MSS* y *DBS*.
- También modela una red de comunicaciones fija que une los *MSS* con los *DBS* y los MSS entre sí, y una red de comunicaciones inalámbrica que une los *MH* y los *MSS*.
- Por esta red viajan paquetes que representan eventos, por ejemplo la orden de comenzar una transacción o el aviso de un salto de celda.
- Los vínculos inalámbricos pueden desconectarse y reconectarse.
- Los nodos de la red (*MH*, *MSS* o DBS) pueden iniciar transacciones, que tienen un comienzo, un fin (éxito o fracaso) y una duración, pueden subdividirse en subtransacciones y sufrir bloqueos.
- Cualquier nodo de la red puede contener una base de datos.
- La base de datos es capaz de administrar transacciones: controlar su estado, determinar su resultado y calcular la ocurrencia y duración de los bloqueos en función de la cantidad y duración de las transacciones activas simultáneamente.
- Los eventos modelados son: ingreso de un *MH*, destrucción de un *MH*, salto de celda, desconexión, inicio de transacción, fin de transacción, bloqueo de transacción, sincronización, transferencia de contexto.
- El simulador tiene un comportamiento predefinido ante cada evento.
- El usuario puede agregar nuevos modelos o modificar el comportamiento de los existentes representando una respuesta del modelo ante los eventos.
- La generación de eventos se efectúa siguiendo distribuciones de probabilidad configurables por el usuario.
- Los números aleatorios se pueden obtener con un generador pseudo-aleatorio interno o con un archivo externo de bits aleatorios reales.
- El usuario puede especificar la variación gradual de un parámetro para analizar tendencias.
- Se loguea información en un archivo plano para ser explotada por otras herramientas.
- El simulador incluye una herramienta de análisis y presentación gráfica de resultados que utiliza el archivo de log.
- El usuario puede comparar distintos modelos en una misma corrida bajo las mismas condiciones.

Parte de una simulación consiste en aislar ciertos aspectos de un problema asumiendo simplificaciones de la realidad. Acorde con esto y teniendo en cuenta aquellas partes de la realidad que nos interesa simular, es que decidimos asumir las siguientes simplificaciones:

- No modelamos datos concretos almacenados en tablas y registros individuales, ni operaciones concretas sobre los datos o sobre las estructuras. Modelar estos aspectos de la realidad implicaría una importante pérdida de generalidad y de facilidad de uso, ya que para comparar modelos, en lugar de modelar un número arbitrario de transacciones cuyas propiedades son determinadas por un número reducido de variables aleatorias, sería necesario "programar" cada transacción en forma individual y con características fijas.
- No verificamos el cumplimiento de las propiedades *ACID* de los modelos. Estas propiedades pueden ser evaluadas únicamente a partir de una traza de lecturas y escrituras en registros determinados, información de la cual carecemos por las razones detalladas en el punto anterior.

- Por la misma razón tampoco modelamos la administración de bloqueos llevando registro de cada ítem loqueado, ni la detección o los efectos de los abrazos mortales, aunque si modelamos muchos de los aspectos de los bloqueos (ver <5.5.2 Bloqueo de transacciones>).
- Consistentemente con los puntos anteriores, tampoco modelamos la interacción del usuario con las transacciones durante su ejecución: asumimos que el usuario inicia una transacción con toda la información que esta necesita.
- Debido a que carecen de interés para el tema que nos ocupa y a que están muy bien estudiados en la literatura, no modelamos los siguientes aspectos de bases de datos:
    - Autenticación y seguridad de comunicaciones.
    - Optimización de transacciones
    - Distribución de carga
    - Recuperación
    - Replicación de datos
- Los nodos de la red fija (*DBS* y *MSS*) no se caen ni se desconectan. Modelar estos aspectos de la realidad complicaría mucho las simulaciones y no aportaría significativamente al objetivo de este trabajo.
- Por la misma razón, consideramos que la red provee un servicio de transmisión de paquetes seguro, por lo que no modelamos la pérdida de paquetes ni errores de transmisión. Un paquete que debe ser enviado a un *MH* que se encuentra desconectado de la red quedará latente hasta que el *MH* se reconecte y eventualmente llegará a destino y sin errores.
- A pesar de que la red simulada provee un servicio de transmisión seguro y es capaz de *rutear* paquetes, no se modela en detalle una pila de protocolos como la definida en el modelo OSI [Ta95]
- Se considera que los *buffers* de mensajes en los nodos son infinitos, por lo que no se modela la sobrecarga de comunicación. Tanto este aspecto como los dos que siguen son tópicos de estudio en ambientes de telefonía móvil y decidimos dejarlos para trabajo futuro.
- Cada *MH* simulado dispone de su propio *MSS* o, lo que es lo equivalente a los efectos de este punto, los *MSSs* disponen de una cantidad infinita de canales, por lo que no se puede dar la situación de que una celda se encuentre saturada.
- No modelamos los aspectos geográficos de la red: las celdas no tienen ubicación, tamaño ni forma y los *MHs* no tienen posición ni vector de movimiento. Simplificamos todos estos factores en una distribución aleatoria que decide en qué momento un *MH* cambia de celda.
- Modelamos un único servidor de bases de datos. Si bien el simulador está preparado para representar más de un DBS, esto reduciría la probabilidad de bloqueo entre transacciones, por lo que, para someter a los modelos a un peor caso, decidimos correr todas las transacciones en el mismo DBS.
- Simplificamos la topología de la red asumiendo que todos los *MSS* tienen una conexión fija con el servidor de bases de datos (*DBS*) y otra conexión fija con el *MSS* que atendió al mismo *MH* antes del último salto de celda. Cada *MH* tiene una conexión inalámbrica con el último *MSS* de la cadena, que representa la celda actual.
- Dado que no modelamos los aspectos geográficos, los tiempos de propagación son independientes de las distancias entre los nodos de la red y están modelados con una distribución aleatoria.
- No consideramos paquetes de distintos tamaños, por lo que la única forma de modelar aplicaciones en las que el tamaño de los paquetes es un factor importante, es variar el tiempo de propagación.

5.3 Conceptos básicos del simulador

Evento: Los eventos del simulador son todos los sucesos ante los cuales el sistema (y en particular los modelos simulados) deben reaccionar. Cada modelo debe definir qué respuesta deberá dar ante la ocurrencia de cada evento. Como abstracción de la realidad sólo tomamos en cuenta una serie de eventos, que se detallan en la Tabla 4.

Scheduler: El *Scheduler* del sistema es el encargado de administrar la cola de eventos de la simulación, es decir registrar y activar en forma ordenada los eventos generados por el resto del sistema y enviarlos a sus destinatarios.

Evento	Significado
NetNewMH	Creación de un *MH* (equivale al encendido de un teléfono celular y su detección en una celda)
NetDestroyMH	Eliminación de un *MH* luego de terminadas sus transacciones (equivale al apagado de un teléfono celular)
NetHandoff	Cambio de celda de un *MH*
NetDisconnect	Interrupción temporaria de la comunicación radial entre un *MH* y su *MSS*
NetReconnect	Reconexión de una comunicación radial interrumpida
TxBegin	Comienzo de una transacción
TxCommit	Fin exitoso de una transacción
TxAbort	Fin no exitoso de una transacción
TxLock	Bloqueo de una transacción por parte de otra
TxSynch	Sincronización de cambios con una base global
TxTransfer	Transferencia del contexto de una transacción entre dos *MSSs*

Tabla 4: Eventos modelados en la Simulación

Tiempo: El tiempo en el simulador avanza en unidades de milisegundos, cada uno de los cuales es denominado un *tick*. Todo evento tiene asignado un *tick* de comienzo, que indica en qué milisegundo del tiempo simulado debe ser activado.

Al comienzo de la simulación el tiempo simulado es cero milisegundos. Cuando el *Scheduler* activa un evento, hace avanzar la hora simulada hasta el momento de activación del evento. Los ticks intermedios son salteados. Por lo tanto, la velocidad en tiempo real de la simulación es independiente de la escala de tiempo utilizada por los eventos. En cambio, depende de la cantidad de eventos que ocurren.

Plataforma: Definimos plataformas como las computadoras que intervienen en el manejo de las transacciones móviles. Una plataforma puede crear y atender eventos y contener una base de datos. Las plataformas implementadas en el simulador se detallan en la Tabla 5.

Plataforma	Significado
MH	Estación móvil capaz de iniciar transacciones.
MSS	Estación fija que sirve de enlace con el *MH* y que, según el modelo, puede tomar parte en la ejecución de las transacciones.
DBS	Estación fija que cumple la función de servidor de bases de datos.

Tabla 5: Tipos de plataforma

Red: La red simulada hace llegar los eventos desde la plataforma que lo lanza hasta la plataforma destino. Todo evento tiene un identificador del origen y del destino, información que la red utiliza para calcular la ruta que el evento debe seguir, en base a las conexiones disponibles. Cada paso en la ruta implica una nueva programación del evento en el *Scheduler*, ya que la transmisión de los eventos por los vínculos de la red lleva un tiempo finito.

Base de Datos: Una base de datos contiene y administra transacciones, decide el destino de una transacción o de una sincronización (éxito o fracaso) y calcula si deben producirse bloqueos entre las transacciones. Por defecto la plataforma *DBS* contiene una base de datos, pero otras plataformas pueden también tener sus bases de datos propias.

Política: Las políticas encapsulan el comportamiento determinado por un modelo. Una política determina qué hacer ante la ocurrencia de un determinado evento en una determinada plataforma. Por ejemplo la política de Kangaroo para el evento de salto de celda en la plataforma *MSS* determina que las transacciones que se encuentran corriendo en el *MSS* viejo deben ser transferidas al *MSS* nuevo. El sistema se encarga de invocar la política adecuada cada vez que un evento pasa por una plataforma, incluso si no se trata de la plataforma destinataria del evento.

Fábrica de eventos: Una fábrica de eventos es una entidad del simulador capaz de generar eventos nuevos. Las plataformas, las bases de datos y las políticas pueden ser fábricas de eventos. Para funcionar, una fábrica de eventos debe registrarse con el *Scheduler*, quien le requerirá a la fábrica de eventos que indique cuánto tiempo falta para que ocurra el siguiente evento y, cuando el momento llegue le requerirá que genere el evento.

Receptor de eventos: Un receptor de eventos es una entidad del simulador capaz de atender eventos y generar algún tipo de respuesta. Las plataformas, las bases de datos y las políticas pueden ser receptores de eventos.

Archivo de configuración: Archivo de texto en el que se definen los valores de los parámetros y las distribuciones de las variables aleatorias que serán usadas en la simulación. Es leído al principio de cada simulación y grabado al final de la misma para registrar los cambios que pudieran haber sufrido las variables.

Log: El log es también un archivo de texto que registra todos los eventos ocurridos durante una simulación. Es usado por el analizador para obtener los gráficos comparativos.

Simulación: Una simulación ejecuta un modelo bajo las condiciones definidas en el archivo de configuración. La simulación comienza con la creación de los primeros eventos por parte de las fábricas y termina cuando no quedan eventos por fabricar y todos los eventos fueron atendidos.

Iteraciones: Algunas de las variables definidas en el archivo de configuración puede indicar iteraciones, es decir que la simulación debe ser repetida variando únicamente el valor de las variables iteradas. Sólo puede iterarse una variable, a elección del usuario. Los valores que debe tomar la variable iterada en cada simulación se especifican dando el valor inicial, la cantidad de iteraciones y una fórmula que calcula el siguiente valor a partir del anterior, como se explica en la sección Formato del archivo de configuración (ver <1.7 Formato del archivo de configuración>).

Repeticiones: Dado que las simulaciones utilizan variables aleatorias, los resultados de un conjunto de iteraciones no necesariamente darán curvas suaves en los gráficos producidos. Por lo tanto, las simulaciones se pueden repetir y los resultados serán promediados. La cantidad de repeticiones que debe ejecutar el simulador está definida en el archivo de configuración.

Corridas: Una corrida es la ejecución del simulador con un archivo de configuración que define un conjunto de repeticiones, cada una de las cuales incluye un conjunto de iteraciones de una simulación. En otras palabras, cada corrida del simulador permite efectuar varias repeticiones de un conjunto de iteraciones sobre un modelo.

Reproducción: Para comparar los modelos es necesario asegurar que son sometidos a las mismas condiciones externas, es decir que las diferencias que aparecen en los resultados sólo puedan ser atribuidas a las diferencias entre los modelos y no al azar que pudiera haber favorecido a un modelo sobre otro.

Para lograr esto el simulador permite utilizar una corrida como modelo para otras corridas. Estas corridas utilizarán los mismos eventos de la corrida original, en el mismo orden y en el mismo momento del tiempo simulado, siempre que estos eventos no hayan sido generados por una política, ya que las políticas son justamente las que definen el comportamiento de cada modelo. En el archivo de configuración puede especificarse que una corrida sea reproducida utilizando varios modelos. Lo usual y aconsejable es utilizar el modelo Flat como corrida original.

Analizador: El analizador es un componente separado que toma como entrada el Log producido por un conjunto de corridas (reproducidas o no), almacena los eventos registrados en el Log en tablas de una base de datos, calcula estadísticas en base a estos datos y finalmente produce un conjunto de gráficos que muestran diversos aspectos del comportamiento de los modelos ante las situaciones definidas en el archivo de configuración.

Con estos conceptos aclarados, es posible describir muy brevemente el funcionamiento básico del simulador: Las plataformas fabrican y reciben eventos que viajan por la red a medida que son activados por el *scheduler*, el cual avanza el tiempo a medida que se suceden los eventos. Los eventos pueden tener asociadas transacciones, que son manejadas por bases de datos, que a su vez se alojan en plataformas. Los eventos pueden ser atendidos también por políticas, que son las que definen el comportamiento de los modelos transaccionales móviles. Toda esta actividad sucede en el marco de una simulación y en base a los parámetros del archivo de configuración. Cada simulación es ejecutada reiteradas veces con valores cambiantes de una variable iterada, lo que permite analizar las tendencias de los modelos, y todo el conjunto de iteraciones es repetido varias veces para promediar los resultados y evitar que los resultados se vean influenciados por valores anómalos de las variables aleatorias. Para comparar modelos, estas corridas son reproducidas para cada modelo repitiendo los mismos eventos con exactitud, lo que garantiza equidad en la comparación. El resultado de estas simulaciones es registrado en el archivo de log, que luego es analizado para obtener indicadores en forma gráfica, sobre los cuales se sacarán las conclusiones.

5.4 Arquitectura física

Dado que los modelos consisten más que nada en algoritmos, la implementación de cualquier modelo en el simulador involucra tareas de programación. Más allá de los modelos que efectivamente seleccionamos en este trabajo, resulta importante simplificar la implementación de cualquier modelo. Entonces, uno de los principales objetivos de diseño es separar el motor de simulación de los modelos simulados.

Para lograr este objetivo, el simulador está estructurado de tal manera que para implementar un modelo sólo es necesario definir cómo deberá responder ante los eventos de interés. El usuario hace esto definiendo las *políticas* del modelo. Cada política define cómo responde un modelo a un cierto evento. Por lo tanto hay políticas de respuesta para todos los eventos generados en el simulador, por ejemplo desconexiones, comienzo de transacción, etc. Para facilitar esta tarea proveemos un conjunto de políticas por defecto, que definen el comportamiento del modelo transaccional tradicional (*Flat*) y que el usuario puede redefinir para cambiar el comportamiento básico, y proporcionamos una interfaz (*API*) para simplificar el acceso a las funcionalidades principales del motor.

También separamos la simulación del análisis de los datos, debido a que son procesos claramente distintos, se realizan en distintos momentos, y las herramientas más apropiadas para uno no son las más apropiadas para el otro (por ejemplo una herramienta capaz de manipular datos con SQL no es adecuada para procesos de cálculo intensivo). Como ventaja adicional, el medio de comunicación entre ambos procesos (el archivo de log), permite además realizar un análisis manual detallado de lo ocurrido en una simulación. En la se pueden apreciar las relaciones entre los principales componentes.

Luego de leer el archivo de configuración, el controlador de la simulación inicializa el simulador y le indica a la cola de eventos que comience a procesarlos. A medida que los eventos se producen, son enviados a las plataformas correspondientes y a su vez a las políticas que definen el comportamiento del modelo. Estos pueden responder generando nuevos eventos, que serán agregados a la cola de eventos. cada evento es reportado al administrador de log, quien lo graba en el archivo de log. Una vez terminada la simulación, el analizador carga el archivo de log a una base de datos y, por medio de consultas SQL, genera los gráficos correspondientes.

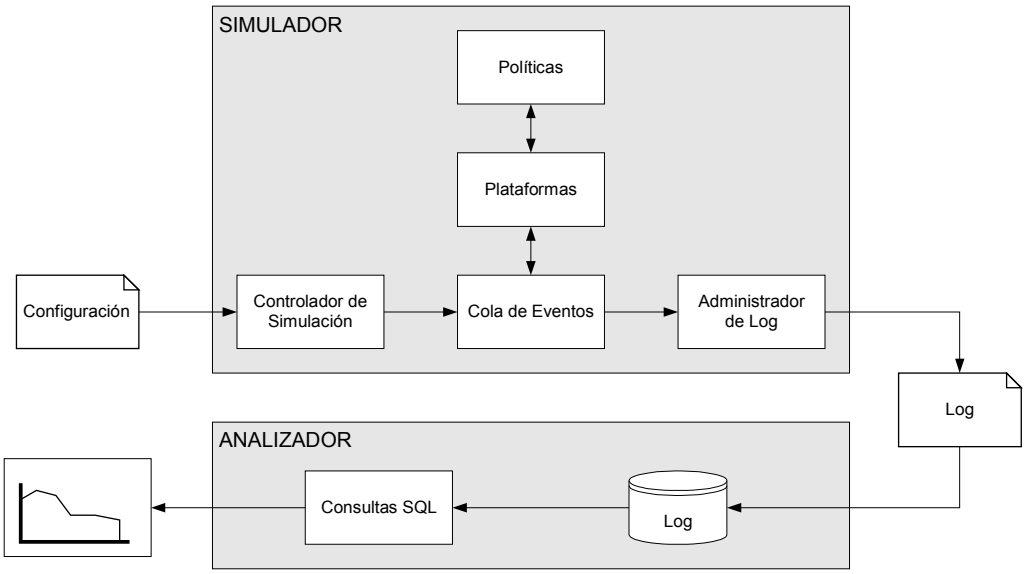


Figura 2: Arquitectura física del simulador

5.5 Aspectos específicos del diseño

5.5.1 Cálculo de la longitud de una transacción

La duración de una transacción se calcula en base a las variables aleatorias `DBMS_AccessesPerTx`, `DBMS_TimeBetweenAccesses` y `DBMS_AccessesProcessingTime`, que siguen distribuciones Gaussianas, y los tiempos de propagación de la red, que están determinados por las variables aleatorias `NetLinkDelay_Radio_MHtoMSS`, `NetLinkDelay_Radio_MSStoMH` y `NetLinkDelay_Wire`, que siguen distribuciones Exponenciales, según la siguiente fórmula:

```
Duración =  DBMS_AccessesPerTx * ( DBMS_TimeBetweenAccesses +
                                 + DBMS_AccessesProcessingTime +
                                 + 2 * NetLinkDelay_Wire +
                                 + NetLinkDelay_Radio_MSStoMH +
                                 + NetLinkDelay_Radio_MHtoMSS )
```

Esta fórmula se interpreta de la siguiente forma: la duración intrínseca de una transacción se calcula como el tiempo que lleva hacer un acceso a la base por la cantidad de accesos. La duración de un acceso a la base se calcula como el tiempo entre dos accesos sucesivos, mas el tiempo que lleva procesar un acceso en la base, más el tiempo que le lleva a un paquete ir desde el *MH* hasta el *DBS* y volver, que a su vez se calcula como el tiempo que lleva transmitir un paquete desde el *MH* hasta el *MSS*, más dos veces el tiempo de transmisión de la red fija (desde el *MSS* hasta el *DBS* y la vuelta), más el tiempo de transmisión desde el *MSS* hasta el *MH*.

Dado que algunos modelos dividen las transacciones en subtransacciones, cuando la transacción es agregada como subtransacción de otra, recalcula su longitud dividiéndola por el valor de la variable aleatoria `DBMS_MaxNumberOfSubTx`, que determina la cantidad de subtransacciones que tendrá la transacción principal. Con esto, la suma de las duraciones de las subtransacciones durará lo mismo que una transacción común de un modelo que no usa subtransacciones.

Cuando el evento de inicio de transacción (*TxBegin*) llega al *DBMS*, esta duración puede variar debido a un bloqueo: se alarga si se produce un bloqueo cuya duración es menor que el tiempo límite de bloqueo, o se acorta si el bloqueo supera el tiempo límite y la transacción aborta antes. Los detalles del cálculo de la duración de un bloqueo pueden verse en el siguiente punto.

5.5.2 Bloqueo de transacciones

Cuando el evento *TxBegin* es atendido por el *DBMS*, se cuenta la cantidad de transacciones activas y se compara con una variable aleatoria de distribución Geométrica que toma como parámetro el valor de `DBMS_txLockProbability`. Esta variable aleatoria representa la cantidad de transacciones necesarias para que se produzca un bloqueo. Si la cantidad de transacciones activas es superior a este número, se produce un bloqueo. Queda entonces calcular el punto de inicio y la duración del bloqueo.

La duración del bloqueo se calcula en base a la duración de la transacción y a la cantidad de transacciones activas, ponderados por dos factores:

```
Duración = duración de la transacción * DBMS_txLockLenProportion *
           (1 + #{transacciones activas} * DBMS_txLockLenPopulationFactor)
```

`DBMS_txLockLenProportion` define la contribución de la duración de la transacción a la duración del bloqueo: si vale 0.5, la duración del bloqueo será proporcional a mitad de la duración de la transacción; si vale 1, la duración del bloqueo será proporcional a la duración total de la transacción.
`DBMS_txLockLenPopulationFactor` determina cuánto influye la cantidad de transacciones activas en la longitud del bloqueo: si es cero, no influye y la duración del bloqueo depende únicamente de la longitud de la transacción; si es 0.5, cada transacción activa incrementa la duración del bloqueo en 50%.

El tiempo de inicio del bloqueo se calcula utilizando una distribución Uniforme que va desde 0 hasta una fracción de la duración de la transacción. Esta fracción se calcula en base a la cantidad de transacciones activas y al parámetro `DBMS_txLockTimePopulationFactor`, que determina cuánto influye la cantidad de transacciones activas en la elección del momento de inicio de un bloqueo. Si es cero, no influye y el bloqueo comienza en cualquier momento de la transacción. Si es N y hay N/2 transacciones activas, el bloqueo sucede en cualquier momento de la primera mitad de la transacción. Si es N y hay N o más transacciones activas, el bloqueo sucede inmediatamente comenzada la transacción.

Luego se calcula si la duración del bloqueo supera el tiempo máximo que una transacción puede estar bloqueada antes de que aborte. Este tiempo máximo está dado, en milisegundos, por el parámetro `DBMS_txLockTimeOut;` Si este valor es cero, las transacciones nunca abortan por bloqueo. Este comportamiento corresponde a uno de los mecanismos de detección de *deadlock* [Ber97], [MS2003].

De esto se desprende un comportamiento interesante: si hay muchas transacciones y el límite es bajo, las transacciones van a abortar frecuentemente antes de la duración prevista originalmente. En otras palabras, un límite bajo aliviana la carga del *DBMS*.

Finalmente se calcula la nueva duración de la transacción. Si hay bloqueo y su duración no supera el límite, se suma su duración a la de la transacción y se programa el evento de fin de la transacción (*TxAbort* o *TxCommit*) para ese momento. Si su duración supera el límite, la duración de la transacción se calcula para que termine luego de superado este límite tras el comienzo del bloqueo y se programa un evento *TxAbort*.

Si el bloqueo se produce, se programa el evento *TxLock* para que quede registrado y para que el usuario pueda definir, si el modelo que está implementando así lo requiere, una política de bloqueo.

Esta forma de calcular el bloqueo produce una anomalía: si se lanzan muchas transacciones en forma casi simultánea, el bloqueo calculado para las primeras será demasiado bajo, ya que al momento de lanzarlas la población de transacciones activas aún baja. Sólo las últimas encontrarán la población real de transacciones activas y calcularán su bloqueo correctamente.

En lugar de modificar la implementación para corregir esta anomalía, observamos que, si hay una cantidad más o menos sostenida de transacciones, a medida que las más antiguas van terminando llega un momento a partir del cual todas las transacciones activas tuvieron al comenzar más o menos la misma cantidad de vecinas; en ese punto los bloqueos estarían bien calculados para todas las transacciones. En otras palabras, el problema se resuelve con el warm-up del simulador descrito en la siguiente sección.

5.5.3 Análisis del log

El componente que analiza el log, implementado en Microsoft Access, realiza una serie de pasos para procesar el archivo generado por la simulación y obtener los gráficos que pueden verse en el capítulo de corridas (ver <Capítulo 6: Corridas y Resultados>):

- Lee el archivo de log y carga sus registros en diferentes tablas, según el tipo de registro. Sólo guarda en las tablas los registros de tipo 1 (evento), 2 (estadística) y 7 (variación).
- Calcula el período de calentamiento de la simulación a partir de las estadísticas de cantidad de *MH*s, usando el algoritmo descrito en el punto siguiente, y descarta todos los eventos de dicho período.
- Obtiene la lista de transacciones ejecutadas en la simulación y completa la siguiente información sobre cada una de ellas: si es una transacción principal o una subtransacción, su longitud y si terminó exitosamente o no.
- Para cada iteración de cada modelo, distinguiendo entre transacciones principales y subtransacciones y entre transacciones exitosas y no exitosas y promediando las distintas repeticiones, calcula las siguientes estadísticas: la duración de la simulación, la cantidad de transacciones, la duración de las transacciones, la cantidad de paquetes de radio y de red fija por transacción y la cantidad de saltos de celda y desconexiones por transacción.
- Con estos datos genera los siguientes gráficos:
    - Carga de trabajo asumida por el *MH* (porcentaje de paquetes de radio)
    - Robustez del modelo (porcentaje de transacciones exitosas)
    - Eficiencia desde el punto de vista del uso de recursos (paquetes enviados por transacción exitosa)
    - Eficiencia desde el punto de vista del usuario (transacciones exitosas por minuto)
    - Costo del fracaso desde el punto de vista del uso de recursos (paquetes invertidos en transacciones no exitosas)
    - Costo del fracaso desde el punto de vista del usuario (relación entre la duración de las transacciones exitosas y las no exitosas)

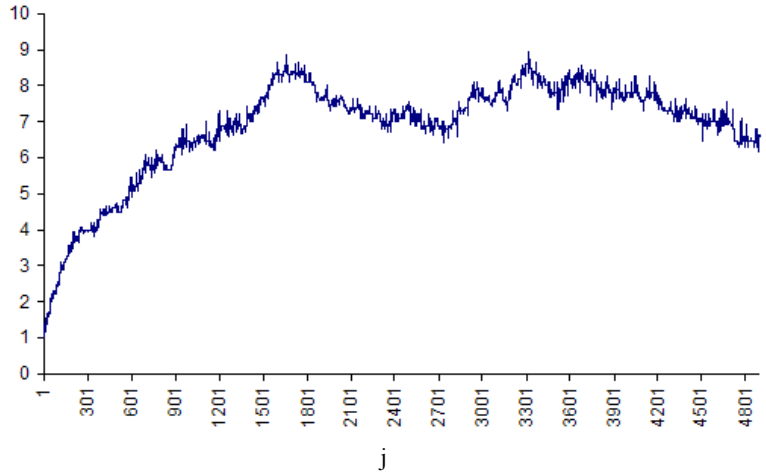
5.5.4 Calentamiento del simulador (warm-up)

En el simulador utilizamos el método descrito en el ítem 5 del apartado <4.3.2 Calentamiento (warm-up)> pero con una variación: calculamos el cambio relativo en el promedio de los datos que quedan luego de eliminar los datos iniciales, pero debido a que usamos la cantidad de *MH*s conectados como un indicador del estado de la simulación y este dato posee gran variabilidad, utilizamos un filtro para eliminar el “ruido” y obtener un resultado más confiable. En nuestro caso, los “datos” son registros de eventos en el log.

En los párrafos que siguen denominamos “observaciones” a los registros de tipo 2 (estadísticas), que son generados periódicamente – luego de una cantidad de registros de tipo 1 (eventos) – y que incluyen la cantidad de *MH*s al momento de la registración.

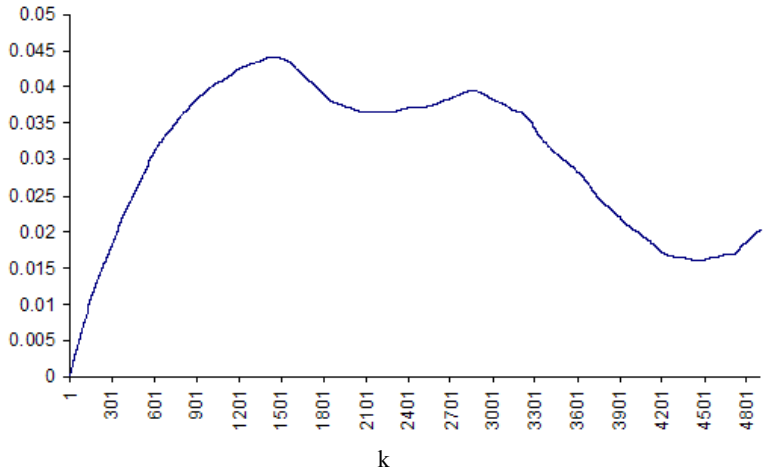
Los gráficos que se presentan a continuación muestran el funcionamiento del algoritmo utilizado, sobre un fragmento de datos obtenidos del simulador (las primeras 5000 observaciones). El algoritmo calcula en primer lugar el promedio por cada observación j de la cantidad de *MH*s en 10 corridas, como se aprecia en la Figura 3.

Figura 3: Evolución de la cantidad de MHs luego de promediar 10 corridas



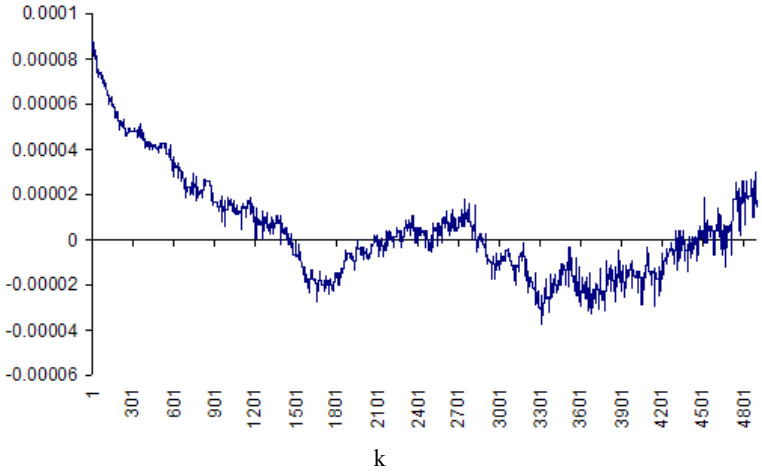
Luego, como se ve en la Figura 4, calcula el promedio parcial eliminando las primeras k observaciones y, con el promedio total , calcula el cambio relativo .

Figura 4: Cambio relativo en el promedio de MHs



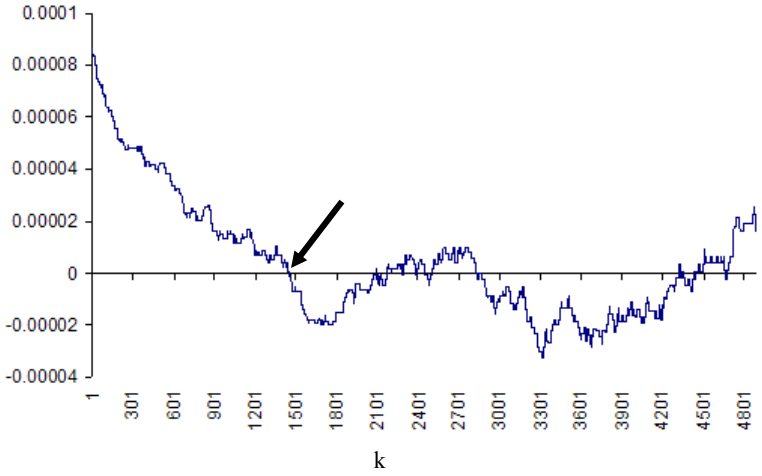
A continuación calcula la variación en el cambio relativo, como muestra la Figura 5.

Figura 5: Variación en el cambio relativo



Luego suaviza esta curva aplicando el filtro que promedia una ventana deslizante de 5 valores de consecutivos, obteniendo así , como muestra la Figura 6.

Figura 6: Variación suavizada



El punto donde cruza el cero por primera vez (aproximadamente 1450 en el ejemplo de la Figura 6), es el número de observaciones que deben ser descartadas.

5.6 Validación y verificación del simulador

Si bien ninguna técnica de verificación ni de validación puede asegurar con absoluta certeza que el sistema construido es el correcto y que funciona correctamente, la aplicación de las técnicas propuestas por [Raj91] (ver <4.3.1 Validación y verificación>) nos permite afirmar, con cierto grado de confianza, que hemos implementado el simulador correcto, correctamente. Los resultados de las simulaciones sobre los que basamos el análisis y posterior conclusión (ver <Capítulo 6: Corridas y Resultados>) fueron obtenidas luego de que el simulador superó exitosamente todas las pruebas que detallamos a continuación.

Inspección de código

Durante el desarrollo del simulador y de las políticas de los modelos elegidos para este estudio se realizaron inspecciones cruzadas, contrastando lo implementado con las definiciones teóricas de los modelos. También se usó la técnica combinar dos personas para las tareas de programación: mientras el programador explica sus razonamientos, otro integrante del equipo sigue la tarea del primero, realiza observaciones y plantea dudas.

Pruebas funcionales

La aplicación de técnicas de prueba funcional resulta particularmente difícil en sistemas estocásticos como el simulador que nos ocupa, dado que no es posible crear por demanda condiciones de entrada específicas para las funcionalidades más interesantes o críticas. Por ejemplo, una condición interesante es la secuencia de eventos: *TxBegin*, *NetDisconnect*, *TxCommit*, *NetReconnect*. Dado que los eventos se generan como resultado de procesos aleatorios, no es posible "configurar" el simulador para que genere exactamente este caso en forma limpia, es decir solamente esta secuencia de eventos.

Sin embargo, aplicamos las técnicas de prueba funcional para condiciones sobre las que sí tenemos control: la configuración de los parámetros de entrada. Incluso en este caso, debido a la gran cantidad de parámetros, resulta imposible realizar una prueba completa con todas las posibles clases de equivalencia. En este caso utilizamos nuestro conocimiento del simulador y de la influencia de las distintas variables sobre la simulación para obtener un subconjunto reducido de condiciones de prueba, diseñado para someter al simulador a situaciones de complejidad creciente. Estas condiciones de prueba se detallan en la Tabla 6.

Cantidad de *MH*	Transacciones por *MH*	Con Salto de Celda?	Con desconexión?
1	1	No	No
> 1	1	No	No
1	> 1	No	No
> 1	> 1	No	No
Aleatorio	Aleatorio	No	No
1	1	Si	No
1	1	No	Si
1	1	Si	Si
> 1	> 1	Si	Si
Aleatorio	Aleatorio	Si	Si

Tabla 6: Condiciones de prueba progresiva

Los resultados esperados para cada una de las condiciones de prueba son imposibles de especificar en forma concisa, no sólo por la complejidad de la salida de la "función" probada (el log de una simulación completa), sino por su naturaleza aleatoria. Por lo tanto, fue necesario aplicar, sobre estas corridas, otras técnicas de validación y verificación de los resultados: verificación de invariantes, inspección visual del log y análisis de las estadísticas.

No obstante las limitaciones naturales de un sistema estocástico para generar por demanda condiciones específicas de prueba, sometimos al simulador a las condiciones más interesantes por el simple método de realizar corridas hasta que la condición buscada se diera en forma espontánea. Las condiciones más frecuentemente verificadas se resumen en la Tabla 7.

Condición	Secuencia de Eventos
Salto de celda	*NetHandoff*
Desconexión	*NetDisconnect*, *NetReconnect*
Desconexión con salto de celda [4]	*NetDisconnect*, *NetHandoff*, *NetReconnect*
Transacción limpia	*TxBegin*, *TxCommit* / *TxAbort*
Transacción con salto de celda	*TxBegin*, *NetHandoff*, *TxCommit* / *TxAbort*
Transacción con desconexión incluida	*TxBegin*, *NetDisconnect*, *NetReconnect*, *TxCommit* / *TxAbort*
Transacción con comienzo desconectado	*NetDisconnect*, *TxBegin*, *NetReconnect*, *TxCommit* / *TxAbort*
Transacción con fin desconectado	*TxBegin*, *NetDisconnect*, *TxCommit* / *TxAbort*, *NetReconnect*
Transacción con múltiple salto de celda	*TxBegin*, *NetHandoff*, *NetHandoff*, *TxCommit* / *TxAbort*
Transacción con múltiple desconexión incluida	*TxBegin*, *NetDisconnect*, *NetReconnect*, *NetDisconnect*, *NetReconnect*, *TxCommit* / *TxAbort*
Transacción con salto de celda y desconexión	*TxBegin*, *NetHandoff*, *NetDisconnect*, *NetReconnect*, *TxCommit* / *TxAbort*
Transacción con desconexión y salto de celda	*TxBegin*, *NetDisconnect*, *NetReconnect*, *NetHandoff*, *TxCommit* / *TxAbort*

Tabla 7: Condiciones de prueba compleja

Inspección visual del log

Una de las técnicas más utilizadas fue la inspección visual del archivo de log del simulador. Para facilitar esta tarea agregamos varios tipos de registro, en particular el registro tipo 3 (comentarios), insertando en puntos críticos de los algoritmos mensajes informativos de las condiciones actuales. A modo de ejemplo, presentamos en la Figura 7 un pequeño fragmento de un log, que muestra registros de tipo 1 (evento) y 3 (comentario) emitidos durante el procesamiento de dos eventos.

```
101;4;WeakStrict;14/02/2003 01:12:19;76793;3;Starting event tb-16 orig m-2 dest d-1 sender m-2 handler m-2
101;4;WeakStrict;14/02/2003 01:12:19;76793;3;Using policy tb-m WeakStrict_Begin
101;4;WeakStrict;14/02/2003 01:12:19;76793;3;WeakStrict TxBegin in MH
101;4;WeakStrict;14/02/2003 01:12:19;76793;3;adding event tb-23 dest d-1 handler m-2 wakeup 76794
101;4;WeakStrict;14/02/2003 01:12:19;76793;3;17 has a new partner: 24
101;4;WeakStrict;14/02/2003 01:12:19;76793;3;24 has a new partner: 17
101;4;WeakStrict;14/02/2003 01:12:19;76793;3;Event tb-16 sent to DBMS
101;4;WeakStrict;14/02/2003 01:12:19;76793;3;adding event ta-25 dest m-2 handler m-2 wakeup 1446376
101;4;WeakStrict;14/02/2003 01:12:19;76793;3;Using policy tb-b WeakStrict_Begin
101;4;WeakStrict;14/02/2003 01:12:19;76793;1;tb;16;m;2;m;2;m;2;m;2;1;;17;2;17;0;1;
101;4;WeakStrict;14/02/2003 01:12:19;76793;3;Asking factory m-2 to make event of kind tb (1 to go)
101;4;WeakStrict;14/02/2003 01:12:19;76793;3;adding event tb-26 dest d-1 handler m-2 wakeup 2254514 (factory)
101;4;WeakStrict;14/02/2003 01:12:19;76794;3;Starting event tb-23 orig m-2 dest d-1 sender m-2 handler m-2
101;4;WeakStrict;14/02/2003 01:12:19;76794;3;Using policy tb-m WeakStrict_Begin
101;4;WeakStrict;14/02/2003 01:12:19;76794;3;WeakStrict TxBegin in MH
101;4;WeakStrict;14/02/2003 01:12:19;76794;3;Event tb-23 sent to DBMS
101;4;WeakStrict;14/02/2003 01:12:19;76794;3;forwarding event tb-23 dest d-1 handler s-200001 wakeup 77516
101;4;WeakStrict;14/02/2003 01:12:19;76794;1;tb;23;m;2;d;1;m;2;m;2;0;WeakStrict_Begin;24;2;17;0;0;
```

Figura 7: Fragmento del log

Verificación de invariantes y postcondiciones

Incluimos en el código la verificación automática de invariantes y postcondiciones, con las que verificamos el correcto funcionamiento del simulador y los modelos. Cuando una invariante o postcondición no se cumple, se emite un mensaje de error y la simulación se detiene. La sensibilidad de esta técnica permitió detectar errores sutiles, resultado de las complejas interacciones entre las políticas de un modelo. En la Tabla 8 presentamos una lista parcial de las invariantes y postcondiciones utilizadas.

[4] En este caso el salto de celda tiene lugar inmediatamente después de la reconexión

Invariante / postcondición	Descripción
weakEventRelations	Luego de cada evento, se mantienen las siguientes relaciones: *#NetNewMH* >= #{*MH*s en la red} *#NetNewMH* >= *#NetDestroyMH* *#NetDisconnect* >= *#NetReconnect* *#TxBegin* >= #{transacciones activas} *#TxBegin* >= *#TxAbort* + *#TxCommit*
strongEventRelations	Al finalizar una simulación, se cumplen las siguientes condiciones: #{*MH*s en la red} = 0 #{transacciones activas} = 0 *#NetNewMH* - *#NetDestroyMH* = #{*MH*s en la red} *#TxBegin* = *#TxAbort* + *#TxCommit* *#NetDisconnect* = *#NetReconnect*
timeFlowsForward	El tiempo de activación del siguiente evento es mayor o igual que la hora simulada actual
parentChildrenLinksOK	Luego de agregar una subtransacción, la estructura jerárquica es consistente (la transacción es padre de todos sus hijos)
eventHasNotBeenCountedBefore	Un evento no fue contabilizado más de una vez en las estadísticas
subTxCountNotExceeded	Luego de crear subtransacciones, no se excedió el límite
canBeCommited	Una transacción está en condiciones de finalizar exitosamente
canBeAborted	Una transacción está en condiciones de abortar
notAbortingCompensatory	La transacción que se está abortando no es compensatoria
notConnected	Luego de un evento de desconexión, el *MH* no está conectado
connected	Luego de un evento de reconexión, el *MH* está conectado
onlyOneIteratedSetting	No hay más que una variable iterada en el archivo de configuración
knownDistribution	La distribución mencionada en el archivo de configuración es conocida
enoughOperands	Un operador de la fórmula de iteración tiene suficientes operandos en la pila
oneResult	Al final del cálculo de la formula de iteración, queda un único valor en la pila

Tabla 8: Invariantes y postcondiciones

Inspección de los resultados gráficos

La inspección de la salida gráfica resultante del procesamiento del log permitió contrastar las estadísticas obtenidas con el comportamiento esperado a partir de nuestro conocimiento de los modelos. Utilizando esta técnica fue posible detectar fallas evidentes, como el fracaso de todas las transacciones de un determinado modelo, aún bajo condiciones favorables, o fallas más sutiles, como una discrepancia en un valor del gráfico con el valor teórico.

5.7 Instanciación de los modelos

Al instanciar los modelos móviles en el simulador, observamos que algunos de estos modelos están subespecificados: se concentran en describir una solución a un problema particular, quedando fuera del alcance cómo resuelven el resto de los problemas de un modelo móvil. Por ejemplo, [Pit95] que presenta WeakStrict no describe cómo actúa el modelo ante los saltos de celda. Consecuentemente para implementar cada uno de los modelos completamos su especificación tratando de interpretar la intención de los autores.

Desconexión: Para los modelos que no especifican un comportamiento ante el evento de desconexión, asumimos que en dicha situación sus transacciones deben abortar, como sucede en las bases de datos comerciales cuando la aplicación pierde la conexión al servidor antes de llegar al commit.

Salto de celda: Ya que la intención de este trabajo es someter los diferentes modelos a los problemas que presentan los ambientes móviles, no asumimos soporte por parte del hardware para resolver dichos problemas. En particular, modelamos un salto de celda como una desconexión para aquellos modelos que no hacen mención al tratamiento de este evento.

Esta decisión fue tomada después de considerar otras alternativas, como demorar los paquetes enviados al *MH* luego del salto de celda por complicarse el ruteo de los mismos. Sin embargo, entendemos que no es el ruteo de los paquetes en la red fija lo que un modelo móvil debe resolver. En cambio, es razonable pensar que un modelo que no soporta el evento de salto de celda perciba un salto de celda como la desconexión de un *MSS* y la posterior reconexión al nuevo *MSS*.

A continuación se describe la manera en que se implementaron en el simulador los modelos elegidos para la comparación. En el apéndice (ver <1.6 Políticas de los modelos implementados>) se detalla su especificación al nivel de las políticas individuales.

Flat (tradicional)

El modelo plano es implementado por las políticas predefinidas. Esto implica que ante un evento de salto de celda o desconexión, las transacciones son abortadas. Las transacciones son lanzadas desde el *MH* y residen en el *DBS*. No genera subtransacciones ni transacciones compensatorias.

Kangaroo

Las políticas que definen este modelo determinan que la transacción principal (KT) se lanza desde el *MH*. Esta llega al *MSS*, quien lanza subtransacciones (*JTs*) las que residen en el *DBS*. La principal también reside en el *DBS* pero su evento de finalización generado automáticamente por el *DBMS* se descarta, ya que su resultado final depende del resultado de las subtransacciones.

Al terminar una subtransacción exitosamente, evalúa si quedan subtransacciones por generar. En caso afirmativo genera las nuevas subtransacciones en el *MSS* actual. En caso contrario, produce un evento de commit para la transacción principal.

Si alguna subtransacción fracasa, se genera un evento de abort para la principal. Además, recorre los *MSS* por los que el *MH* pasó anteriormente buscando subtransacciones completadas en forma exitosa. Si Kangaroo corre en modo compensatorio, se ejecutan las transacciones compensatorias; si corre en modo split, para cada subtransacción se decide por distribución si se compensa o no.

Las desconexiones son ignoradas; y ante un salto de celda provee el comportamiento por defecto, sin afectar la administración de las transacciones.

Moflex

Este modelo lanza subtransacciones en el *MSS* las cuales residen en el *DBS*. Ante el inicio de la transacción original, se generan por distribución subtransacciones en el *MSS*. La transacción original también reside en el *DBS* pero se ignora su evento de finalización generado automáticamente por el *DMBS*.

Al terminar una subtransacción, evalúa mediante distribuciones (distintas según si terminó con éxito o fracaso) si llegó a un estado aceptable. Si es así y el *MH* se encuentra conectado, produce en evento de commit para la transacción principal; mientras que si el *MH* no se encuentra conectado, genera un evento de abort que llegará al *MH* cuando éste se reconecte. Si se determinó que no arribó a un estado aceptable, se generan subtransacciones en forma análoga al tratamiento de la transacción principal hasta agotar la cantidad predefinida de subtransacciones. Si no quedan subtransacciones por generar, se envía un evento de abort para la principal.

Ante un salto de celta, para cada subtransacción y según un parámetro del archivo de configuración, determina la regla de salto de celda a aplicar (restart, continue, split-restart, split-continue). Por otra parte, el evento de desconexión es ignorado.

IOT

En este modelo, la transacción principal es lanzada por el *MH* y reside en una base de datos local al *MH*. Para cada transacción principal se genera una segunda transacción asociada, que reside en el *DBS*. Esto modela el sistema de "cache" de IOT.

Ante una desconexión, el modelo adopta el comportamiento por defecto: la transacción asociada es abortada. Esto es usado por el modelo como memoria de la desconexión: siempre que la transacción asociada se encuentre abortada, significa que hubo una desconexión.

El evento de finalización exitosa de la transacción principal es reenviado al *DBS*, donde se determina si el *MH* tuvo una desconexión en algún momento durante el transcurso de la transacción. Esto se detecta por el estado de la transacción asociada, que estará abortada si hubo una desconexión. Si no hubo tal desconexión, el *DBS* contesta con otro evento commit, modelando las transacciones de primera clase. Si se produjo alguna desconexión, se trata de una transacción de segunda clase y debe ser sincronizada. Esto se modela con una distribución que determina si el *DBS* responde con un commit o con un abort, indicando el resultado de la sincronización. En caso de que la transacción principal fracase se utiliza un mecanismo similar, salvo que no hay sincronización: la respuesta del *DBS* es siempre abort.

Si al momento de finalizar la transacción principal el *MH* se encuentra desconectado, la transacción queda en estado pendiente de sincronización. Cuando se produce la reconexión, el *MH* envía el evento de finalización al *DBS* para esperar la respuesta de la sincronización.

Ante un salto de celda el modelo utiliza el comportamiento por defecto, es decir que aborta la transacción asociada residente en el *DBS*, por lo que cuando la transacción principal termine, si lo hace exitosamente, deberá sincronizarse.

WeakStrict

La transacción principal se lanza desde el *MH* y reside en el mismo *MH*. Al momento de crear la transacción se determina mediante una distribución si todos sus datos están disponibles en le ambiente local en cuyo caso se categoriza a la transacción como débil. Si la transacción es estricta y el *MH* se encuentra conectado se crea una transacción asociada, Si el *MH* esta desconectado y la transacción principal es estricta, esta se aborta inmediatamente.

La finalización de una transacción principal estricta es determinada por la asociada. Si la transacción principal es débil y finaliza exitosamente, envía al *DBS* un evento de sincronización y espera su respuesta. El *DBS* determina por distribución el resultado de la sincronización y lo envía al *MH*. Al recibir esta respuesta, el *MH* genera un evento commit o abort, según corresponda, para la transacción principal. En caso de encontrarse desconectado, el evento de sincronización se envía al momento de la reconexión.

Ante una desconexión o un salto de celda, se abortan las transacciones principales estrictas No se transforma la transacción local en débil porque si inicialmente decidió que debía ser estricta es porque no tenía los datos en el cluster local (entendemos que el objetivo del modelo es usar débil siempre que se pueda).

Capítulo 6: Corridas y Resultados

6.1 Objetivo de las corridas

El objetivo de las corridas es comparar el comportamiento de los distintos modelos transaccionales móviles y evaluarlos cuantitativamente para identificar, ratificar o rectificar claramente sus diferentes características y reacciones al someterlos a diversas situaciones. Los modelos móviles que comparamos fueron Moflex, Kangaroo, WeakStrict e IOT; asimismo incorporamos Flat en las mediciones para contrastar el modelo transaccional tradicional contra los modelos móviles.

Utilizamos el simulador desarrollado (ver <Capítulo 5: Diseño y Construcción del Simulador>) para tomar las mediciones necesarias para efectuar el estudio, y decidimos efectuar dichas mediciones desde dos puntos de vista: el tecnológico y el de un usuario de aplicaciones móviles.

Para definir los indicadores que permiten analizar el comportamiento de los modelos desde el punto de vista del usuario, asumimos que un usuario espera que las aplicaciones móviles sean eficientes y robustas. Por lo tanto, medimos los siguientes indicadores:

Eficiencia desde el punto de vista del usuario: la eficiencia es percibida por el usuario como la velocidad de respuesta del sistema frente a sus requerimientos. Medimos esto como la cantidad de transacciones exitosas por unidad de tiempo.

Costo del fracaso desde el punto de vista del usuario: el usuario no quiere invertir gran cantidad de tiempo en operaciones que finalmente fracasarán. Por lo tanto, el fracaso de las operaciones debe ser comunicado al usuario lo antes posible. Medimos esto como la relación de duración entre transacciones abortadas y exitosas.

Robustez del modelo: dado que el entorno móvil presenta numerosos desafíos para el éxito de las operaciones transaccionales, el usuario espera que un modelo robusto administre estas dificultades, impidiendo que deriven en el fracaso de sus transacciones. El porcentaje de transacciones principales exitosas es el indicador elegido para representar este aspecto.

Desde el punto de vista tecnológico, se espera que los modelos móviles produzcan el menor impacto posible en la infraestructura física. Por lo tanto, definimos los siguientes indicadores:

Eficiencia desde el punto de vista de la utilización de recursos: se requiere que la carga en los enlaces de comunicación sea mínima para completar una transacción. Medimos este indicador como la cantidad de paquetes enviados por transacción exitosa.

Costo del fracaso desde el punto de vista de la utilización de recursos: es deseable que los recursos utilizados para administrar una transacción que fracasa sean los mínimos posibles. Esto lo medimos como la cantidad de paquetes invertidos en transacciones abortadas.

Carga de trabajo asumida por el MH: dado que el uso de la red inalámbrica tiene un mayor costo y probabilidad de error que la red fija y tomando en cuenta las limitaciones de los dispositivos móviles, se espera que un modelo móvil delegue la mayor parte posible de su carga de trabajo en los componentes fijos. Este aspecto es medido con el porcentaje de paquetes de radio por transacción.

Como estamos estudiando los modelos en ambientes móviles, es fundamental considerar en estas mediciones los aspectos que caracterizan fuertemente a estos ambientes, es decir:

Nivel de soporte de movilidad: el sistema debe mantener la ejecución de la transacción más allá de los cambios de celda de la computadora móvil.

Nivel de soporte de desconexiones: ya que el sistema debe poder ejecutar transacciones aún cuando la unidad móvil se encuentre desconectada.

Manejo eficiente de la concurrencia: ya que habitualmente se trabaja sobre base de datos globales en forma concurrente

Nivel de soporte ante transacciones de larga duración: por la movilidad del cliente y por las posibles desconexiones durante el transcurso de una transacción

Para cada uno de estos factores, efectuamos un seguimiento de su evolución en varias corridas y mostramos en los gráficos cómo van cambiando los resultados al variar el factor, por ejemplo, cómo evoluciona la cantidad de transacciones a medida que aumenta la frecuencia de cambio de celda.

Cada una de estas mediciones, y en forma comparativa entre los modelos que se estudian, se ve reflejada en su correspondiente gráfico.

6.2 Planificación de las corridas

A fin de poder efectuar el estudio comparativo de los distintos modelos transaccionales móviles elaboramos un plan de ejecución consistente en las siguientes etapas:

1. Diseñar o seleccionar un conjunto de aplicaciones móviles con características diferentes.
2. Instanciar dichas aplicaciones en el simulador, manipulando parámetros y distribuciones de probabilidad en el archivo de configuración.
3. Simular estas aplicaciones con cada uno de los modelos
4. Obtener los gráficos comparativos
5. Analizar dichos gráficos

En la etapa 1, elegimos el conjunto de aplicaciones que utilizamos para medir los aspectos que nos interesan sobre cada modelo transaccional móvil. Para medir el nivel de soporte de movilidad elegimos un sistema de emergencias de ambulancias, para medir el soporte ante desconexiones elegimos una aplicación de seguimiento de tornados o "twister", para evaluar el manejo eficiente de la concurrencia una edición de un libro en colaboración y finalmente, para medir el soporte de transacciones de larga duración elegimos la aplicación de sonda espacial.

En la etapa 2, como primer paso antes de instanciar las aplicaciones en el simulador, definimos una taxonomía para clasificar las aplicaciones en base a las principales propiedades de los sistemas transaccionales móviles. Los aspectos que seleccionamos para esta clasificación son:

- Probabilidad de desconexión
- Tiempo medio de desconexión
- Probabilidad de salto de celda
- Cantidad de *MH*s simultáneos
- Cantidad de transacciones lanzadas
- Duración de la transacción principal
- Ancho de banda de las transmisiones radiales

En la Tabla 9 presentamos una clasificación de las aplicaciones seleccionadas en función de estos aspectos. También agregamos una columna denominada "Utilizada de ejemplo en el Modelo", en la que indicamos si esta aplicación fue sugerida en la presentación de un modelo en particular, en cuyo caso acentuamos la validación de las afirmaciones efectuadas en dicha presentación.

Vemos en la Tabla 9 que las aplicaciones elegidas cubren relativamente bien el espectro de valores posibles para los factores que definen las características de un ambiente móvil.

Nombre de la Aplicación	Probabilidad de Desconexión	Tiempo medio de desconexión	Probabilidad de salto de celda	Cantidad de *MHs* simultáneos	Cantidad de Transacciones lanzadas	Duración de la Transacción Principal	Ancho de Banda de Transmisión	Utilizada de ejemplo en el Modelo
Edición de un Libro en Colaboración	Baja	Medio	Baja[5]	Variable, entre baja y media	Pocas	Larga[6]	Medio[7]	Weak Strict
Seguimiento de Tornados	Variable entre media y alta	Corto	Baja	Uno	Media	Media	Medio	-
Sistema de Emergencias de Ambulancias	Baja	Corto	Variable entre baja y alta	Media	Pocas	Media	Medio	Moflex
Sonda Espacial	Muy Baja	Corto	Baja	Uno	Muchas	Depende del ancho de banda	Variable entre bajo y alto	-

Tabla 9: Aspectos característicos por aplicación

Para finalizar con la etapa 2, instanciamos cuantitativamente las columnas de la tabla mediante distribuciones de probabilidad (ver <1.4 Distribuciones utilizadas>). Dicha instanciación se efectuó en el archivo de configuración del simulador.

Luego se corrieron las simulaciones, se obtuvieron los gráficos comparativos y se efectuó el análisis de los mismos, cumpliendo con las etapas 3. 4. y 5., según se detalla en la siguiente sección del trabajo.

6.3 Resultados de las corridas y su análisis

A continuación detallamos cada aplicación elegida, su instanciación en el simulador, los gráficos obtenidos mediante el procesamiento de los resultados de las corridas y el análisis efectuado sobre los mismos.

6.3.1 Sistema de emergencias de ambulancias

En esta aplicación, una ambulancia busca reservar una cama en un hospital mientras se desplaza con un paciente. Si un hospital no tiene camas libres, prueba con otro hasta encontrar una. Distintos pacientes presentan distintos grados de urgencia, por lo que la ambulancia puede moverse a distintas velocidades por la ciudad mientras efectúa la búsqueda.

El objetivo de esta aplicación es someter a los modelos a un escenario con distintos grados de movilidad.

[5] Es una decisión arbitraria, fijamos que en esta aplicación los usuarios trabajarán mayormente en su casa u oficina.
[6] Entendemos por transacción principal a la tarea de escribir el Capítulo, con pocas actualizaciones de índices y referencias cruzadas..
[7] Asumimos que la PC donde se escribe el libro dispone de una conexión dial-up para realizar las actualizaciones necesarias.

Instanciación en el simulador

Las principales variables que se instanciaron en el simulador son las siguientes:

- ***Probabilidad de desconexión***
  Aproximadamente una desconexión cada 30 minutos, por pasar por una sombra de radio.
  - *NetDisconnect_Distribution: ExponentialDistribution(0.0000005)*

- ***Tiempo medio de desconexión***
  Se reconecta ni bien sale de la sombra de radio (5 segundos).
  - *NetReconnectDelay: ExponentialDistribution(0.0002)*

- ***Duración de la transacción principal***
  Una ambulancia está ocupada atendiendo una emergencia durante aproximadamente 20 minutos, luego de lo cual pasan aproximadamente 10 minutos hasta la siguiente.
  - *DBMS_AccessesPerTx: GaussianDistribution(25,5)*
  - *DBMS_TimeBetweenAccesses: ConstantDistribution(44000)*
  - *DBMS_AccessesProcessingTime: ConstantDistribution(1000)*
  - *NetLinkDelay_Radio_MHtoMSS: ExponentialDistribution(0.0005)*
  - *NetLinkDelay_Radio_MSStoMH: ExponentialDistribution(0.001)*
  - *NetLinkDelay_Radio_Symetrical: false*

- ***Probabilidad de éxito de las transacciones***
  En hora pico suponemos que la mitad de los hospitales no tienen camas libres.
  - *DBMS_txSuccess: BernoulliDistribution(0.5)*

- ***Probabilidad de salto de celda***
  En caso de emergencia cambia de celda cada 2 minutos, si es menos urgente, cada hora.
  - *NetHandoff_Distribution: ConstantDistribution(120000;6;696000+)*

- ***Cantidad de transacciones lanzadas***
  Cada ambulancia atiende en promedio 3 emergencias durante la hora y media que dura la simulación. Cada emergencia es una transacción (obtener la cama) e implica las siguientes subtransacciones: buscar datos del paciente, buscar hospitales especializados, buscar una cama libre probando en distintos hospitales hasta encontrar una, y finalmente, informar el estado del paciente al hospital elegido. Suponemos que es necesario interrogar a lo sumo cuatro hospitales antes de encontrar una cama libre.
  - *MH_TransactionsPerMH: GaussianDistribution(3,1)*

- ***Cantidad de MHs simultáneos***
  Se modelaron 7 ambulancias simultáneas
  - *NetNewMH_Distribution: GaussianDistribution(7,0)*
  - *NetNewMH_Delay_Distribution: Exponential(0.00002)*

- ***Ancho de banda de transmisión***
  La velocidad de transmisión de un paquete es de aproximadamente tres segundos.
  - *NetLinkDelay_Radio_MHtoMSS: ExponentialDistribution(0.0005)*
  - *NetLinkDelay_Radio_MSStoMH: ExponentialDistribution(0.001)*
  - *NetLinkDelay_Wire: ExponentialDistribution(0.01)*

- ***Otros valores***
  La probabilidad de bloqueo es alta (0.8) porque todos quieren acceder al registro para obtener la cantidad de camas libres, pero el tiempo que dicho registro permanece bloqueado es muy bajo (5% de la subtransacción) ya que decrementar el número de camas libres lleva muy poco tiempo.

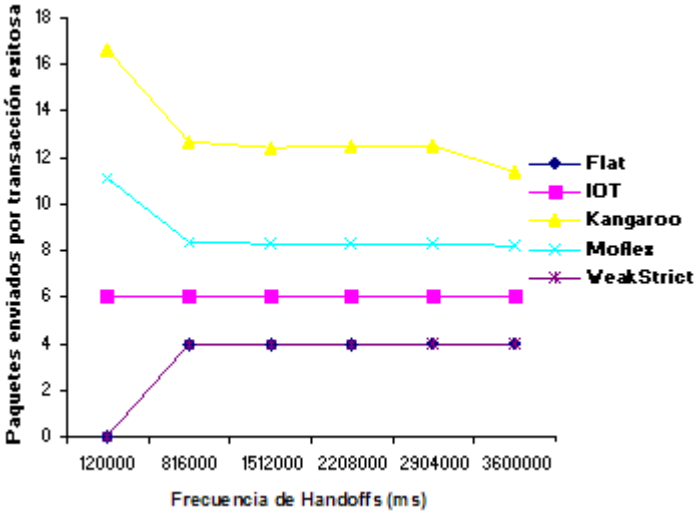
- *Probabilidad de lock: 0.8*

La variación considerada en este caso es sobre la *frecuencia de cambio de celda*, efectuando uno desde una vez cada 2 minutos (en caso de emergencia) hasta una vez cada 17 minutos.

Análisis de los resultados obtenidos

A continuación mostramos las mediciones obtenidas y su correspondiente análisis:

Eficiencia desde el punto de vista de la utilización de recursos



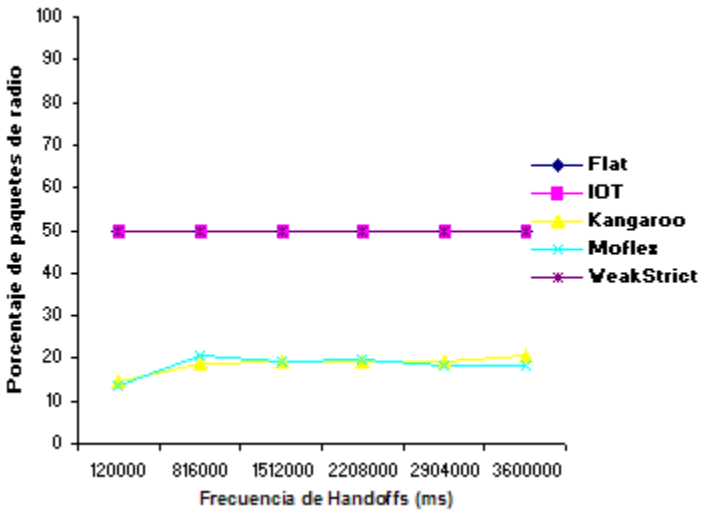
En el caso de WeakStrict y de Flat, como ambos modelos son sensibles al salto de celda, ninguna de sus transacciones sobrevive a dicho evento hasta que el mismo sucede en forma lo suficientemente infrecuente. Particularmente WeakStrict aborta sus transacciones ante el evento de salto de celda cuando las mismas son *strict*, caso constante en esta aplicación.

Por su parte IOT tiene un costo fijo debido a que no utiliza los *MSS* para la administración de sus transacciones y por lo tanto su costo es independiente del número de *MSS*s involucrados en su ejecución.

Se observa que Moflex tiene un costo más bajo que Kangaroo debido a que Moflex termina exitosamente con la primer subtransacción exitosa (primer estado aceptable modelado en el archivo de configuración), mientras que Kangaroo requiere que todas las subtransacciones sean exitosas (más allá de la instanciación hecha para la aplicación).

La diferencia entre 4 o 6 paquetes por transacción exitosa (en los casos de IOT, Flat y WeakStrict) se debe a la presencia o ausencia de subtransacciones, cuya administración implica la presencia de más paquetes en la red simulada.

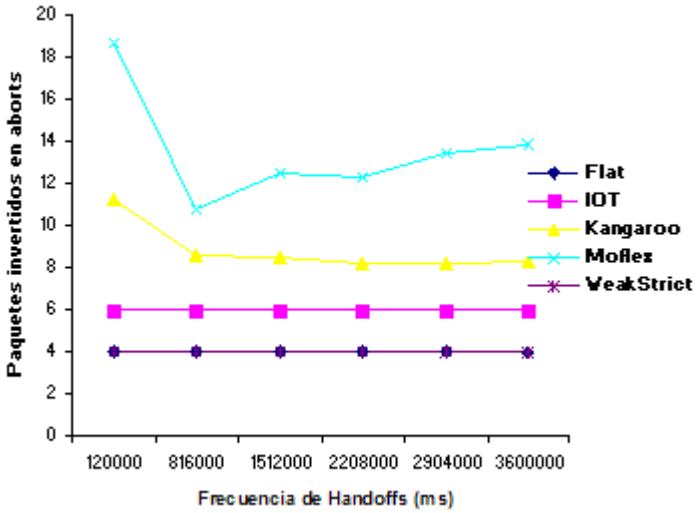
Carga de trabajo asumida por el *MH*



El porcentaje de paquetes de radio se mantiene constante para Flat, IOT y WeakStrict, dado que todo el trabajo lo asume el *MH* y cada paquete viaja un tramo por la red fija y otro por radio.

En cambio para Moflex y Kangaroo, por implementar sus transacciones en el *MSS*, se observa una menor carga en el *MH*. La misma disminuye relativamente por el mayor trabajo de los *MSS*s ante la mayor frecuencia de saltos de celda.

Costo del fracaso desde el punto de vista de la utilización de recursos

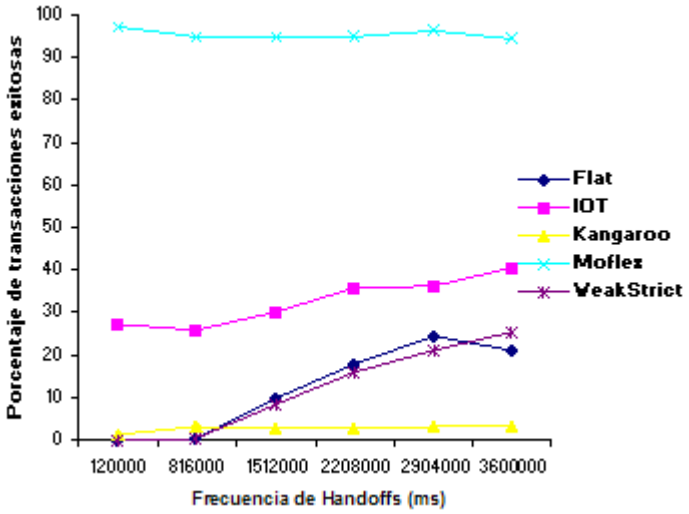


La cantidad de paquetes invertidos en aborts se mantiene constante para los modelos que no tienen definidas subtransacciones (Flat, IOT y WeakStrict) ya que la frecuencia de saltos de celda no influye en la cantidad de paquetes necesarios para resolver el *abort*. En el caso de Flat, como aborta ante el primer salto de celda, no hay diferencia en cuanto a qué proporción de la transacción transcurrió hasta ese momento porque el simulador no modela los paquetes de comunicación de datos y comandos dentro de cada subtransacción. Es de esperar que en la realidad la cantidad de paquetes invertidos en un *abort* disminuya a medida que aumente la frecuencia de saltos de celda. IOT y WeakStrict deciden el resultado al final de la transacción (en la sincronización), por lo que tampoco les afecta el momento en que se produce el salto de celda.

En cambio para Kangaroo, cuyas subtransacciones se ejecutan en el *MSS*, se puede observar que la cantidad de paquetes utilizados en el *abort* disminuye a medida que baja la frecuencia de saltos de celda porque disminuye la cantidad de paquetes afectados a la transferencia de entorno entre los *MSS*s involucrados en el salto de celda.

Moflex muestra que tiene un mayor costo de fracaso, ya que ejecuta todas las subtransacciones antes de abortar la transacción principal. Los picos en el gráfico se deben a que muy pocas transacciones de Moflex fracasaron, quedando una muestra muy pequeña para obtener resultados estables.

Robustez del modelo

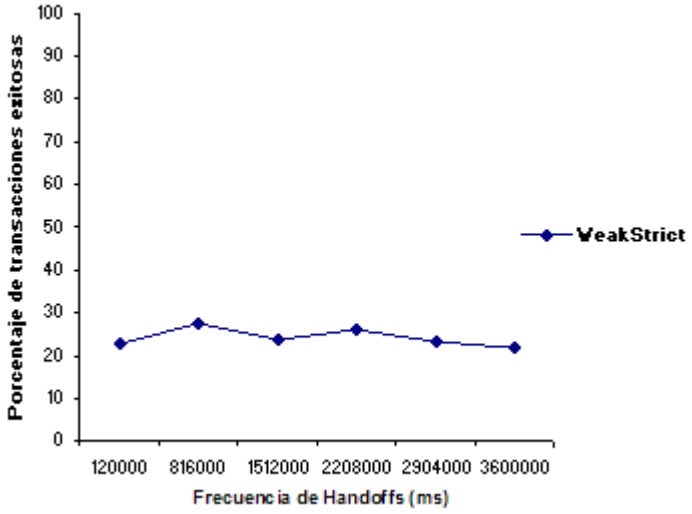


Claramente Moflex es el modelo más robusto para esta aplicación, como se esperaba.

Para IOT, a medida que aumenta la frecuencia de salto de celda, aumenta la influencia de la sincronización, ya que asumimos que el salto de celda en IOT equivale a una desconexión. Inicialmente cerca del 40% de las transacciones son exitosas ya que pocas transacciones (20%) son alcanzadas por un salto de celda y cada una tiene una probabilidad de éxito de 0.5. Hacia el final el porcentaje de transacciones exitosas se acerca al 25% ya que todas las transacciones son alcanzadas por un salto de celda, por lo que las exitosas son sincronizadas con una probabilidad de éxito de 0.5.

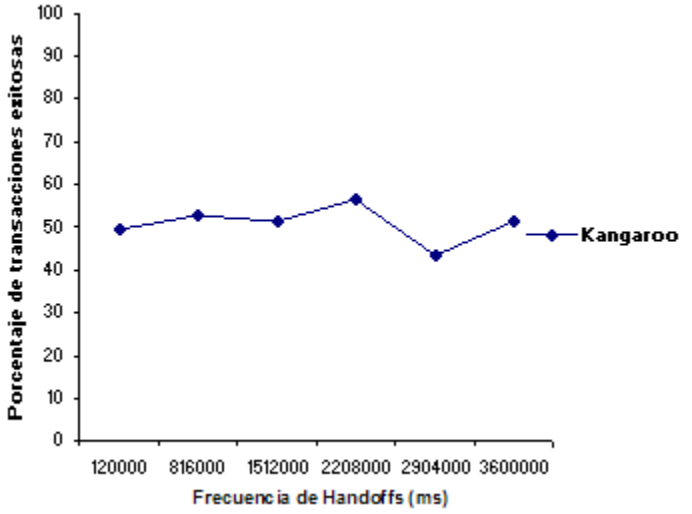
Tanto Flat como WeakStrict tienen pésima performance cuando sus transacciones son alcanzadas por saltos de celda, ya que no soportan este evento (en el caso de WeakStrict, esto sucede sólo cuando la transacción es *strict*). En el gráfico puede observarse que la efectividad de ambos modelos tiende a mejorar al disminuir la cantidad de transacciones afectadas por saltos de celda. Esta efectividad no llega al nivel de IOT porque, a diferencia de éste, Flat y las transacciones *strict* de WeakStrict no soportan desconexiones, que en esta aplicación no están ausentes.

En el siguiente gráfico podemos verificar que WeakStrict soporta saltos de celda cuando las transacciones son de tipo *weak*, viéndose en este caso afectado el éxito de las transacciones por la sincronización, resultando en un 25% de éxitos debido a que la probabilidad de éxito de una transacción y la de la sincronización son ambas del 50%:

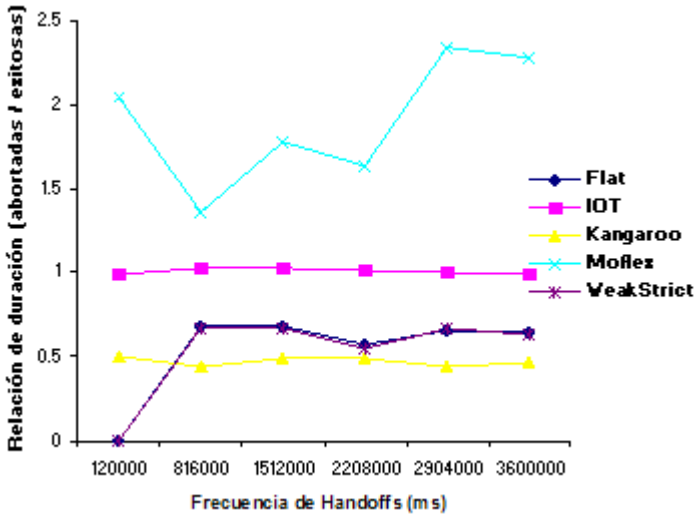


Kangaroo tiene muy baja performance ya que, por modelo, ante la primera subtransacción que aborta (el primer hospital que no tenga camas disponibles), produce el fracaso de toda la transacción.

Si para Kangaroo aumentamos la probabilidad de éxito de una transacción a 0.9, observamos que aumenta su robustez porque disminuye la probabilidad de que una de las primeras subtransacciones aborte. Dado que la cantidad de subtransacciones tiene una esperanza de 7, la probabilidad media de éxito de la transacción principal pasa a ser $0.9^7 \approx 0.47$, como se observa en el siguiente gráfico:



Costo del fracaso desde el punto de vista del usuario

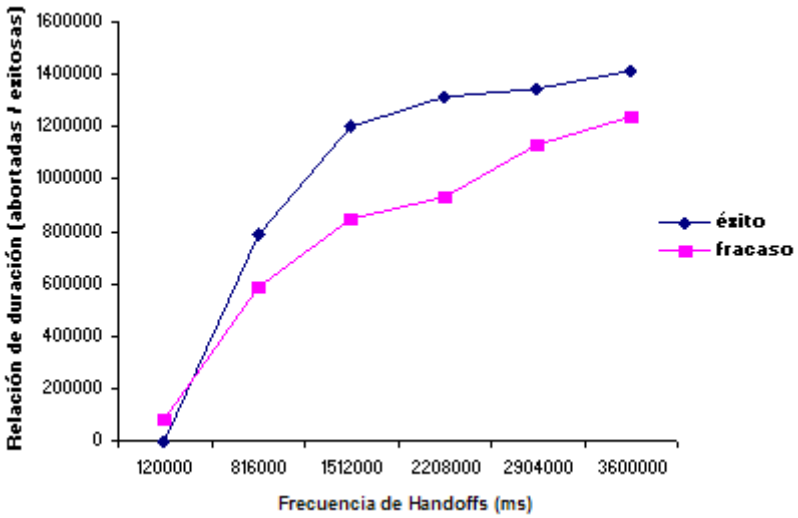


Kangaroo tiene un bajo costo de fracaso ya que finaliza ante el primer *abort*. Sin embargo, en esta aplicación éste no es un resultado deseable ya que significa que no agotó las posibilidades de encontrar una cama libre para el paciente; ante el primer hospital que no tiene cama libre, toda la transacción aborta.

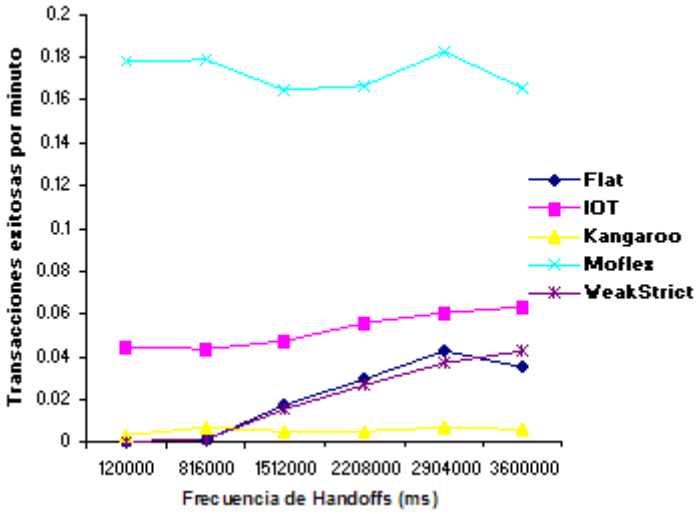
En cambio Moflex refleja un mayor costo ya que agota todas las instancias de búsqueda, mientras que finaliza exitosamente con la primer cama libre encontrada.

Para IOT, la duración de las transacciones abortadas es similar a las transacciones exitosas y como su duración tampoco es afectada por el salto de celda, el costo de fracaso es aproximadamente 1.

En el caso de WeakStrict, a medida que disminuye la frecuencia de saltos de celda esperábamos observar un mayor costo de fracaso, ya que el primer salto de celda ocurre, en promedio, más tarde, con lo que las transacciones abortadas ocupan recursos durante más tiempo. Sin embargo, esto no es evidente en el gráfico, debido a que las transacciones exitosas son justamente aquellas cuya duración les permitió completarse sin sufrir un salto de celda. Podemos comprobar esta afirmación mostrando las duraciones de las transacciones de WeakStrict, distinguiendo entre las exitosas y las abortadas:



Eficiencia desde el punto de vista del usuario



Se observa que Moflex produce una cantidad estable de transacciones exitosas por minuto, realizando aproximadamente una transacción exitosa cada 10 minutos. Se puede ver que una transacción cuya duración máxima estimada era de 20 minutos, la resuelve en 10 minutos porque finaliza ante la primer subtransacción exitosa.

IOT, Kangaroo, Flat y WeakStrict reflejan la performance que se observó en el gráfico de robustez.

6.3.2 Seguimiento de tornados

Esta aplicación representa a un equipo de científicos, moviéndose en una camioneta, que persiguen tornados para tomar mediciones de los mismos, mientras bajan imágenes satelitales y datos del clima y almacenan información del tornado. A medida que se acercan a la tormenta, las desconexiones son más frecuentes debido a las descargas eléctricas y a la estática.

El objetivo de esta aplicación es someter los modelos a un escenario con distintas frecuencias de desconexión.

Instanciación en el simulador

Las principales variables que se instanciaron en el simulador son las siguientes:

- ***Probabilidad de desconexión***
  Se produce una desconexión cada minuto cuando la camioneta se encuentra cerca del tornado hasta una cada 15 minutos cuando se aleja del mismo.
  - *NetDisconnect_Distribution: ExponentialDistribution (0.000016;6;1.7/)*

- ***Tiempo medio de desconexión***
  Se reconecta rápidamente (1 segundo)
  - *NetReconnectDelay: ExponentialDistribution(0.001)*

- ***Duración de la transacción principal***
  Cada transacción dura 15 minutos.
  - *DBMS_AccessesPerTx: GaussianDistribution(50,5)*
  - *DBMS_TimeBetweenAccesses: ConstantDistribution(1000)*

- *DBMS_AccessesProcessingTime: ConstantDistribution(4600)*
- *NetLinkDelay_Radio_MHtoMSS: ExponentialDistribution(0.0005)*
- *NetLinkDelay_Radio_MSStoMH: ExponentialDistribution(0.001)*
- *NetLinkDelay_Radio_Symetrical: false*

- ***Probabilidad de éxito de las transacciones***

  La probabilidad de éxito de cada subtransacción es alta, de 0.8 en promedio
  - *DBMS_txSuccess: BernoulliDistribution(0.8)*

- ***Probabilidad de salto de celda***

  Aproximadamente un salto de celda cada hora ya que se encuentra en zona de campos donde las celdas son grandes.
  - *NetHandoff_Distribution: ConstantDistribution(3600000)*

- ***Cantidad de transacciones lanzadas***

  Cada medición es una transacción e implica tres subtransacciones: bajar imágenes satelitales, bajar datos de radar y almacenar información del tornado. Se considera que la camioneta intenta hacer 10 mediciones del tornado.
  - *MH_TransactionsPerMH: GaussianDistribution(10,1)*

- ***Cantidad de MHs simultáneos***

  Se considera un solo *MH*, en la camioneta que se va desplazando.
  - *NetNewMH_Distribution: GaussianDistribution(1,0)*
  - *NetNewMH_Delay_Distribution: Exponential(0.001)*

- ***Ancho de banda de transmisión***

  La velocidad de transmisión de un paquete es de aproximadamente tres segundos.
  - *NetLinkDelay_Radio_MHtoMSS: ExponentialDistribution(0.0005)*
  - *NetLinkDelay_Radio_MSStoMH: ExponentialDistribution(0.001)*
  - *NetLinkDelay_Wire: ExponentialDistribution(0.005)*

- ***Otros valores***

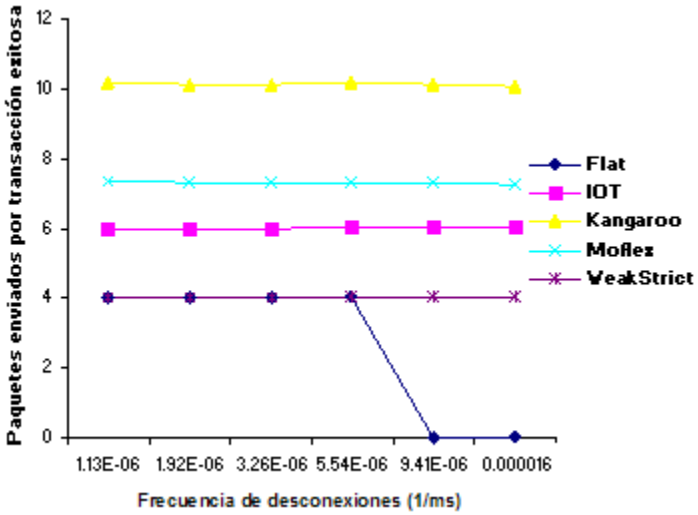
  La probabilidad de bloqueo es nula ya que sólo son consultas y almacenamiento de la información recibida en cada toma.
  - *Probabilidad de lock: 0*

La variación considerada es sobre la *frecuencia de desconexión*, produciéndose una cada minuto cuando la camioneta se encuentra cerca del tornado hasta una cada 15 minutos cuando se aleja del mismo.

Análisis de los resultados obtenidos

A continuación mostramos las mediciones obtenidas y su correspondiente análisis:.

Eficiencia desde el punto de vista de la utilización de recursos



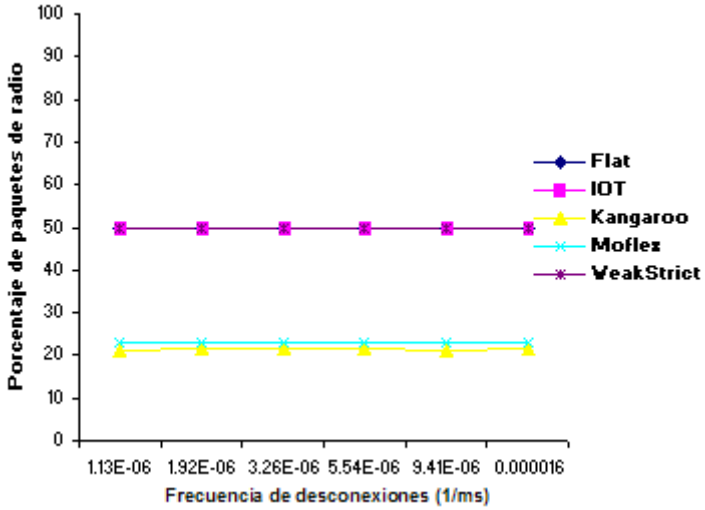
En el caso de Moflex y Kangaroo, se puede observar que como ambos son inmunes a las desconexiones, no se observan cambios al variar la frecuencia de dicho evento, permaneciendo estable el número de paquetes enviados por transacción exitosa. El número no es constante pues ambos modelos poseen subtransacciones, que implican la presencia de más paquetes.

Como Flat es sensible a las desconexiones, se observa que a mayor frecuencia, no sobrevive ninguna transacción.

WeakStrict también es sensible a los eventos de desconexión siempre y cuando la transacción sea *strict*. Como tuvo suficientes transacciones del tipo *weak* (que resuelven las desconexiones con éxito) a pesar de las desconexiones, el número de paquetes se mantiene constante.

La diferencia entre 4 o 6 paquetes por transacción exitosa (en los casos de IOT, Flat y WeakStrict) se debe a la presencia o ausencia de subtransacciones, cuya administración implica la presencia de más paquetes en la red simulada.

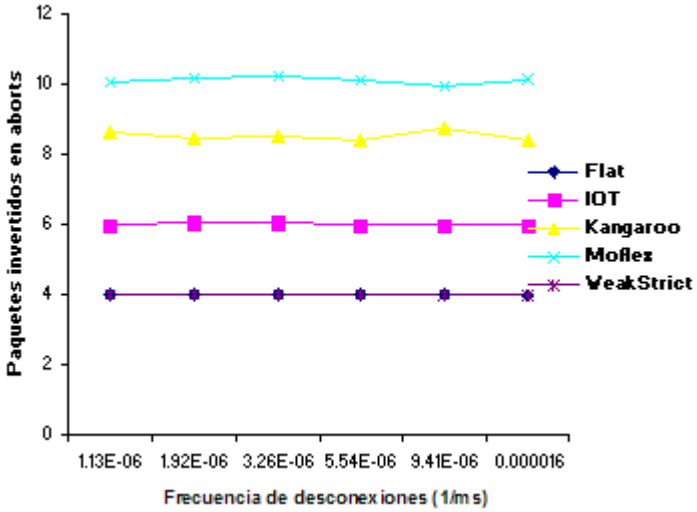
Carga de trabajo asumida por el *MH*



El porcentaje de paquetes de radio se mantiene constante para Flat, IOT y WeakStrict, dado que todo el trabajo lo asume el *MH* y cada paquete viaja un tramo por la red fija y otro por radio.

En cambio para Moflex y Kangaroo, por implementar sus transacciones en el *MSS*, se observa una menor carga en el *MH*.

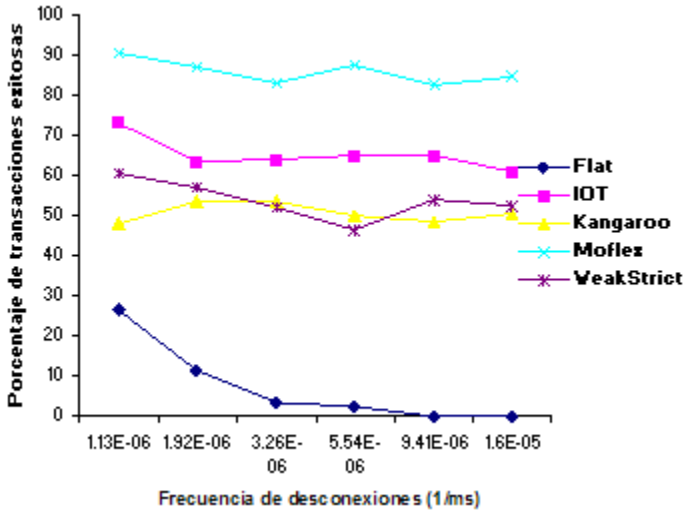
Costo del fracaso desde el punto de vista de la utilización de recursos



La cantidad de paquetes invertidos en aborts se mantiene constante para los modelos que no tienen definidas subtransacciones (Flat, IOT y WeakStrict) ya que la frecuencia de desconexión no influye en la cantidad de paquetes necesarios para resolver el *abort*.

En el caso de Moflex y Kangaroo, puede verse que Moflex gasta más paquetes en transacciones que finalmente terminan abortadas, dado que puede continuar una transacción a pesar de que alguna de sus subtransacciones aborte. Kangaroo, en cambio, no continúa en esa situación, por lo que invierte menos paquetes.

Robustez del modelo



Claramente se puede observar la precariedad de Flat ya que, a medida que aumenta la frecuencia de desconexiones, no sobrevive ninguna de sus transacciones.

Tanto Moflex como Kangaroo son inmunes al eventos de desconexión, sin embargo se observa una mejor performance para Moflex ya que Kangaroo hace fracasar toda la transacción ante la primer subtransacción que aborta.

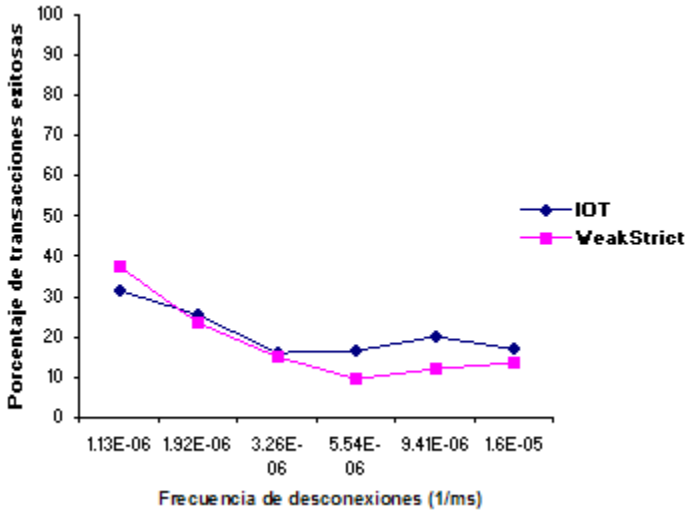
El éxito de Moflex se debe a que las subtransacciones exitosas (que representan el 80% del total) tienen tres oportunidades para causar la finalización exitosa de la transacción principal (lo hacen el 60% de las veces):
- 1ra vez: de 100 subtransacciones, 80 son exitosas y el 60% de ellas (48) causan el éxito.
- 2da vez: de las 52 restantes, 42 son exitosas y el 60% de ellas (25) causan el éxito.
- 3ra vez: de las 27 restantes, 22 son exitosas y el 60% de ellas (13) causan el éxito.

Entonces, 86% de la veces (48+25+13) la transacción principal termina exitosamente, como se puede observar en el gráfico.

En el caso de IOT, a medida que más de sus transacciones sufren una desconexión, interviene en mayor proporción el efecto de la sincronización (que puede hacer abortar una transacción que localmente había terminado exitosamente), por lo que se obtiene alrededor del 64% de éxitos (80% de probabilidad de éxito de un transacción y sobre éstas, un 80% sobrevive a la sincronización)

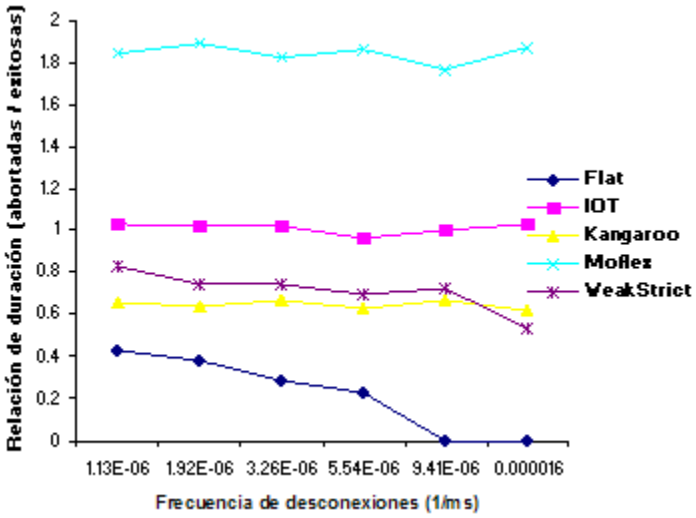
En WeakStrict se observa una buena performance ya que el modelo fue instanciado con un 80% de probabilidad de que una transacción pueda trabajar con un cluster local, sobre éstas, el 80% tienen probabilidad de éxito, sobreviviendo el 80% de ellas a la sincronización con la base de datos central.

Observamos entonces, que el éxito de IOT y WeakStrict depende en gran medida de la capacidad de la aplicación para trabajar en forma local. Para verificar esto, variamos para IOT el parámetro que define el éxito de la sincronización (bajando de 80% a 20%), y para WeakStrict el parámetro que determina si la transacción es local (bajando del 80% al 20%).



Podemos ver en el gráfico que, en efecto, ambos modelos pierden robustez en este caso, comprobando la anterior afirmación.

Costo del fracaso desde el punto de vista del usuario



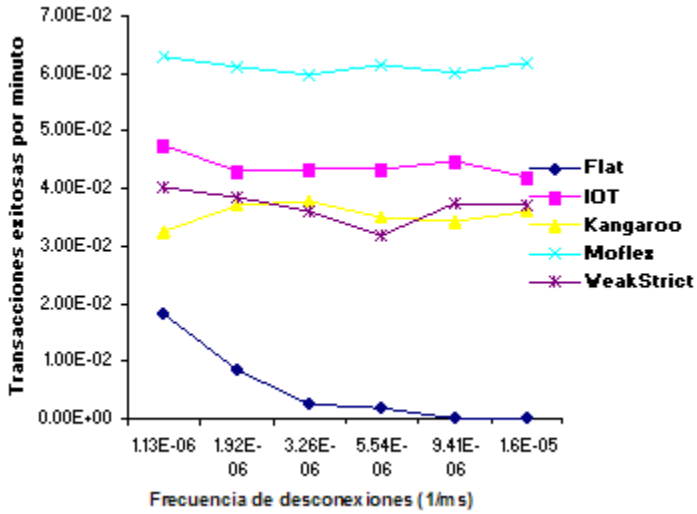
Flat refleja su gran sensibilidad ante las desconexiones, a medida que aumenta su frecuencia, no sobrevive ninguna de sus transacciones.

Kangaroo tiene un bajo costo de fracaso ya que finaliza ante el primer *abort*. En cambio Moflex refleja un mayor costo ya que agota todas las instancias de medición.

Para IOT, la duración de las transacciones abortadas es similar a las transacciones exitosas y como su duración tampoco es afectada por el evento de desconexión, el costo de fracaso es aproximadamente igual al de los éxitos.

Para WeakStrict, la duración de las transacciones abortadas disminuye a medida que aumenta la frecuencia de desconexión ya que en esos casos, la primera desconexión sobreviene más rápidamente. Esto es aplicable sólo a las transacciones de tipo *strict* (20% de transacciones). El resto, las de tipo *weak*, no se ven afectadas por el evento de desconexión, por lo cual el efecto del evento de desconexión no es tan pronunciado como en el caso de Flat.

Eficiencia desde el punto de vista del usuario



Se observa que Moflex produce una cantidad estable de transacciones exitosas por minuto, al igual que WeakStrict y Kangaroo: la duración de las transacciones no se ve afectada por las desconexiones ya que el tiempo que permanecen desconectados es pequeño (1 segundo).

IOT disminuye su eficiencia a medida que aumentan las desconexiones y sus transacciones deben por ello sincronizarse con la base de datos central.

Flat refleja su baja performance por su sensibilidad ante el evento de desconexión..

6.3.3 Edición de un libro en colaboración

En esta aplicación, un grupo de tesistas trabajan en un documento de tesis en formato Word. Cada uno toma a su cargo la redacción de un conjunto de capítulos distinto, aunque a veces necesita actualizar referencias cruzadas en otros capítulos. Cada tesista utiliza una plataforma móvil para realizar el trabajo. Según el tiempo que pueden dedicarle al documento, puede suceder que en un momento dado uno sólo de ellos esté editando el documento, o bien que varios o incluso todos ellos estén editando el documento al mismo tiempo.

El objetivo de esta aplicación es someter los modelos a un escenario con distintas cantidades de *MH*s trabajando simultáneamente.

Instanciación en el simulador

Las principales variables que se instanciaron en el simulador son las siguientes:

- ***Probabilidad de desconexión***
  Se produce una desconexión cada media hora.
  - *NetDisconnect_Distribution: ExponentialDistribution (0.0000005)*

- ***Tiempo medio de desconexión***
  Se reconecta en un minuto
  - *NetReconnectDelay: ExponentialDistribution 0.00002)*

- ***Duración de la transacción principal***
  La duración es de aproximadamente 2 horas, correspondiente a una sesión de trabajo.
  - *DBMS_AccessesPerTx: GaussianDistribution(10,2)*
  - *DBMS_TimeBetweenAccesses: ConstantDistribution(750000)*
  - *DBMS_AccessesProcessingTime: ConstantDistribution(5000)*
  - *NetLinkDelay_Radio_MHtoMSS: ExponentialDistribution(0.0005)*
  - *NetLinkDelay_Radio_MSStoMH: ExponentialDistribution(0.001)*
  - *NetLinkDelay_Radio_Symetrical: false*

- ***Probabilidad de éxito de las transacciones***
  La probabilidad de éxito de cada transacción es alta ya que no hay razones para suponer que la grabación del documento pueda fallar.
  - *DBMS_txSuccess: BernoulliDistribution(0.95)*

- ***Probabilidad de salto de celda***
  Aproximadamente un salto de celda cada media hora.
  - *NetHandoff_Distribution: ConstantDistribution(0.0000005)*

- ***Cantidad de transacciones lanzadas***
  Se considera una sola transacción por *MH*, que representa una sesión de trabajo de aproximadamente 2 horas. En los modelos que soportan subtransacciones, hay una subtransacción por transacción principal, ya que la edición de un documento Word no es divisible en subtransacciones. Asumimos que el MH dispone de memoria suficiente para almacenar una unidad de edición.

- *MH_TransactionsPerMH: Gaussian(1,0)*

- ***Cantidad de MHs simultáneos***

  Cada aparición de un *MH* modela una sesión de trabajo de un tesista en el documento. Se modelan 50 sesiones de trabajo y por lo tanto igual número de *MH*s. Las sesiones se inician en un intervalo variable que va desde una cada media hora hasta una cada hora y media, por lo que la cantidad de sesiones simultáneas variará aproximadamente entre 1 y 4.
  - *NetNewMH_Distribution: Gaussian(50,0)*
  - *NetNewMH_Delay_Distribution: Exponential(0.0000005;5;1s/1800000+1s/)*

- ***Ancho de banda de transmisión***

  La velocidad de transmisión de un paquete es de aproximadamente tres segundos.
  - *NetLinkDelay_Radio_MHtoMSS: Exponential(0.0005)*
  - *NetLinkDelay_Radio_MSStoMH: Exponential(0.001)*
  - *NetLinkDelay_Wire: Exponential(0.005)*

- ***Otros valores***

  La probabilidad de bloqueo es 1 ya que Word no permite que más de una persona por vez edite un documento.
  - *Probabilidad de lock: 1*

  Consideramos que una transacción aborta si está bloqueada por más de media hora.
  - *Timeout: 1800000*

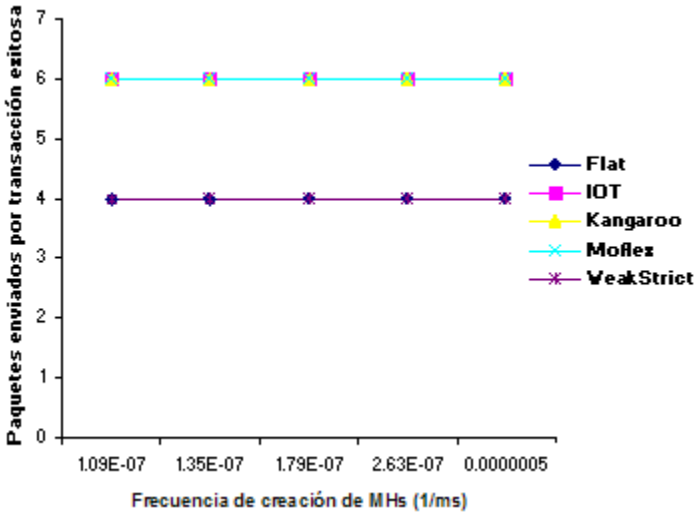
  Consideramos que los cambios realizados sobre partes comunes del documento se modelan en WeakStrict con transacciones *Strict*, mientras que los cambios dentro de los capítulos asignados se realizan con transacciones *Weak*.
  Almacenar en forma local los capítulos que edita el tesista tiene una probabilidad del 80%, el restante 20% puede corresponder a partes del documento comunes o ajenas.
  La probabilidad de éxito de la sincronización en las transacciones que corren en el *MH* difiere entre los modelos IOT y WeakStrict ya que en IOT , a diferencia de WeakStrict , los cambios sobre secciones comunes del documento se realizan localmente, por lo que al sincronizar es mayor la probabilidad de fracaso.
  - *WeakStrict_IsLocal: BernoulliDistribution(0.8)*
  - *IOT_Commit_Abort: BernoulliDistribution(0.72)*
  - *DBMS_SynchSuccess:BernoulliDistribution(0.9)*

La variación considerada es sobre la frecuencia de creación de *MH*s. Cada aparición de un *MH* modela una sesión de trabajo de un tesista en el documento.

Análisis de los resultados obtenidos

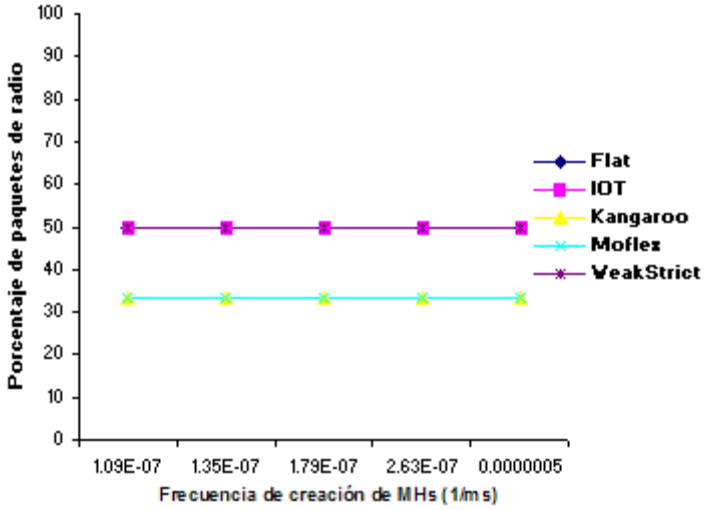
A continuación mostramos las mediciones obtenidas y su correspondiente análisis:

Eficiencia desde el punto de vista de la utilización de recursos



Como todos los modelos presentaron por lo menos alguna transacción exitosa, la diferencia que se observa entre 4 o 6 paquetes se debe a la presencia o ausencia de subtransacciones, cuya administración implica la presencia de más paquetes en la red simulada.

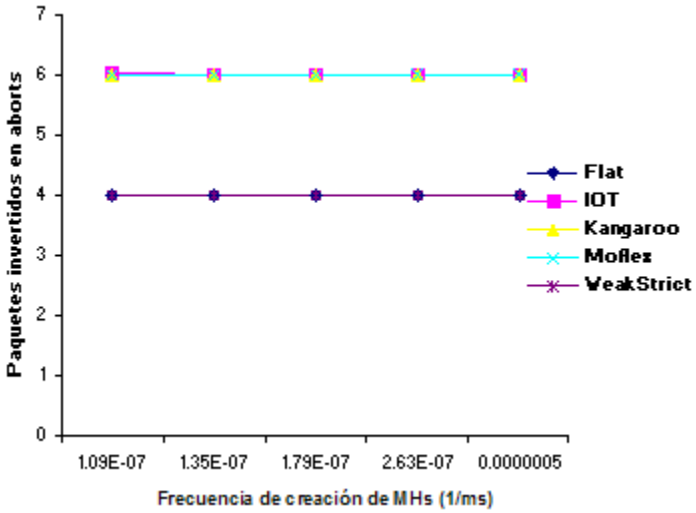
Carga de trabajo asumida por el *MH*



El porcentaje de paquetes de radio se mantiene constante para Flat, IOT y WeakStrict, dado que todo el trabajo lo asume el *MH* y cada paquete viaja un tramo por la red fija y otro por radio.

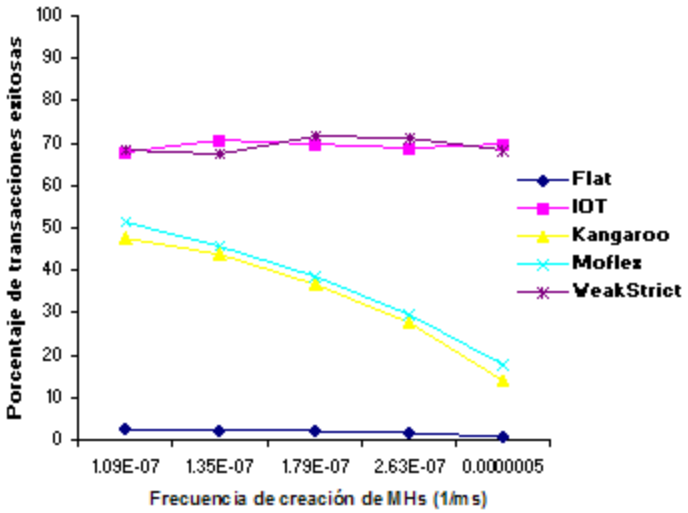
En esta aplicación también se observa una menor carga en el *MH* para Moflex y Kangaroo por implementar sus transacciones en el *MSS*. En este caso, se observa una cantidad constante de paquetes debido a que se modeló una sola subtransacción por transacción principal.

Costo del fracaso desde el punto de vista de la utilización de recursos



Se puede observar que la cantidad de paquetes invertidos en aborts es la misma que la cantidad de paquetes invertidos en éxitos para todos los modelos. Para aquellos que no implementan subtransacciones (Flat, IOT y WeakStrict) es el resultado habitual pero en este caso también lo cumplen Moflex y Kangaroo ya que se modeló una sola subtransacción por cada transacción principal.

Robustez del modelo



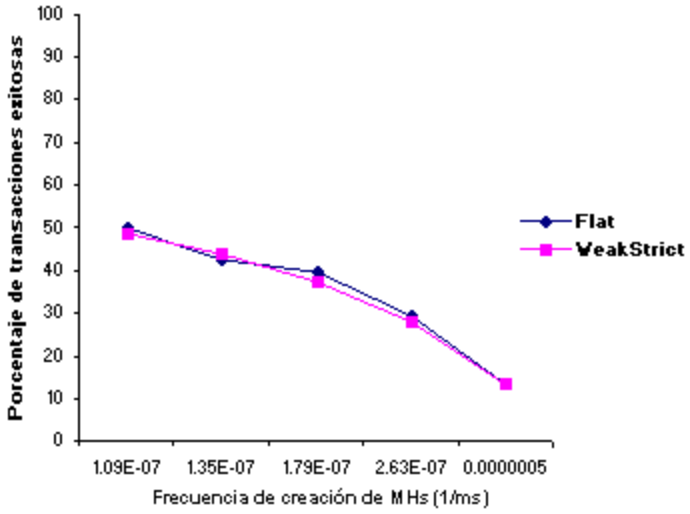
El gráfico muestra que los modelos que mejor implementan una solución para el problema son, como se esperaba, IOT y WeakStrict. En ambos casos el porcentaje de transacciones exitosas se mantiene constante alrededor del 70%; para WeakStrict debido a que el 80% de sus transacciones son *weak*, de éstas el 95% son exitosas y cuando se sincronizan, el 90% de éstas finalizan exitosamente. Para IOT, la sincronización es exitosa en un 72% de los casos, sobre un 95% de transacciones exitosas.

En realidad, a mayor frecuencia de creación de *MH*s, debería observarse una pequeña disminución de la robustez, pero como nuestro simulador modela la probabilidad de sincronización independiente de la superposición de transacciones, esta disminución no se ve reflejada.

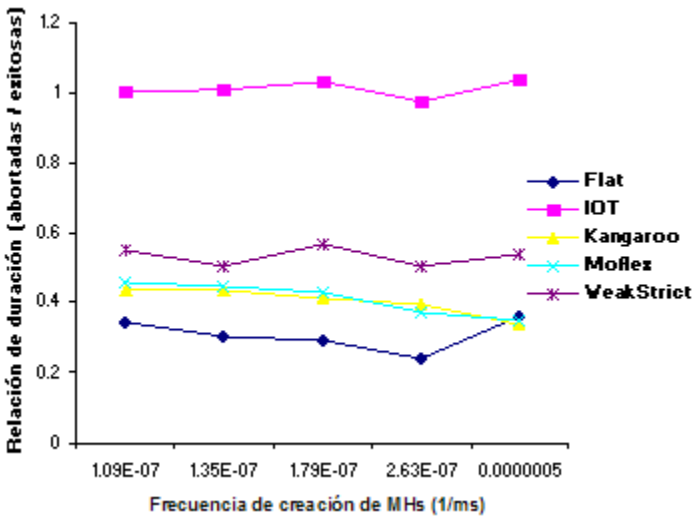
Los modelos Moflex y Kangaroo muestran que, a mayor frecuencia de bloqueo por superposición de transacciones, pierden robustez. Esto se debe a que sus transacciones trabajan sobre el mismo servidor global,

donde la probabilidad de bloqueo se incrementa al aumentar la frecuencia en la creación de *MH*s y, por consiguiente, las transacciones abortan por *timeout*.

En el caso de Flat, se observa una baja performance ya que no soporta desconexiones ni saltos de celdas. Como son transacciones de larga vida, es alta la probabilidad de que alguno de estos eventos las alcancen, generando aborts. Lo mismo debe suceder para las transacciones tipo strict de WeakStrict. Sin embargo, como el 80% de las transacciones son de tipo weak, este efecto pasa desapercibido en el gráfico precedente. Para verificar el impacto de la concurrencia en Flat y en las transacciones tipo *strict* de WeakStrict, repetimos la corrida eliminando los eventos de salto de celda y desconexión y forzamos todas las transacciones de tipo *strict*. Como puede observarse en el siguiente gráfico, ambos modelos soportan concurrencia, aunque, al igual que Moflex y Kangaroo, a mayor probabilidad de bloqueo por superposición de transacciones, pierden robustez:



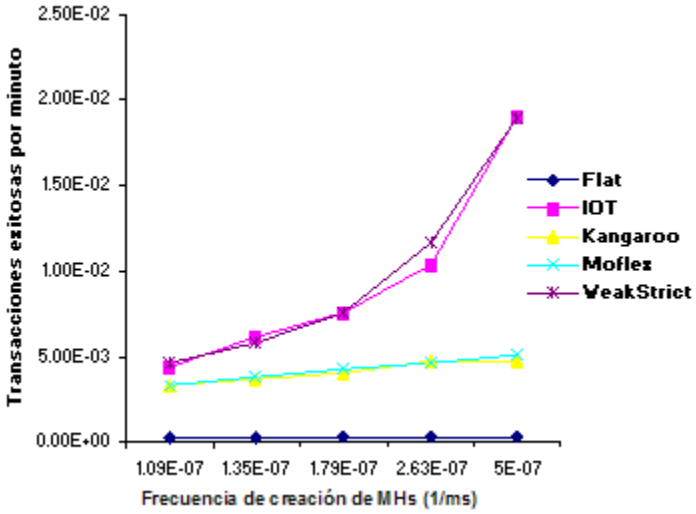
Costo del fracaso desde el punto de vista del usuario



Para IOT, la duración de las transacciones abortadas es similar a las transacciones exitosas y como su duración tampoco es afectada por la frecuencia de creación de *MH*s, el costo de fracaso es aproximadamente igual al de los éxitos.

Para Kangaroo y Moflex el costo del fracaso tiene un leve aumento a medida que disminuye el bloqueo por superposición de transacciones, ya que cuando aumenta la cantidad de transacciones simultáneas, las transacciones que abortan por *timeout* debido al bloqueo son más cortas (el evento de bloqueo sucede antes).

Eficiencia desde el punto de vista del usuario



Se observa que IOT y WeakStrict aumentan su cantidad de transacciones exitosas por minuto a medida que se incrementa la frecuencia en la creación de *MH*s ya que al no variar su robustez, se contabilizan más transacciones ante la mayor existencia de *MH*s.

En Kangaroo y Moflex, se puede observar una disminución en la performance cuando es alta la incidencia de la superposición de transacciones porque baja notoriamente el porcentaje de transacciones exitosas que presenta.

En el caso de Flat, su performance es siempre baja ya que su porcentaje de transacciones exitosas es constantemente bajo.

6.3.4 Sonda espacial

En esta aplicación, una sonda espacial recibe instrucciones y transmite datos científicos mientras es atendida por diferentes estaciones de seguimiento. Dicha sonda se ubica en una orbita no geoestacionaria por lo que la estación terrena controladora variará conforme a la rotación de la Tierra. Se modela como un sólo *MH* que genera muchas transacciones de larga duración (una sola por vez).

El objetivo es ver cómo se comportan los modelos ante distintas velocidades de transmisión y de duración de transacciones.

Instanciación en el simulador

Las principales variables que se instanciaron en el simulador son las siguientes:

- ***Probabilidad de desconexión***
  Suponemos que la sonda está generalmente al alcance de una estación de seguimiento y que no sufre desconexiones más que una vez por día.
  - *NetDisconnect_Distribution: ExponentialDistribution(0.00000001)*

- ***Tiempo medio de desconexión***
  Las desconexiones son muy cortas, causadas por ruido atmosférico, ya que otras causas de desconexión causarían la pérdida de la sonda (por ejemplo desorientación de la antena de la sonda o pérdida de potencia de alimentación) y en este trabajo asumimos un funcionamiento nominal de los sistemas.
  - *NetReconnectDelay: ExponentialDistribution(0.02)*

- ***Duración de la transacción principal***

  Varía entre tres segundos y 16 minutos, según la distancia entre la sonda y la Tierra.

  - *DBMS_AccessesPerTx: GaussianDistribution(1,0)*
  - *DBMS_TimeBetweenAccesses: ConstantDistribution(1000)*
  - *DBMS_AccessesProcessingTime: ConstantDistribution(1000)*
  - *NetLinkDelay_Radio_MHtoMSS: ExponentialDistribution(0.000001;6;10*)*
  - *NetLinkDelay_Wire: ExponentialDistribution(0.003)*
  - *NetLinkDelay_Radio_Symetrical: true*

- ***Probabilidad de éxito de las transacciones***

  La probabilidad de éxito de una transacción es alta.

  - *DBMS_txSuccess: BernoulliDistribution(0.8)*

- ***Probabilidad de salto de celda***

  Estimamos seis saltos por día (salta a medida que la Tierra gira y la sonda va quedando fuera del campo de visión de la estación de seguimiento).

  - *NetHandoff_Distribution: ConstantDistribution(0.00000007)*

- ***Cantidad de transacciones lanzadas***

  Simulamos una gran cantidad de transacciones (aproximadamente 50) sin bloqueo y con una sola subtransacción, por tratarse de transmisiones de imágenes.

  - *MH_TransactionsPerMH: GaussianDistribution(50,10)*

- ***Cantidad de MHs simultáneos***

  Consideramos una sola sonda (*MH*)

  - *NetNewMH_Distribution: GaussianDistribution(1,0)*
  - *NetNewMH_Delay_Distribution: Exponential(0.001)*

- ***Ancho de banda de transmisión***

  La variación considerada es sobre la demora de la transmisión inalámbrica, en un rango que va desde los 16 minutos hasta los 10 milisegundos. La demora de la transmisión por la red fija es de 300 milisegundos

  - *NetLinkDelay_Radio_MHtoMSS: ExponentialDistribution(0.000001;6;10*)*
  - *NetLinkDelay_Radio_MSStoMH: igual a la anterior*
  - *NetLinkDelay_Wire: ExponentialDistribution(0.003)*

- ***Otros valores***

  La probabilidad de bloqueo es baja por tratarse de transmisiones de imágenes.

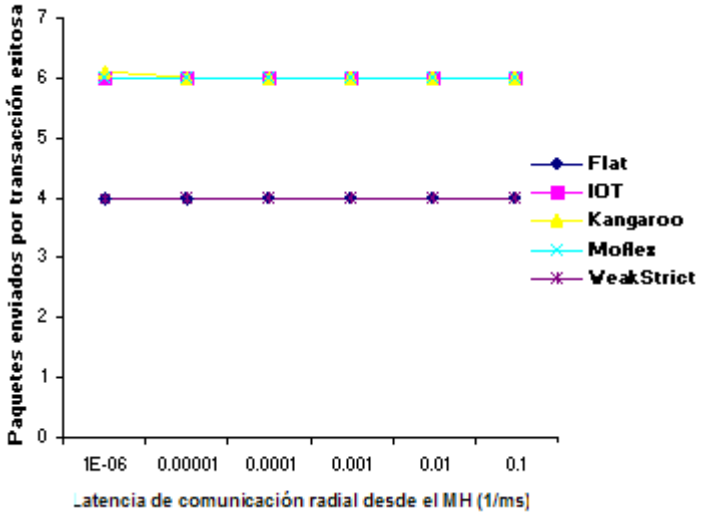
  - *Probabilidad de lock: 0*

La variación considerada es sobre el *delay* de la transmisión inalámbrica, en un rango que va desde los 16 minutos hasta los 10 milisegundos. El *delay* de la transmisión por la red fija es de 300 milisegundos.

Análisis de los resultados obtenidos

A continuación mostramos las mediciones obtenidas y su correspondiente análisis:

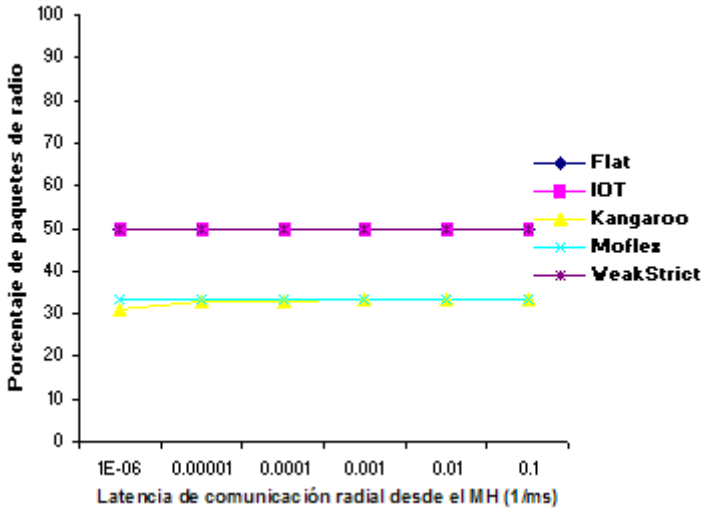
Eficiencia desde el punto de vista de la utilización de recursos



La cantidad de paquetes enviados por transacción exitosa no varía al aumentar la latencia de comunicación radial, ya que ese factor no influye en la longitud ni en la topología de la ruta seguida por los paquetes. Además, la implementación de la aplicación es simple, tenemos para cada transacción una sola subtransacción.

La diferencia entre 4 o 6 paquetes por transacción exitosa se debe a la presencia o ausencia de subtransacciones, cuya administración implica la presencia de más paquetes en la red simulada.

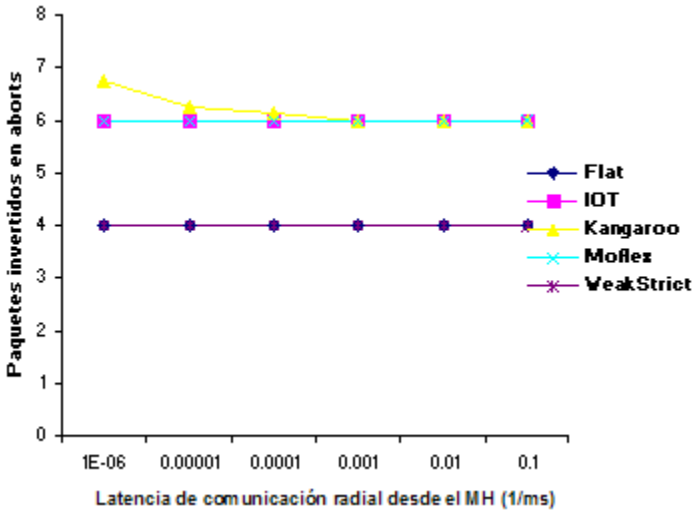
Carga de trabajo asumida por el *MH*



El porcentaje de paquetes de radio se mantiene constante al variar la latencia de comunicación radial por la misma causa que en el gráfico de paquetes enviados por transacción exitosa. La única diferencia se ve en IOT cuyo paquete extra de administración de la subtransacción viaja tanto por la red fija como por la red inalámbrica.

En Kangaroo se ve una pequeña variación al registrarse una mayor latencia de comunicación radial porque al producirse un salto de celda durante la transacción, el paquete que informa el resultado de la misma debe viajar desde el *MSS* que la originó al *MSS* nuevo.

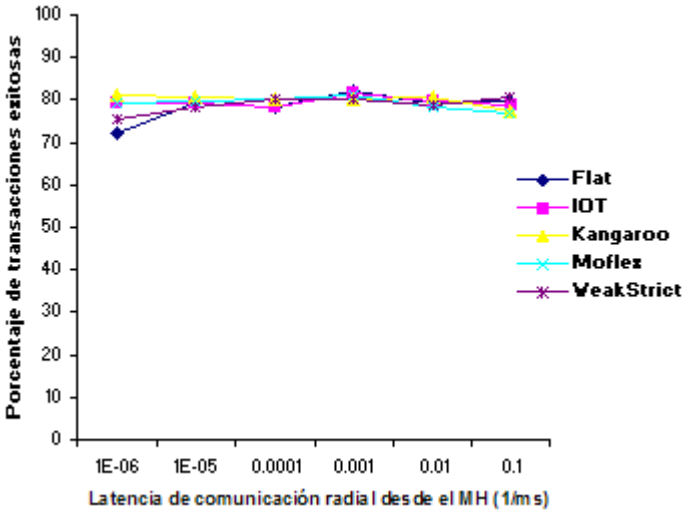
Costo del fracaso desde el punto de vista de la utilización de recursos



La cantidad de paquetes invertidos en aborts se mantiene constante para los modelos que no tienen definidas subtransacciones (Flat, IOT y WeakStrict) ya que la latencia de comunicaciones no influye en la cantidad de paquetes necesarios para resolver el *abort*.

En el caso de Moflex y Kangaroo, puede verse que Kangaroo utiliza más paquetes a mayor latencia de comunicación radial por el salto de celda que se produce y el consiguiente paquete que viaja desde el *MSS* original al nuevo. Si bien Moflex es un modelo que implementa subtransacciones, en esta aplicación sólo se instancia una subtransacción y por esa causa, su costo de fracaso es constante.

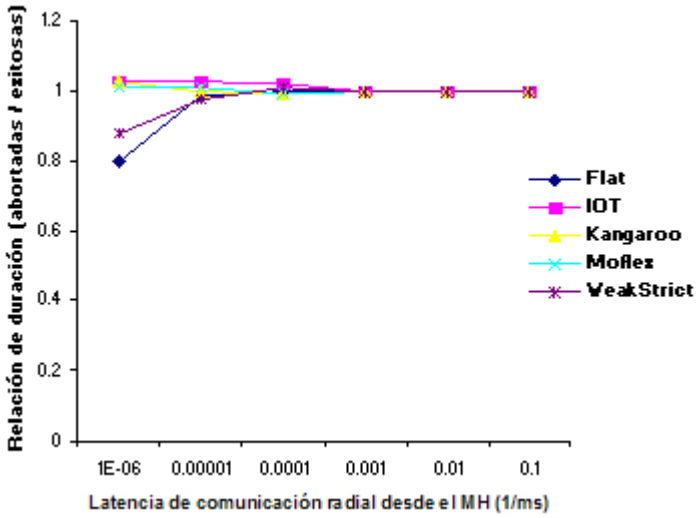
Robustez del modelo



El porcentaje de transacciones exitosas se mantiene aproximadamente constante para todos los modelos debido a que la latencia de comunicaciones no influye en esta medición y a que la esperanza de éxito de una transacción fue determinada en 0.8 con una pequeña aleatoriedad como se observa en el gráfico.

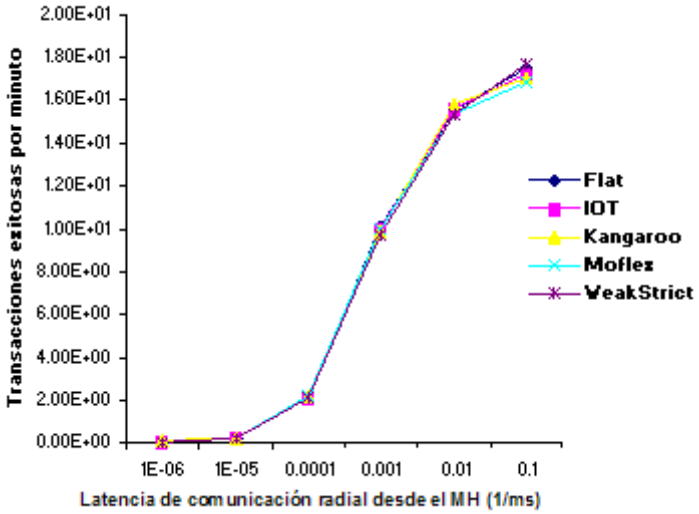
Sin embargo, a mayor latencia puede observarse una disminución en la robustez para los modelos que no soportan el salto de celda, es decir Flat y WeakStrict; en este último caso, se debe a que el 100% de las transacciones son de tipo *strict*.

Costo del fracaso desde el punto de vista del usuario



Dada la simplicidad de las condiciones de la aplicación (una sola subtransacción, pocos eventos de salto de celda y desconexiones), el proceso para abortar una transacción es similar al de completarla con éxito, es decir que no es más costosa una transacción exitosa que una abortada, excepto para Flat y WeakStrict que son sensibles al salto de celda que se presenta en mayor medida cuanto mayor es la latencia de comunicación.

Eficiencia desde el punto de vista del usuario



Se observa que a medida que la latencia de comunicación disminuye, aumenta la cantidad de transacciones exitosas por minuto. En el gráfico se pueden ver dos puntos de interés: uno cuando la comunicación de ida y vuelta tarda aprox. 20 segundos y el otro cuando la misma tarda aproximadamente un segundo. En el primer segmento, donde la comunicación es más lenta, cada transacción tarda más de un minuto, por lo que muy pocas transacciones se completan exitosamente en un minuto dado. En el tercer segmento, donde la velocidad de comunicación cae por debajo de un segundo, ésta deja de ser el factor

preponderante en la determinación de la duración de una transacción, cobrando mayor importancia factores como el tiempo entre accesos a la base y el tiempo de procesamiento de cada uno, por lo que la curva se estabiliza.

Capítulo 7: Conclusiones y Trabajo Futuro

El avance tecnológico de los últimos años en el campo de las comunicaciones y la computación móvil ha posibilitado la aparición de un nuevo campo de aplicación para los sistemas transaccionales. Este nuevo contexto presenta dificultades a las aplicaciones transaccionales debido a las desconexiones frecuentes, la movilidad y las limitaciones de capacidad de los dispositivos móviles.

Ante estas dificultades, se han presentado diversos modelos transaccionales móviles de los cuales no existen implementaciones ni estudios comparativos experimentales. Para efectuar este estudio, desarrollamos un simulador con el cual instanciamos y comparamos algunos modelos.

Todos los modelos móviles estudiados relajan las propiedades ACID de las transacciones [Cor2000]. Si bien es un aspecto interesante para medir experimentalmente, hacerlo hubiese implicado modelar las lecturas y escrituras de cada transacción a la base de datos y por ende, también los datos y los registros de la misma. Modelar estos aspectos de la realidad implicaría una importante pérdida de generalidad y de facilidad de uso del simulador, ya que para comparar modelos sería necesario "programar" cada transacción en forma individual. Por lo tanto, decidimos priorizar otros aspectos y no medir el cumplimiento de las propiedades ACID.

En este capítulo mencionamos a qué conclusiones hemos arribado luego de analizar los modelos móviles en el simulador implementado. Previamente identificamos las cualidades de cada modelo bajo diferentes situaciones y contrastamos su comportamiento teórico con el observado. Finalizamos enunciando algunos puntos de interés para ser estudiados como trabajo futuro.

7.1 Comparación entre modelos

En este trabajo sometimos a cuatro modelos móviles (IOT, Kangaroo, Moflex, WeakStrict) y uno no móvil (Flat) a diversas situaciones elegidas para poner de manifiesto las principales características de los ambientes móviles (saltos de celda, desconexiones, transacciones de larga duración y alto grado de concurrencia) y observamos el comportamiento de cada modelo.

7.1.1 Saltos de celda

Moflex y Kangaroo están explícitamente diseñados para soportar los saltos de celda, mientras que IOT es ajeno a este evento. Interpretamos que para Flat, IOT y WeakStrict, dado que no especifican un comportamiento para el salto de celda, este evento es equivalente a una desconexión. En el caso de Flat, ésto implica el fracaso de la transacción. Para IOT, al finalizar la transacción deberá sincronizarse con el servidor, mientras que para WeakStrict la desconexión aborta las transacciones de tipo *strict* pero no las de tipo *weak*.

Pudimos observar que Moflex se destaca frente a los demás modelos, mientras que IOT se desenvuelve aceptablemente. La aplicación elegida para evaluar movilidad era la propuesta en el paper de Moflex, por lo que era de esperar que su actuación fuera destacada. Si se quisiera implementar dicha aplicación con otro modelo, se lo debería rodear de la lógica necesaria para lograr la misma o similar funcionalidad que Moflex. Este modelo soporta los saltos de celda explícitamente; además, según la semántica de la aplicación, el fracaso de una subtransacción no implica el fracaso de la aplicación.

El desempeño aceptable de IOT se debe a que decidimos implementar el salto de celda como una desconexión y el modelo está diseñado para soportar desconexiones. El comportamiento observado de WeakStrict y de Flat se corresponde con el teórico: Flat simplemente no soporta saltos de celda, mientras que WeakStrict sólo los soporta para sus transacciones de tipo *weak*. En otras palabras, WeakStrict emula a IOT para las transacciones de tipo *weak* y a Flat para las transacciones de tipo *strict*.

Kangaroo teóricamente debería funcionar bien, pero nuestras observaciones se vieron fuertemente afectadas por su sensibilidad al fracaso de las subtransacciones, obteniendo resultados relativamente aceptables cuando las subtransacciones fueron pocas y tenían una alta probabilidad de terminar exitosamente. Según pudimos observar, Kangaroo es inadecuado para aplicaciones con muchas subtransacciones (ya que aumenta la exposición al fracaso) y en las que algunas subtransacciones pueden fallar sin que ello deba significar el fracaso de toda la transacción. Sin embargo, para la mayoría de las aplicaciones tradicionales es razonable que la transacción fracase si alguna de sus partes falla, por lo que Kangaroo se revela como un buen modelo móvil para aplicaciones tradicionales.

7.1.2 Desconexiones

Al completar los aspectos subespecificados de los modelos analizados, determinamos que deben distinguirse dos niveles de soporte de la desconexión: el primer nivel de soporte implica que las transacciones se ejecutan en un nodo de la red fija, por ejemplo un *MSS* y que una desconexión del *MH* no implica el fracaso automático de las transacciones, aunque impide al usuario trabajar interactivamente con la base de datos mientras dura la desconexión. El segundo nivel de soporte indica que las transacciones se ejecutan en el *MH* contra una base de datos local, con lo que una desconexión no impide al usuario seguir trabajando en forma interactiva. En este trabajo consideramos únicamente el primer nivel de desconexión y los resultados experimentales reflejan esta elección.

Ante la presencia de desconexiones, los papers que presentan los modelos Moflex y Kangaroo no definen una respuesta específica; sin embargo, dado que ambos modelos trasladan a los *MSS* la responsabilidad de administrar las transacciones, interpretamos que, mientras el usuario no necesite una respuesta interactiva del sistema durante el procesamiento de la transacción y la aplicación en el MH no requiera nuevas subtransacciones, el momento crítico ante una desconexión es la comunicación del resultado final de la transacción a la aplicación que reside en el *MH*. La definición de IOT incluye una copia local de la base datos que se mantiene actualizada incluso estando conectado, por lo que las desconexiones están explícitamente soportadas. En el caso de WeakStrict, la literatura define que las transacciones de tipo *weak*, que corren localmente, soportan desconexión, mientras que las de tipo *strict* o globales abortan frente a este evento. Por lo tanto, el éxito de una aplicación implementada con este modelo depende de la proporción de transacciones *weak* que utilice.

Los resultados obtenidos con el uso del simulador indican que, tal como lo esperábamos, Kangaroo, Moflex, IOT y WeakStrict soportan razonablemente bien las desconexiones. La eficacia de cada modelo depende de la aplicación que lo usa. En particular, si la aplicación puede trabajar localmente, es particularmente apta para WeakStrict y para IOT.

7.1.3 Transacciones de larga duración

Las transacciones de larga duración son un factor importante en los modelos móviles porque pueden retener la reserva de recursos compartidos por largo tiempo, disminuyendo el grado de concurrencia posible. Podemos distinguir tres escenarios que producen este efecto: transacciones largas por naturaleza (por ejemplo, actualizaciones masivas), períodos prolongados de desconexión y bajo ancho de banda. Decidimos someter a los modelos a la tercera opción porque, en primer lugar, la primera no es una aplicación común móvil y, en segundo lugar, las últimas dos son indistinguibles desde el punto de vista de los modelos móviles.

En este contexto, ninguno de los modelos debería tener problemas para completar las transacciones exitosamente. Sin embargo, debido a que todos están sometidos a la misma limitación física (la velocidad de propagación de los paquetes), no se espera que ninguno de ellos destaque sobre los demás. El único aspecto en el que pueden diferenciarse es el tiempo que retienen bloqueos en la base, dado que aquellos modelos que delegan el procesamiento de las transacciones en el *DBS* pueden completar la ejecución en menor tiempo y así liberar los bloqueos.

Sin embargo, esta diferencia no puede observarse en las corridas por la forma en que implementamos el cálculo de la duración de las subtransacciones: el tiempo de propagación por la red inalámbrica es incluido en el cálculo de la duración de la transacción principal; la longitud de las subtransacciones se calcula dividiendo la duración de la principal por la cantidad de subtransacciones, sin importar en qué plataforma se ejecuten. Esta decisión de diseño nos impide mostrar el efecto mencionado. Parte del trabajo futuro sería tomar en cuenta en el cálculo la plataforma en la que se ejecuta la transacción.

Pudimos observar que el modelo Kangaroo es menos eficiente a la hora de abortar una transacción de larga duración. En un entorno móvil la probabilidad de salto de celda aumenta con el tiempo, por lo que las transacciones largas se ven afectadas en mayor medida por este evento que las transacciones cortas. En un escenario con muchos saltos de celda, Kangaroo tiene un costo mayor para abortar, debido a que debe compensar las transacciones completadas exitosamente en los *MSS* por los que pasó.

7.1.4 Alto grado de concurrencia

A medida que aumenta la longitud de las transacciones (por los motivos explicados en el punto anterior), crece la probabilidad de conflicto entre transacciones. Por lo tanto, aquellos modelos que eviten dichos conflictos utilizando bases de datos locales tendrán un mejor desempeño. Es el caso de IOT y las transacciones de tipo *weak* del modelo WeakStrict.

Nuestras observaciones confirman esta hipótesis. Podemos concluir que IOT y WeakStrict son los modelos más apropiados para aplicaciones que permiten trabajar localmente en el *MH*.

7.1.5 Contraste de los resultados con el framework de comparación

Los resultados obtenidos experimentalmente en aquellos modelos que implementamos coinciden en general con la taxonomía definida en [Cor2000] (ver Tabla 1) Sin embargo, cabe aclarar que dicha taxonomía se refiere a la definición teórica de los modelos, mientras que en este trabajo fue necesario llevar dichos modelos al plano de la implementación, completando aspectos no especificados. Como estos aspectos son justamente los que en la taxonomía figuran como no soportados, al proponer una implementación obtuvimos algunos resultados diferentes, que explicamos a continuación.

En cuanto al soporte de desconexión del modelo, Kangaroo y Moflex no permiten al usuario seguir interactuando con la base de datos en forma desconectada. Sin embargo las transacciones pueden seguir funcionando porque trabajan desde los MSSs, a diferencia del modelo Flat, que aborta la transacción en el momento de la desconexión. Por lo tanto, podemos distinguir dos niveles de soporte de desconexión: el primer nivel implica la supervivencia de las transacciones al evento de desconexión; el segundo nivel permite al usuario continuar interactuando con una copia local de la base de datos en forma desconectada. Según el nivel que se considere, se podrá decir que Kangaroo y Moflex dan o no soporte a este evento. En nuestro trabajo, dado que no modelamos la interacción del usuario con la aplicación, tomamos en cuenta el primer nivel de soporte de desconexión. En el *framework* definido en [Cor2000] se toma en cuenta el segundo nivel. En el caso de IOT y WeakStrict, ambos están diseñados para tolerar los dos niveles de desconexión. Esta clasificación se presenta en la Tabla 10.

Modelo Transaccional		Soporte de Desconexión Nivel 1	Soporte de Desconexión Nivel 2
Kangaroo		√	×
Moflex		√	×
IOT		√	√
Weak/Strict	weak	√	√
	strict	×	×
Flat		×	×

Tabla 10: Clasificación según nivel de soporte de desconexión.

Por otra parte, si bien es cierto que IOT y WeakStrict no definen un comportamiento explícito para el evento de salto de celda (como puede observarse en la taxonomía), al modelar este evento como una desconexión, y dado que estos modelos permiten trabajar sobre una copia local de la base de datos, dicho evento no necesariamente resulta en el fracaso de las transacciones.

7.1.6 Clasificación de los modelos según el comportamiento observado

Para complementar la clasificación teórica presentada por [Cor2000] y ampliada en el punto anterior, elaboramos una nueva clasificación (ver Tabla 11), similar a la anterior, pero basada en las observaciones del comportamiento de los modelos (principalmente la robustez) ante saltos de celda, desconexiones, transacciones de larga duración y concurrencia.

En esta clasificación definimos tres niveles de soporte: Si todas las transacciones de un modelo fracasan al presentarse la situación[8], decimos que el modelo no la soporta. Si existe una correlación entre la supervivencia de las transacciones y el grado de exposición a la situación, decimos que el modelo la soporta pero "se ve afectado". Si tal correlación no existe y las transacciones son exitosas decimos que el modelo la soporta "sin verse afectado"

Modelo		Salto de celda	Desconexión	LLT[9]	Concurrencia
Kangaroo		√√	√√	√√	√
Moflex		√√	√√	√√	√
IOT		√	√	√√	√√
WeakStrict	*weak*	√	√	√√	√√
	strict	×	×	√	√
Flat		×	×	√	√

Tabla 11: Clasificación empírica de los modelos.

Notación:
×: El modelo no soporta la característica
√: El modelo soporta la característica pero se ve afectado por ella
√√: El modelo soporta la característica y no se ve afectado por ella

7.2 Conclusiones finales

Este trabajo nos permitió comparar distintos modelos desde el punto de vista tecnológico y desde el punto de vista del usuario. Hemos podido observar que los modelos más sofisticados, si bien presentan un mayor costo tecnológico, ofrecen al usuario final una solución aceptable para los problemas a los que se enfrentan las transacciones en los ambientes móviles.

Desde el punto de vista tecnológico, el modelo de menor costo es Flat, ya que utiliza pocos recursos del MH y ninguno en el MSS, sin embargo, claramente no es adecuado para transacciones móviles. Los modelos IOT y WeakStrict permiten al usuario trabajar en forma desconectada a costa de mantener una copia local de los datos en el MH. En cambio, Moflex y Kangaroo liberan los recursos del MH transfiriendo el costo de ejecutar las transacciones a los MSS, haciendo un uso más eficiente de la red inalámbrica y mejorando sensiblemente la respuesta frente los saltos de celda, pero impidiendo al usuario interactuar con la aplicación durante una desconexión.

Sin embargo, a partir de los resultados de las simulaciones de aplicaciones individuales, pudimos comprobar que, a la hora de implementar una solución móvil a un problema concreto, la elección del modelo no depende únicamente de estos factores; es necesario tomar en cuenta las características de dicho problema.

[8] Llamamos "situación" a un salto de celda, una desconexión, una transacción de larga duración o un alto grado de concurrencia.
[9] Transacciones de larga duración (Long Lived Transactions)

Por ejemplo, Kangaroo no es apropiado para una aplicación en la que son aceptables algunos fracasos parciales.

Al tomar en cuenta los resultados en forma conjunta, comprobamos experimentalmente que los modelos estudiados se polarizan en tres grupos: aquellos que no soportan saltos de celda ni desconexiones (Flat), los que soportan ambos eventos pero su rendimiento se ve afectado (IOT y WeakStrict), y los que soportan ambos eventos sin verse afectados (Moflex y Kangaroo).

En forma complementaria, clasificamos los modelos según su nivel teórico de soporte al evento de desconexión: distinguimos entre los que soportan desconexiones del primer nivel (Moflex y Kangaroo) y los que soportan desconexiones del segundo nivel (IOT y WeakStrict). Los resultados experimentales fueron obtenidos considerando únicamente el soporte de primer nivel.

Es evidente la ausencia de modelos que soporten tanto la movilidad como ambos niveles de desconexión. Creemos que un modelo que tome elementos de Moflex y WeakStrict puede ocupar este lugar vacante y superar a los demás modelos. Esto puede comprobarse experimentalmente implementando dicho híbrido en el simulador presentado en esta tesis. Dejamos esta tarea como trabajo futuro.

Finalmente, observamos que el hecho de implementar los modelos permite detectar aspectos de los mismos que no están especificados con suficiente nivel de detalle.

7.3 Trabajo futuro

Durante el desarrollo del presente trabajo hemos encontrado diversos cursos de investigación que estaban fuera del alcance de esta tesis. Entre los trabajos futuros vislumbramos, en primer lugar, la implementación de nuevos modelos y, en segundo lugar, una serie de mejoras al simulador para aumentar su precisión y facilitar su uso.

Implementación de nuevos modelos: Para la presente tesis modelamos el comportamiento de cuatro modelos. Como trabajo futuro se puede encarar la implementación de otros modelos existentes, modificaciones a los mismos o modelos nuevos. Los autores de nuevos modelos pueden utilizar el simulador para comparar su performance con la de otros modelos.

Uno de los modelos nuevos que creemos sería interesante implementar es un modelo híbrido que tome elementos de Moflex y WeakStrict (ver <7.2 Conclusiones finales>), ya que creemos que un modelo con esas características puede comportarse sensiblemente mejor que los modelos existentes.

Capacidad del MH: En el simulador desarrollado para esta tesis, los elementos de un entorno móvil asumen capacidades infinitas de procesamiento, memoria y batería. Dado que las limitaciones de los *MH* en estos aspectos son un factor importante de los entornos móviles (ver <1.4 Características de un entorno móvil>), y tomando en cuenta que los modelos transaccionales móviles difieren entre si en la carga que representan para el *MH*, resulta de gran interés modelar estas limitaciones.

Saturación de canales: De manera similar a la capacidad de los *MH*, en este trabajo hemos asumido que las celdas en las que se mueven los *MH* tienen una capacidad infinita. Aunque este aspecto de las comunicaciones móviles ha sido estudiado en forma extensa, resulta de interés analizar el impacto de una capacidad limitada de canales por celda en los modelos transaccionales móviles, especialmente aquellos que hacen un uso más eficiente del medio inalámbrico.

Movimiento del MH: Una de nuestras asunciones es modelar eventos simples como un salto de celda en base a probabilidades. Para realizar una simulación mas precisa se podría modelar modelar el salto de celda en base a la posición, dirección y velocidad de un móvil (descritas en la Tabla 12) y a la posición, forma y tamaño de las celdas.

Overhead de tareas básicas: En la versión actual del simulador, la mayoría de las tareas administrativas que realizan los *MHs*, *MSSs* y *DBSs* no tienen una demora asociada. Por ejemplo, la creación de una base de datos local en el MH es instantánea. Tomar en cuenta el costo de tareas de este tipo aumentaría el realismo de las simulaciones.

Múltiples DBS: Si bien el simulador permite definir más de un *DBS*, todas las simulaciones que realizamos asumen que hay un sólo servidor de bases de datos. Como trabajo futuro se puede estudiar el impacto de utilizar más de un *DBS* en una simulación. Creemos, sin embargo, que el único resultado visible será la reducción de la probabilidad de ocurrencia de bloqueos en las transacciones, ya que dos transacciones que operan en distintas bases de datos no pueden bloquearse. Por otra parte, la multiplicidad de bases de datos es un punto de comienzo para experimentar el efecto de distintas estrategias de coordinación de finalización (por ejemplo, two-phase-commit).

Broadcast: En este trabajo asumimos que los paquetes de información se envían a un destino único. Sin embargo, es posible pensar esquemas de actualización y sincronización de cambios en las bases de datos locales a los *MH*, que utilice el mecanismo de *broadcast* para enviar un paquete que llegue a todos los *MH* conectados. Por ejemplo, un modelo transaccional móvil podría decidir abortar o compensar las transacciones locales al enterarse de que otro *MH* cambió globalmente un dato que estas transacciones usaron, ahorrándose de esta manera el gasto de tiempo, batería y ancho de banda asociado con una sincronización fallida con el servidor. Por lo tanto, para ampliar el espectro de modelos pasibles de ser implementados en el simulador, podrían modificarse los algoritmos de ruteo de paquetes para incluir el mecanismo de *broadcast*.

Interfaz de usuario: La interfaz del simulador provista con esta tesis es muy elemental y obliga al usuario a realizar una considerable cantidad de trabajo manual, especialmente para ejecutar simulaciones con reproducción (ver <2.2 Configuración>). Sería conveniente disponer de un mayor soporte a estas tareas, así como también una forma más conveniente y segura de editar los parámetros de la simulación, por ejemplo agregando un analizador que valide la sintaxis de los parámetros y que no falte ninguno.

Variable de movimiento	Distribución de probabilidad	parámetros	Fuente
Giro con un ángulo	Normal	(0, 30) grados	[Pe99]
Salto de celda (hacia cuál celda vecina)	Uniforme	N (cantidad de vecinas)	[Lij96]
Movimiento del *MH*	Normal	(5, 0.1) segundos	[Pe99]
Movimiento del *MH*	Gausiana truncada		[Lij96]
Velocidad promedio	Normal	0.1 y 0.45 unidades	[Pe99]

Tabla 12: Distribuciones con respecto al movimiento del móvil

Bibliografía Comentada

En este capítulo mencionaremos la bibliografía utilizada para desarrollar esta tesis así como un breve comentario sobre cada una de dichas lecturas.

Nota: *la fecha de actualización de estas citas es Febrero de 2003.*

Referencias bibliográficas

- [Bar99] Barbará, D. (1999). "Mobile Computing and Databases - A Survey". IEEE Transactions on Knowledge and Data Engineering, 11, 1.
    - *En este paper se analiza el estado del arte de la computación móvil, las diferencias entre un ambiente distribuido y las características propias todo desde el punto de vista del desarrollo de a paliaciones de base de datos. Se definen, además varios escenarios de utilización de entornos móviles.*
- [Ber97] Bernstein, P., Newcomer, E. (1997). "Principles of Transaction Processing for the Systems Professional". Morgan-Kaufmann.
    - *Este libro abarca una gran cantidad de temas con respecto a TP Systems y varios aspectos como arquitecturas, tolerancia a fallas, recuperación, etc. Aunque sin ahondar demasiado en ninguno de los temas*
- [Bu2000] Bu, R. C Simulación: Un enfoque práctico 2000, Limusa Noriega Editores. I.S.B.N. 968-18-1506-8
    - *Este libro nos ayudó a entender conceptos de simulación así como la forma de generar números aleatorios siguiendo distribuciones de probabilidad no uniformes.*
- [Buch92] Buchmann, A., Ozsu, M., Hornick, M., Georgakopoulos, D., Manola, F. (1992). "A Transaction Model for Active Distributed Object Systems. Chapter 5 from Database Transaction Models for Advanced Applications", G. Weikum, H. J. Schek, Morgan-Kaufmann
    - *Se define el modelo Multitransactions. Presentan el modelo D.O.M. (Distributed Object Management) para aplicaciones sobre objetos que corren en sistemas de procesamiento diferentes y heterogéneos.*
- [Chr93] Chrysanthis, P. (1993). "Transaction Processing in a Mobile Computing Environment". Proc. IEEE Workshop on Advances in Parallel and Distributed Systems, 77-82.
- [Cor2000] Coratella, A., Felder, M., Hirsch, R., Rodriguez, E. (2000). A *framework* for Analyzing Mobile Transaction, Journal of Database Management, Vol. 12, No 13, Julio-Septiembre 2001
    - *En este paper se define un framework para evaluar y comparar modelos transaccionales en ambientes móviles. Es el leit movit para el desarrollo del simulador descripto en esta tesis.*

- [Dal2000] Dalglish, B. A "Refresher" On Some Basics of Telecommunications Technology. *http://www.telecomtechstocks.com/nontechies.htm#history*
    - *Historia y tecnologías de comunicación inalámbrica.*
- [Dev87] Devore, J. Probability and Statistics for Engineering and the Sciences. Brooks / Cole Company 1987 ISBN 0-534-06828-6
    - *Este libro presenta de forma concisa la mayoría de los temas de estadística utilizadas en esta tesis.*
- [Dir2001] Dirckze, R, Le Gruenwald 2001 "A Pre-serialization transaction management technique for mobile multidatabases"
    - *En este paper se define el modelo de pre-serializacion para federaciones de bases de datos. Se definen criterios de aislamiento y atomicidad globales así como desconexiones y migración de contexto.*
- [Du97] Dunham, M., Helal, A., Balakrishnan, S. (1997). "A mobile transaction model that captures both the data and movement behavior". ACM Mob. Netw. Appl. 2, 149-162.
    - *En este paper se presenta el modelo Kangaroo.*
- [ECO90] Umberto Eco (1990). Cómo se hace una tesis. Círculo de Lectores, 1990, ISBN: 84-226-2823-6.
    - *Este libro da algunas ideas organizativas acerca de cómo preparar una tesis*
- [El90] Elmagarmid, A., Leu, Y., Litwin, W., Rusinkiewicz, M. (1990). A Multidatabase Transaction Model for InterBase. Proc. 16th VLDB Conference, Brisbane, Australia.
    - *En este paper se presenta el modelo Flexible Transactions que permite que algunas funcionalidades puedan ser ejecutadas por mas de un TP System. Además introduce un modelo mixto en donde conviven transacciones compensables con no compensables. Es el modelo base extendido por Moflex [Ku98].*
- [For94] G. H. Forman y J. Zahorjan. The Challenges of Mobile Computing. IEEE computer, Abril 1994
    - *Aquí se definen un conjunto de características propias de los ambientes móviles.*
- [Fra98] Frank, L., Zahle, T. (1998). "Semantic *ACID* Properties in Multidatabases Using Remote Procedure Calls and Update Propagations". Software-Practice & Experience, 28, 77-98.
    - *Describe un modelo en el cual se preservan las propiedades ACID a través de varios DBMS y explica como garantizar el cumplimiento de las mismas.*
- [Fra99] Frank, L. (1999). Atomicity Implementation in Mobile Computing. The 2nd International Workshop "Mobility in Databases and Distributed Systems", DEXA'99, Italy, 105-113.
- [Freu99] Freudenrich, C. How Laptops Work, A Brief History. http://computer.howstuffworks.com/laptop1.htm
    - *Breve historia de la evolución de las laptops.*

- [Gam95] Gamma, E. Helm, R. Johnson R. Vlissides, J Design Patterns: Elements of Reusable Object Oriented Software 1995 Addison Wesley
    - *Realizamos parte de diseño del simulador utilizando las ideas y conceptos explicados en este libro.*
- [Gar87] Garcia-Molina, H., Salem, K. (1987). "Sagas". Proc. SIGMod 1987 Annual Conference. ACM Press, 249-259,
    - *En este paper clásico se presenta una adaptación al modelo transaccional tradicional y se definen las Sagas que son explicadas en el capítulo de modelos transaccionales avanzados de esta tesis. Se presenta además un núcleo común de problemas correspondientes a las Transacciones de Larga Vida así como recuperación (hacia atrás y adelante) e ideas sobre como implementar este modelo sobre un TP System que respete las propiedades ACID.*
- [Grey93] Gray, J., Reuter, A. (1993). "Transaction Processing: Concepts and Techniques". Morgan Kaufmann.
    - *Este libro fue nuestra base para comenzar el estudio de TP Systems. Se definen muchos conceptos de transacciones.*
- [Imp99] Impagliazzo, J. Cronology of Computing History, Hofstra University. http://www.hofstra.edu/ComputingHistory/
    - *Eventos más significativos en la historia de la computación*
- [Ioa91] Ioannidis J, D. Duchamp y G. Q. Maguiere – "IP-Based protocols for Mobile Internetworking", Proceedings of ACM SIGCOMM Symposium on Communication, Architectures and Protocols pp 235 – 245
    - *Este paper dio origen al estándar Mobile IP, se definen soluciones para localizar a un móvil y para re rutear los mensajes correspondientes de manera transparente como si se tratara de un cliente fijo.*
- [Gra96] Gray, J., Helland, P., O'Neil, P., Shasha, D. (1996). "The Dangers of Replication and a Solution". ACM SIGMOD, Montreal, Canada.
- [Hei2000] Heidermann, J Bulusu N, "Effects of Detail in Wireless Network Simulation"
    - *En este paper se describen aspectos comunes al modelado de comunicaciones telefónicas a través de una red inalámbrica y estrategias para poder abstraer aspectos en la simulación. El paper ataca básicamente las capas de mas bajo nivel de la red móvil. Además se muestran diferentes trade-off según la estrategia utilizada y se habla bastante de la forma de visualizar los datos.*
- [Hos95] Hossein Arsham Systems Simulation: The Shortest Path from Learning to Applications http://ubmail.ubalt.edu/~harsham/simulation/sim.htm
    - *Este apunte es un compendio de información sobre todo tipo de técnicas de simulación.*

- [Imi92] Imielinksi T y Badrinath – "Querying in Highly Mobile Distributed Environment" Proceedings of 18th Conference on VLDB 1992
    - *Este paper fue uno de los primeros acercamientos a la distribución de transacciones en un ambiente móvil aunque sólo limitado a la posibilidad de hacer consultas.*
- [Ku98] Ku, K., Kim, Y. (1998). "Moflex Transaction Model for Mobile Heterogeneous Multidatabase Systems". Proceedings of the 10th International Workshop on Research Issues in Data Engineering, IEEE.
    - *En este paper se presenta el modelo Moflex*
- [Knu81] Capítulo 3 de ``The Art of Computer Programming. Vol 2: Seminumerical Algorithms". Donald E. Knuth. Addison Wesley, 1981.
- [Lee97] Lee, J., Simpson, K. (1997). "A High Performance Transaction Processing Algorithm for Mobile Computing". IASTED International Conference on Intelligent Information Systems (IIS '97)
- [Lee99] Lee, W., Hu, Q., Lee, D. (1999). "A study on channel allocation for data dissemination in mobile computing environments". Mobile Networks and Applications, 4, 117-129.
- [Liu96] Liu W, Chanig C, Wu H, Jha W, Gerla M Bagrodia R. "Parallel Simulation Environment for Mobile Wireless Networks" Proceedings of the 1996 Winter Simulation Conference
    - *Los autores muestran como llevar un simulador centralizado y secuencial hasta convertirlo en uno distribuido con mecanismos de envío de mensajes. Se basa principalmente en las capas inferiores de Mobile IP y se implementa utilizando MAISIE*
- [Lij96] Liljenstarm, M y Ayani, R "A Model for Parallel Simulation of Mobile Telecommunication Systems" Proceedings of the 4th International Workshop on modeling, analysis and simulation of computer telecommunications system (MASCOTS 96).
    - *Se describe cómo simular una corrida de un ambiente móvil balanceando la carga en varias computadoras paralelas. Además se describe un ejemplo completo con todos los parámetros y distribuciones de probabilidad de la simulación y criterios de validación de las corridas.*
- [Lu94] Lu, Q., Satyanarayanan, M. (1994). "Isolation-Only Transactions for Mobile Computing". ACM Operating Systems Review, 28, 2, 81-87
- [Lu95] Lu, Q., Satyanarayanan, M. (1995). "Improving Data Consistency in Mobile Computing Using Isolation-Only Transactions". Proc. 5th workshop on Hot Topics in Operating Systems.
    - *Se define el modelo Isolation-Only Transactions*
- [Maz99] Mazumdar, S., Chrysanthis, P. (1999). "Achieving Consistency in Mobile Databases through Localization in PRO-MOTION". Proc. 2nd DEXA Int'l Workshop on Mobility in Databases and Distributed Systems, Florence, Italy, 82-89.
- [Me2001] Mery, C. History of Range Palm. www.pda-mobile.com/anglais/ancienpalm.html
    - *Información sobre la historia de las Palm Pilot.*

- [MS2003] Microsoft Corporation. Technical Resources for MS SQL Server. "Detecting a Blocked Connection" http://www.microsoft.com/sql/techinfo/tips/development /blockedconnection.asp
    - *Referencia sobre el parámetro de control de timeout de bloqueo en SQL Server.*
- [Pan93] Panos K. Chrysanthis, Krithi Ramamritham. "Synthesis of Extended Transaction Model using ACTA". Dept. of Computer Science. University of Pittsburgh, 1993
- [Pe99] Perkins, Charles, Kuang Wang. Optimized Smooth Handoffs in Mobile IP. Proceedings of IEEE Symposium on Computers and Communications, Egypt, July '99
    - *Se propone una extensión al protocolo de Mobile IP para evitar el síndrome del triángulo. Además se mide la duración de un salto de celda y se analizan aspectos como tamaño de celda, cantidad de nodos, tamaño de los buffers etc.*
- [Pit94] Pitoura, E., Bhargava, B. (1994). "Building Information Systems for Mobile Environments". Proc. 3rd International Conference on Information and Knowledge Management, Gaithesburg, MD, 371-378.
- [Pit95] Pitoura, E., Bhargava, B. (1995). "Maintaining Consistency of Data in Mobile Distributed Environments". Proc. 15th International Conference on Distributing Computing Systems, Canada.
    - *En este paper se describe el modelo **Weak/Strict**, además se mencionan varias características comunes a todos los modelos móviles.*
- [Pit98] Pitoura, E. (1998). "Supporting Read-Only Transactions in Wireless Broadcasting". Proc. DEXA98 International Workshop on Mobility in Databases and Distributed Systems.
- [Pu88] Pu, C., Kaiser, G., Hutchinson N. (1988). Split-Transactions for Open-Ended Activities. Proc.14th VLDB Conference, Los Angeles, California.
    - *En este paper se presenta el modelo Split como una solución para el desarrollo de partes en sistemas CAD/CAM.*
- [Raj91] "The Art of Computer Systems Performance Analysis. Techniques for Experimental Design, Measurement, Simulation and Modeling" de Raj Jain, editado por John Wiley & Sons, Inc, ISBN 0-471-50336. Pág. 425.
    - *En este libro se describen técnicas de análisis de rendimiento.*
- [Re99] Q. Ren and M. H. Dunham. Using semantic caching to manage location dependent data in mobile computing. In Sixth Annual International Conference on Mobile Computing and Networking, MobiCom 2000, pages 210-221, Boston, MA, USA, August 2000. ACM SIGMOBILE. *http://ww.seas.smu.edu/~mhd/8391f01/ren00using.pdf*
    - *Este paper sugiere técnicas para responder consultas móviles basándose en la ubicación de los datos (L.D.D. – Location Data Dependecy), además muestra técnicas de caché semántico para búsquedas similares dentro de un conjunto de datos próximos.*

- [Roc2000] Rocha, M Mateus G, da Silva S. "QoS and Simulation Models in Mobile Communication Networks". Pages: 119 - 122 Series-Proceeding-Article 2000 ISBN:1-58113-304-9
    - *Presentan un simulador de eventos llamado Simula 2 que permite estudiar modelos de comunicaciones móviles. Esta pensado para protocolos orientados a conexión. Se simulan intentos de llamadas móviles en una ciudad discriminando por tipo de persona.*
- [San2001] S. K. Madria, B. K. Bhargava, "A Transaction Model to Improve Data Availability in Mobile Computing", Distributed and Parallel Databases, 10(2): 127-160, 2001. www.umr.edu/~madrias/cs401-02/transactionmodel.ppt
    - *Se define el modelo de preescritura.*
- [Sat96] Satyanarayanan, M. (1996). "Fundamental Challenges in Mobile Computing". School of Computer Science, Carnegie Mellon University, PODC, Philadelphia PA, USA.
- [Shi2000] Shiow-yang Wu, Yu tse-Chang, "An active Replication Scheme for Mobile Data Management" www.acm.org/sigmod/disc/p_anactivereplicashyu.htm
    - *En este paper se definen estrategias para replicación de datos en ambientes móviles basados en perfiles que 'siguen' a los usuarios.*
- [Sh95] Short, J, Bagrodia, R, Kleinrock, L "Mobile Wireless Network System Simulation" ACM Mobile Computing and Networking conference (Mobicom 95) November 1995
    - *Se define un ambiente de simulación para problemas de índole móvil (aunque no transaccionales). Se utiliza el lenguaje de simulación MAISIE. Este paper es citado por muchos otros subsiguientes ya que definió un framework común para simulaciones móviles.*
- [Ta95] Tanembaum – Computer Networks
- [Tew95] Tewari, R., Grillo, P. (1995). "Data Management for Mobile Computing on the Internet". Computer & Information Center - Temple University, ACM, Philadelphia PA, USA.
- [Ull88] Jeffrey D. Ullman. "Database And Knowledge" – Base Systems. Computer Science Press, 1998
    - *Libro clásico y fundacional para entender los conceptos transaccionales. La mayoría de las definiciones de base de datos fueron tomadas de este texto.*
- [Var98] Varshney, U. (1998). "Mobile Computing over the Internet". Proc. 31st Hawaii International Conference on System Sciences (HICSS'98).
- [Wa97] Gary D. Walborn, Panos K. Chrysanthis, PRO-MOTION: management of mobile transactions, Proceedings of the 1997 ACM symposium on Applied computing, p.101-108, April 1997, San Jose, California, United States .
    - *Este paper presenta un modelo móvil parecido a Weak/Strict basándose en el concepto de campacts para guardar información semánticamente relacionada.*
- [Wei91] Weikum, G. (1991). "Principles and Realization Strategies of Multilevel Transaction Management". ACM Transactions on Database Systems, 16, 1, 132-180.
    - *En este paper se presentan las transacciones multiniveles*

- [Yen97] Yen, L., Huang, T., Hwang, S. (1997). "A Protocol for Causally Ordered Message Delivery in Mobile Computing Systems". ACM Mobile Networks and Applications 2, 365-372.
- [Zha94] Zhang, A., Nodine, M., Bhargava, B., Bukhres, O. (1994). "Ensuring Relaxed Atomicity for Flexible Transactions in Multidatabase Systems". Proc. ACM SIGMOD Conf, 67-78.

Sitios de Interés

http://citeseer.org
Motor de búsqueda de papers del área de informática.

http://members.aol.com/johnp71/course/course.html
Curso de estadística y generación de números aleatorios con una distribución establecida.

http://ubmail.ubalt.edu/~harsham/simulation/sim.htm
Compendio de notas sobre simulación.

http://www-gris.det.uvigo.es/~rebeca/lsim/simulacion/node2.html
Otro compendio sobre simulación.

http://www.causascientia.org\math_stat\Dists\Compendium.html
Explicación acerca de varias distribuciones conocidas.

http://www.webnz.com\robert\index.html
Sitio web con información acerca de generación y testing de números aleatorios.

http://www.unsam.edu.ar/unsam/Biblioteca/trabajotesis/manejobibliografico.htm
Sitio con información acerca de cómo escribir una tesis.

Apéndices

Apéndice 1: Implementación del Simulador

1.1 Elección de las herramientas de implementación

Decidimos utilizar Java para codificar el simulador porque buscábamos un lenguaje orientado a objetos que resultara simple de utilizar y con la perspectiva de ejecutar simulaciones en diferentes sistemas operativos, pese a que esto último nunca ocurrió. A pesar de que suponíamos que el uso de un lenguaje de las características de Java tendría un impacto negativo en la performance del simulador, descubrimos que los tiempos de ejecución eran muy razonables.

Para el análisis de los resultados utilizamos Microsoft Access porque el volumen de datos generado por el simulador esta dentro de lo que esta herramienta puede manejar, es fácil de usar y genera gráficos razonablemente buenos. Sin embargo, a la hora de utilizar Access para analizar los resultados, descubrimos que con frecuencia superábamos el volumen de datos que es capaz de manejar, resultando en ocasionales cancelaciones, si bien luego vimos que era posible reducir el número de simulaciones sin por ello perder calidad en los resultados. También encontramos que los requerimientos de memoria eran superiores a los esperados, lo cual nos limitó al uso de una única máquina, dentro de las que teníamos a nuestra disposición, que cubría los requisitos de memoria y procesador.

1.2 Diseño de Clases

Clases de simulación

Estas clases definen el simulador al más alto nivel. La clase Simulador permite iniciar y frenar la simulación. A esta clase se asocian la clase *Network*, que representa la red con sus nodos (*MH*s, *MSS*s y *DBS*s) y vínculos (fijos y de radio), y la clase *Model*, que contiene las políticas del modelo que se está simulando.

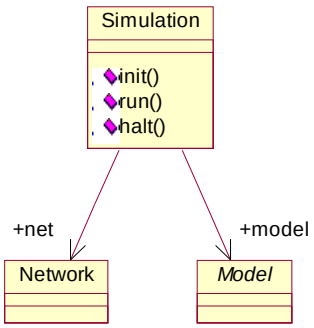


Figura 8: Clases de simulación

El método *run*() de la clase *Simulation* corre una simulación de acuerdo a los parámetros del archivo de configuración. La ejecución no termina hasta que se haya vaciado la cola de eventos, a menos que, por alguna condición anómala (por ejemplo la falla de un *assert*), se invoque el método *halt*(), en cuyo caso se vacía la cola de eventos, se impide la creación de nuevos eventos y se termina de procesar el evento actual.

Clases de modelado de eventos

Este grupo de clases define los eventos que maneja el simulador y las interfaces que deben cumplir todas las clases que deseen fabricar o atender eventos.

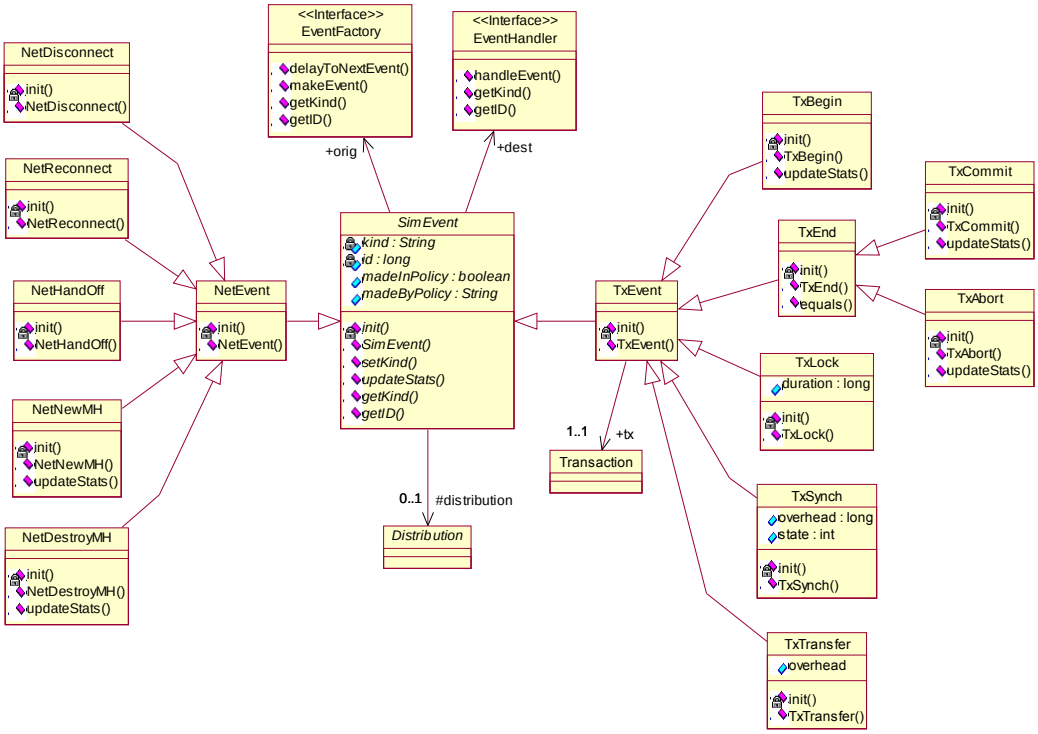


Figura 9: Clases de modelado de eventos

Todo evento es un *SimEvent*, es decir que tiene un tipo y un identificador, es capaz de actualizar las estadísticas globales (ver clase *StatsManager*), es fabricado por una clase que implementa la interfaz *EventFactory* y es enviado a una clase que cumple con la interfaz *EventHandler*.

Para fabricar eventos, una clase debe implementar la interfaz *EventFactory*. Esto implica que debe tener un tipo y un identificador y que debe implementar el método *delayToNextEvent*() que, dado un tipo de evento, debe indicar cuánto tiempo falta para que se produzca el siguiente evento de ese tipo (si devuelve cero significa que no producirá más eventos de ese tipo). También debe implementar el método *makeEvent*() que, dado un tipo de evento, devuelve un evento de ese tipo. Estos métodos son utilizados por la clase *Scheduler* para alimentar la cola de eventos.

Para atender eventos, una clase debe implementar la interfaz *EventHandler*, es decir que debe tener un tipo y un identificador y que debe implementar el método *handleEvent*(), que sea capaz de recibir un evento y actuar en consecuencia. Este método debe devolver un valor que indica, entre otras cosas, si el evento fue consumido o no. Esta información es usada para manipular la propagación de los eventos por la red y para recabar estadísticas para la herramienta de análisis. Específicamente, la respuesta del método *handleEvent*() puede tomar los valores detallados en la Tabla 13, que pueden ser sumados para combinar los efectos:

Símbolo	Valor	Significado
NO_USED	0	El evento no fue atendido
USED	1	El evento fue atendido y debe ser contabilizado en las estadísticas
NO_FORWARD	2	El evento debe ser consumido, no será retransmitido por la red hasta su destino
NO_DBMS	4	El evento no debe ser entregado al *DBMS* de la plataforma actual

Tabla 13: Códigos de procesamiento de eventos

Clases de administración de cola de eventos

Estas clases se encargan de administrar la cola de eventos y procesarlos en orden, y registran la información que será explotada al finalizar la simulación por el analizador de log.

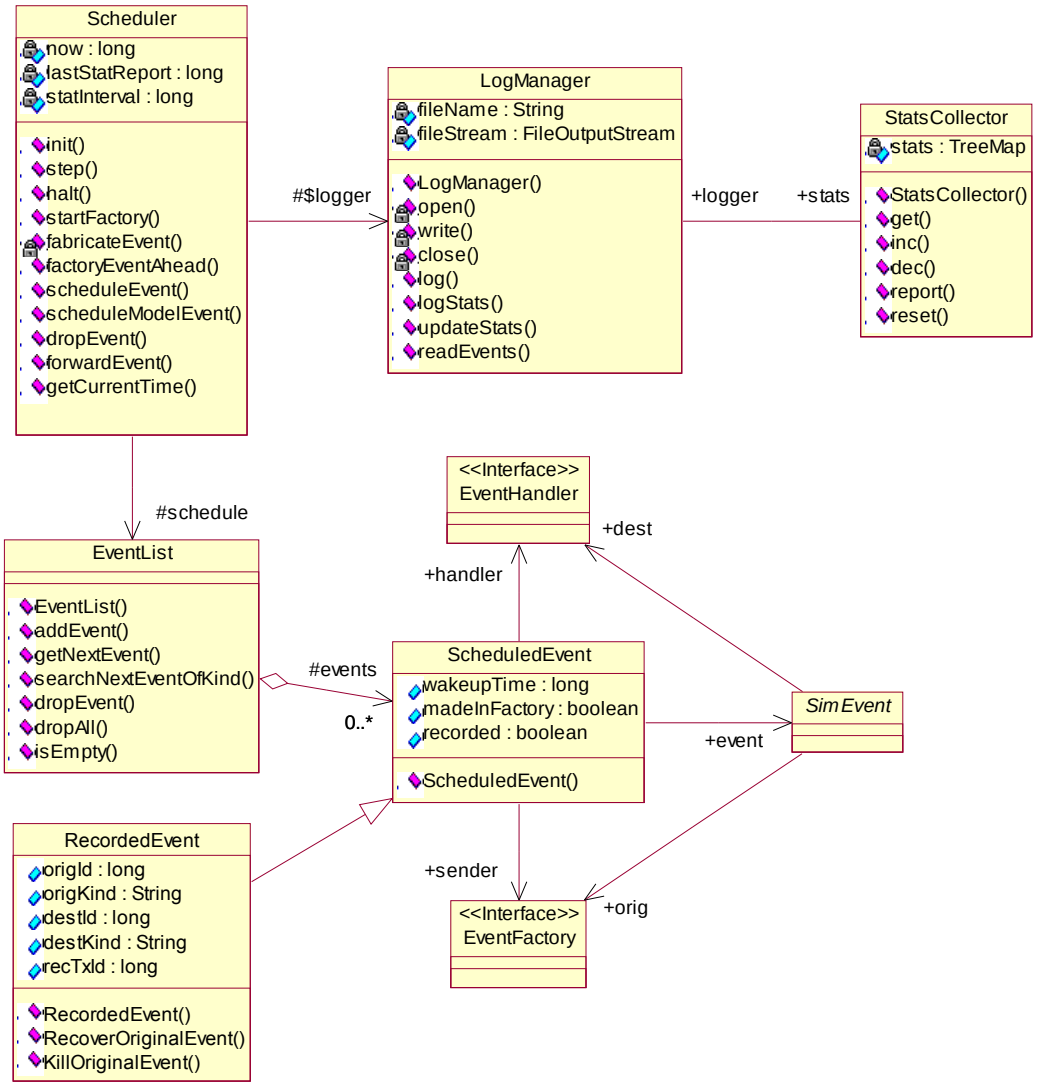


Figura 10: Clases de administración de cola de eventos

La clase *Scheduler* es el centro del simulador, ya que administra y distribuye los eventos que dan vida a la simulación. El método *step*() del *Scheduler* es invocado por el método *run*() de la clase *Simulation* para avanzar la simulación un paso, es decir procesar un evento. Esto implica:

- obtener el primer evento de la cola de eventos invocando a *getNextEvent*() de la clase *EventList*;
- instanciarlo, si es un evento reproducido, por medio de *recoverOriginalEvent*() de la clase *RecordedEvent*;
- avanzar el tiempo de la simulación (atributo *now*) hasta el tiempo del evento (atributo *weakeupTime* de la clase *ScheduledEvent*);
- entregarlo al *EventHandler* que deba atenderlo (atributo *handler* de *ScheduledEvent*);
- registrar el evento por medio del método *log*() de la clase *LogManager*;

- actualizar las estadísticas, de ser necesario, con el método *updateStats*() de la clase *SimEvent*, que utiliza el *StatsColector* para acumular estadísticas; y
- fabricar un nuevo evento del mismo tipo si el evento era fabricado, utilizando el método *makeEvent*() de la clase que lo fabrica (implementa la interfaz *EventFactory*) y el método *delayToNextEvent*() para determinar el *wakeupTime* del nuevo evento.

El *Scheduler* permite, también, registrar una fábrica de eventos, que será invocada inmediatamente para fabricar su primer evento, comenzando así la cadena de producción de cada fábrica, con el método *startFactory*(); determinar si un determinado evento está en la cola de eventos con el método *factoryEventAhead*(); agregar eventos nuevos con los métodos *scheduleEvent*() y *scheduleModelEvent*(); eliminar un evento de la cola con *dropEvent*() y consultar la hora simulada por medio de *getCurrentTime*().

La clase *EventList* mantiene una lista de eventos ordenada por tiempo. Cuando se agrega un evento con *addEvent*(), éste ocupa su lugar según su atributo *wakeUpTime*. Si ya existe un evento en ese mismo tiempo, el nuevo evento se atrasa un milisegundo. El método *getNextEvent*() devuelve el primer evento de la lista (el de menor *wakeUpTime*) y lo elimina de la lista. *EventList* también es capaz de recorrer la lista en busca de un evento de un tipo determinado, facilidad usada por el *Scheduler* para sus métodos *factoryEventAhead*() y *dropEvent*().

La clase *ScheduledEvent* es un contenedor para un evento, que le permite ser encolado para su procesamiento en un tiempo futuro (*wakeupTime*). Un caso especial de *ScheduledEvent* es el *RecordedEvent*, que agrega la posibilidad de encolar un evento que fuera grabado en una simulación anterior. Dicho evento carece, en el momento de ser encolado, de vínculos a los nodos a los que hace referencia, ya que los eventos grabados son encolados al principio de una simulación, antes de que los nodos sean creados. Por lo tanto el evento debe ser reconstruido por medio de *recoverOriginalEvent*() al momento de ser procesado. Este método obtiene de *Network*, a partir de sus identificadores, los nodos reales a los que el evento grabado hace referencia, transformándose en un evento funcionalmente idéntico al original.

El *LogManager* graba en un archivo de texto (el log) la información que recibe del *Scheduler*, para su posterior utilización en el Analizador del Log. La clase implementa distintas versiones del método log(), que generan distintos tipos de registro en el archivo de log. Los tipos de registro se detallan en la Tabla 14.

Tipo	Descripción
1	Información completa de un evento: su tipo, identificador, origen, destino, si es fabricado, política que lo generó, si fue atendido o no; si se trata de un evento de transacción, información de la transacción: identificador, *MH* al que pertenece, etc.
2	Estadística reportada por la clase *StatsManager*: cantidad de *MH*s, de transacciones activas, de eventos de cada tipo, etc.
3	Comentario insertado para facilitar el seguimiento manual del log
4	Definición del modelo actual: nombres de las políticas.
5	Información de una nueva fábrica de eventos: su tipo, identificador, tipo de evento que fabrica y la cantidad de fabricará.
6	Definición de una variable aleatoria: el tipo de distribución y sus parámetros
7	Variable iterada: el nombre de la variable y el valor que adquiere en la iteración actual

Tabla 14: Tipos de Log

El parámetro del archivo de configuración `LogFilter` permite especificar qué tipos de registro deben incluirse en el log. Por ejemplo, `LogFilter = 127` le indica al *LogManager* que sólo debe grabar registros de tipo 1, 2 y 7. Estos son, justamente, los tipos de registro indispensables para el funcionamiento del Analizador de Log. Si se desea realizar un seguimiento manual del log, se recomienda agregar el registro de tipo 3.

Finalmente, la clase *StatsColector* administra un conjunto de contadores para registrar estadísticas tales como la cantidad de ocurrencias de cada tipo de evento, la cantidad actual de *MH*s y la cantidad actual de transacciones activas. Los métodos *inc*() y *dec*() toman el nombre del contador y le suman o restan 1; *reset*() lo vuelve a cero, *get*() devuelve su valor actual y *report*() agrega al log el valor actual de todos los contadores.

Las estadísticas son afectadas por los eventos. Para ello, el *Scheduler* invoca el método *updateStats*() de cada evento que es atendido, y el evento incrementa o decrementa los contadores apropiados, de ser necesario. Por ejemplo, el evento *TxBegin* incrementa el número actual de transacciones activas y los eventos *TxCommit* y *TxAbort* lo decrementan.

Calcular las estadísticas de esta forma, si bien es más elegante y eficiente que, por ejemplo, pedirle a cada *DBMS* que cuente sus transacciones al momento de reportar las estadísticas, es más sensible a errores en la implementación de los modelos, ya que depende de la información que devuelve cada *EventHandler* respecto de si utilizó el evento o no. Sin embargo, esta sensibilidad permite detectar dichos errores: usando la técnica de las aserciones (implementada en la clase *Assert*), el simulador monitorea constantemente la consistencia de las estadísticas. Por ejemplo, ningún evento puede ser contabilizado dos veces, la cantidad de *MH*s debe ser cero al terminar la simulación, etc.

Clases de modelado de la red

Estas clases modelan un grafo con la topología de la red de comunicaciones simulada, con nodos unidos por dos tipos de vínculos: los enlaces de la red fija y los enlaces radiales.

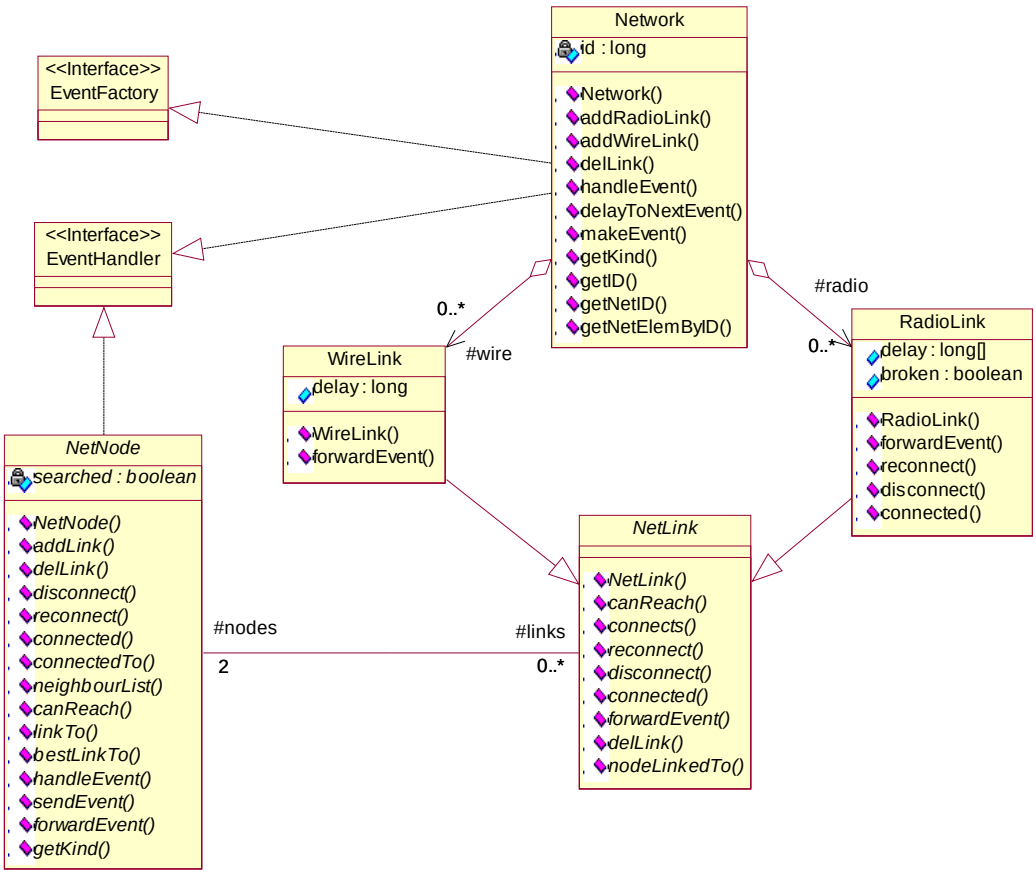


Figura 11: Clases de modelado de la red

La clase *Network* implementa las interfaces *EventFactory* y *EventHandler* porque fabrica el evento de creación de *MH*s (*NetNewMH*) y atiende los eventos de salto de celda (*NetHandoff*) y desconexión (*NetDisconnect*) y reconexión (*NetReconnect*). Además es la encargada de construir la red, agregando nodos y vínculos. También asigna identificadores únicos a los elementos de la red y es capaz de devolver un elemento de la red dado su identificador. Esto es útil para reproducir simulaciones, ya que los eventos grabados no pueden referenciar a sus nodos origen y destino directamente sino a través de sus identificadores.

Los enlaces (*WireLink* y *RadioLink*) unen nodos y envían eventos entre los nodos que unen. El enlace radial puede, además, desconectarse y reconectarse. Todo enlace tiene una velocidad asociada, valor que se define en el archivo de configuración.

Los nodos de la red (*NetNode*) también atienden eventos, ya que deben decidir por cual de sus enlaces reenviarlos para que lleguen a destino. Para ello, los métodos *bestLinkTo*() y *minDistanceTo*() colaboran utilizando el método *nodeLinkedTo*() de *NetLink* para entregarle a *forwardEvent*() la información necesaria para enviar el evento a su destino por el camino más corto (de menor cantidad de saltos). Los nodos también pueden desconectarse de la red y volver a reconectarse, desconectando o reconectando todos sus enlaces.

Clases de modelado de plataformas

Las plataformas son nodos de la red capaces de contener bases de datos, fabricar y atender eventos, y albergar políticas de modelos.

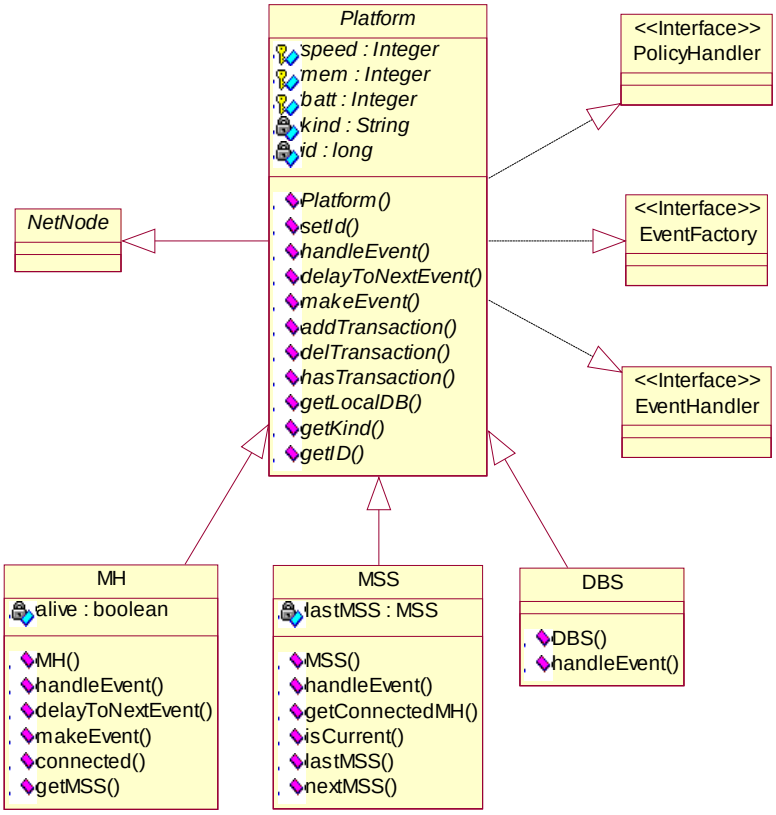


Figura 12: Clases de modelado de plataformas

Las tres plataformas simuladas en este trabajo (*MH*, *MSS* y *DBS*) son subclases de *Platform*. Estas clases, como todo *EventHandler*, pueden atender eventos con *handleEvent*(), crear eventos con *makeEvent*() y decir cuándo deben ser ejecutados, por medio de *delayToNextEvent*(). Las plataformas pueden además ser dueñas de transacciones: por ejemplo, un *MH* que inicia una transacción es dueño de la misma. Esta propiedad se implementa con *addTransacction*() para agregar una transacción, *delTransaction*() para eliminarla, y *hasTransaction()* para interrogar sobre la presencia de una transacción.

Si bien las plataformas están preparadas para simular ciertas propiedades físicas con los atributos *mem*, *speed* y *batt* (capacidad de la memoria, velocidad de procesamiento, capacidad de la batería respectivamente), dejamos como trabajo futuro la implementación del impacto de estas limitaciones en la simulación.

Una propiedad fundamental de las plataformas es que albergan las políticas de los modelos. Una política define qué acciones toma un modelo cuando un cierto evento arriba a una cierta plataforma. Por lo tanto,

luego de responder a un evento en forma genérica (por ejemplo, un *MSS* se desconecta cuando le llega el evento *NetDisconnect*), una plataforma invoca la política apropiada para ese evento en esa plataforma (por ejemplo, el *MSS* puede invocar a la política *NetDisconnectPolicy*, que aborta las transacciones activas) para que se tomen las acciones específicas del modelo simulado. Finalmente, si la plataforma contiene una base de datos y el evento es de tipo transacción (*TxBegin*, *TxCommit*, *TxAbort*, *TxLock* o *TxSynch*), le envía el evento a la base de datos para que lo atienda.

La plataforma *MSS* puede además responder si es el *MSS* al que está actualmente conectado el *MH*, y devolver los *MSS* anterior y siguiente en la cadena de *MSS*s que se va formando a medida que el *MH* cambia de celda y nuevos *MSS*s son creados.

Clases de modelado de bases de datos

Estas clases representan un subconjunto de las características de los *DBMS* y de las transacciones.

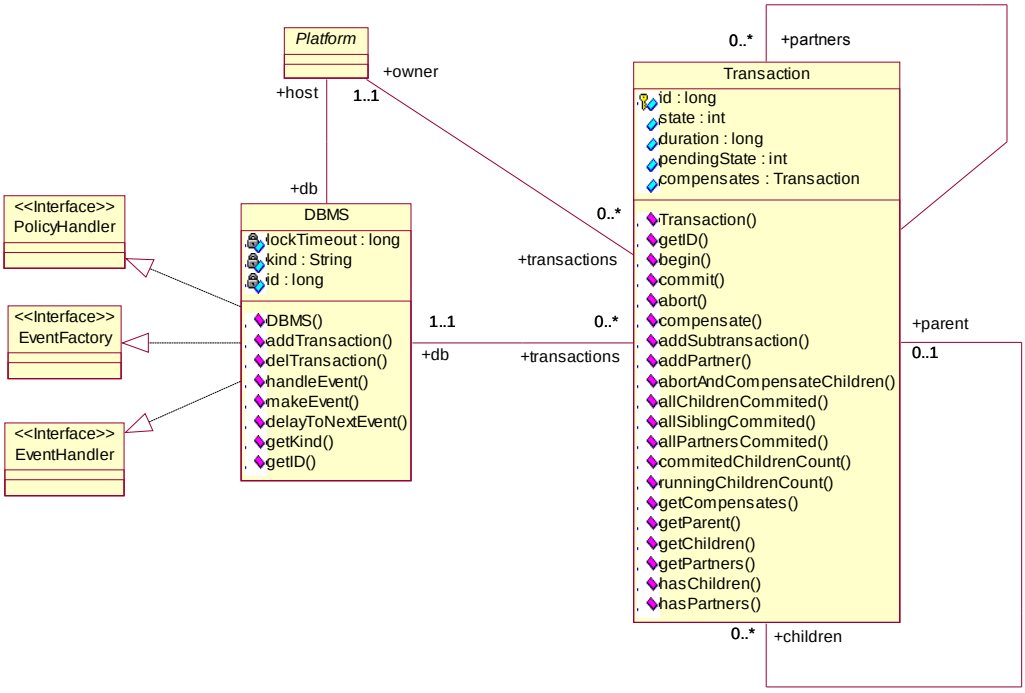


Figura 13: Clases de modelado de base de datos

La base de datos (*DBMS*) está alojada en una plataforma, a la que considera su "*host*". Al igual que las plataformas, la *DBMS* implementa las interfaces *EventHandler*, *EventFactory* y *PolicyHandler*, por lo que puede fabricar y atender eventos propios y albergar políticas de modelo. También contiene y administra transacciones, es decir que implementa el comportamiento de una transacción en una base de datos. Por ejemplo, al atender el evento *TxBegin* programa el evento de fin de la transacción (*TxCommit* o *TxAbort*) y sus posibles bloqueos (*TxLock*), y al atender el evento de fin de la transacción cambia el estado de ésta y avisa a su dueño. También responde al evento de sincronización (*TxSynch*), decidiendo si la sincronización fue exitosa o no y avisando el resultado a la plataforma que envió el evento.

La clase *Transaction* implementa las transacciones de las que hemos estado hablando. Toda transacción tiene una plataforma que la generó (atributo *owner*) y una base de datos que la alberga (atributo *db*). Cuando una transacción es creada, junto con el evento *TxBegin*, calcula su duración (en el atributo *duration*) en base a parámetros del archivo de configuración, y asume el estado *NEW* (en el atributo *state*). Cuando el evento *TxBegin* llega a su *DBMS* destino, se invoca el método *begin*(), que cambia su estado a *RUNNING*. Cuando se produce el evento *TxCommit* o *TxAbort*, asume el estado *COMMITED* o *ABORTED* según sea el caso. Un

modelo puede también utilizar los estados *WAITING_COMMIT* y *WAITING_ABORT*, o el atributo *pendingState* para implementar diversos estados intermedios de las transacciones.

Las transacciones también pueden relacionarse entre si de dos maneras: una relación jerárquica de transacción principal y subtransacciones (implementada con el método *addSubtransaction*() y los atributos *parent* y *children*), o una relación de pares (implementada con el método *addPartner*() y el atributo *partners*). Una transacción que participa de estas relaciones puede entregar las transacciones con las que se relaciona y ser interrogada respecto del estado de las mismas.

Las transacciones pueden ser compensadas con otra transacción (es la única forma de abortar una transacción que ya publicó sus cambios mediante un *commit*). Como una transacción compensatoria no puede abortar, la base de datos siempre generará un *TxCommit* para este tipo de transacciones.

Clases de abstracción de políticas de respuesta a eventos

Las políticas definen el comportamiento de un modelo ante un evento determinado en una plataforma determinada. Definir un modelo consiste en definir sus políticas.

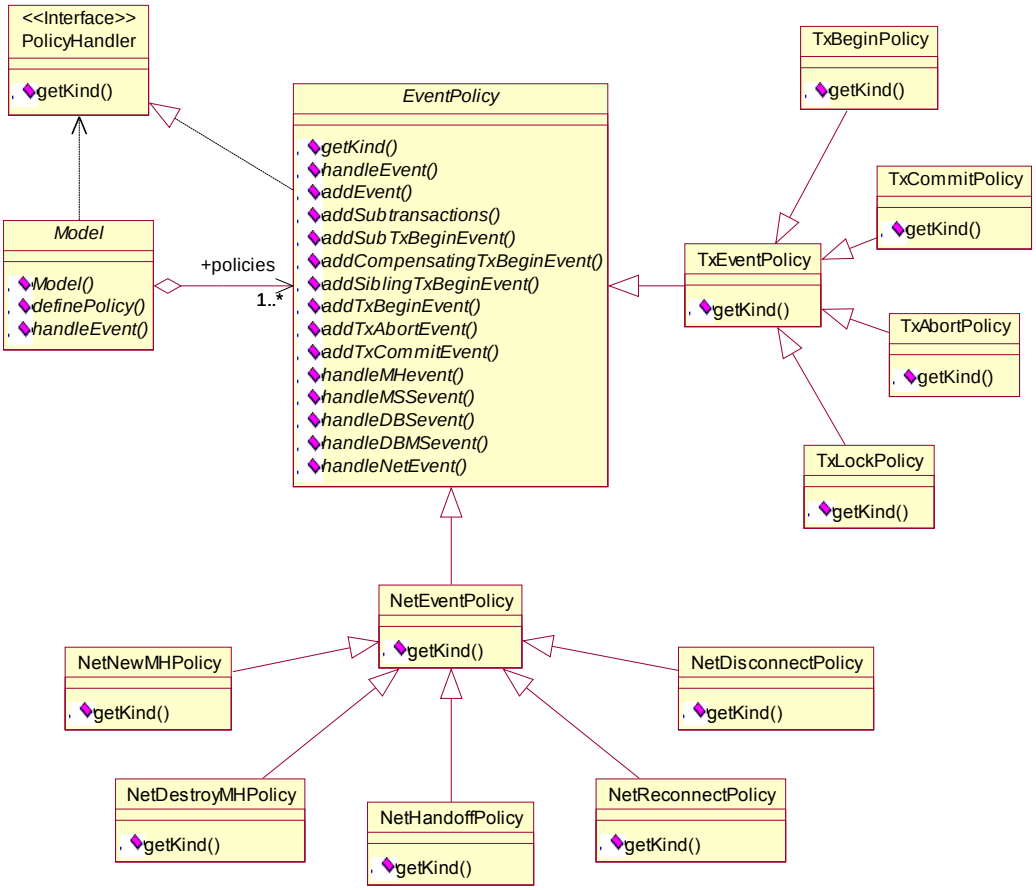


Figura 14: Políticas de abstracción de políticas de respuesta a eventos

La clase *EventPolicy* cumple una doble función: abstraer todas las políticas para poder referenciarlas en forma genérica y proveer una interfaz (API) para el programador que le simplifique el uso de los servicios más comunes del simulador.

El usuario del simulador que desee definir un nuevo modelo deberá crear nuevas políticas como subclases de *EventPolicy* o alguna de las políticas disponibles y luego declarar estas políticas en el archivo de configuración.

La clase *Model* es la encargada de vincular las clases del modelo descritas en la configuración con el resto del simulador. Cuando una plataforma, luego de atender un evento en forma genérica, desea pasarlo a la política correspondiente, invoca el método *handleEvent*(), que analiza de qué evento se trata y en qué plataforma se encuentra, e invoca el método apropiado de la política apropiada del modelo actual. Por ejemplo, si se trata de un *TxAbort* en un *DBS* y el modelo actual es IOT, invocará el método *handleDBSevent*() de la política cuyo nombre se encuentra definido en el parámetro *Model_IOT_Policy_Abort* del archivo de configuración. El valor de este parámetro debe coincidir con el nombre de la clase que implementa dicha política.

En la Tabla 15 se describen los métodos correspondientes a la interfaz del programador (API) implementados por la clase *EventPolicy*:

Método	Descripción
addEvent (*SimEvent, EventHandler, long*)	Toma como parámetro un nuevo evento, un destinatario del evento y la demora en milisegundos hasta su activación, y lo agrega a la cola de eventos.
addTxBeginEvent (*Platform, Platform*)	Programa un evento *TxBegin* a ser atendido inmediatamente, asignando como dueño de la nueva transacción y como *host* de base de datos las plataformas indicadas en el primer y segundo parámetro respectivamente.
addTxCommitEvent (*Platform, Transaction*)	Programa un evento *TxCommit* para la transacción pasada por parámetro, a ser atendido inmediatamente, tomando como origen del evento la plataforma del argumento.
addTxAbortEvent (*Platform, Transaction*)	Ídem anterior pero programa un evento *TxAbort*.
addSubTxBeginEvent (*Platform, Platform, Transaction*)	Crea un evento *TxBegin* de atención inmediata, con el origen y destino especificados en el primer y segundo parámetro respectivamente. La nueva transacción será subtransacción de la especificada en el tercer parámetro.
addCompensatingTxBeginEvent (*Platform, Platform, Transaction*)	Crea un evento *TxBegin* para una transacción compensatoria de la indicada en el segundo parámetro, asignando como dueño de la nueva transacción y como *host* de base de datos las plataformas indicadas en el primer y segundo parámetro respectivamente.
addSiblingTxBeginEvent (*Platform, Platform, Transaction*)	Es equivalente a invocar a *addSubTxBeginEvent()* con la transacción padre, es decir que crea un *TxBegin* para una nueva transacción con el mismo padre que la del parámetro.
addSubtransactions (*long, Platform, Platform, Transaction*)	Crea un tantos eventos *TxBegin* de atención inmediata como se indica en el primer parámetro, asignando como dueño de la nueva transacción y como *host* de base de datos las plataformas indicadas en el segundo y tercer parámetro respectivamente. Las nuevas transacciones serán hijas de la transacción indicada en el cuarto parámetro, y su número no superará el indicado como límite en el parámetro del archivo de configuración *"DBMS_MaxNumberOfSubTx"*.

Tabla 15: Métodos públicos de la API

Clases de modelado de distribuciones

Estas clases implementan las distintas distribuciones discretas y continuas que se utilizan en el simulador.

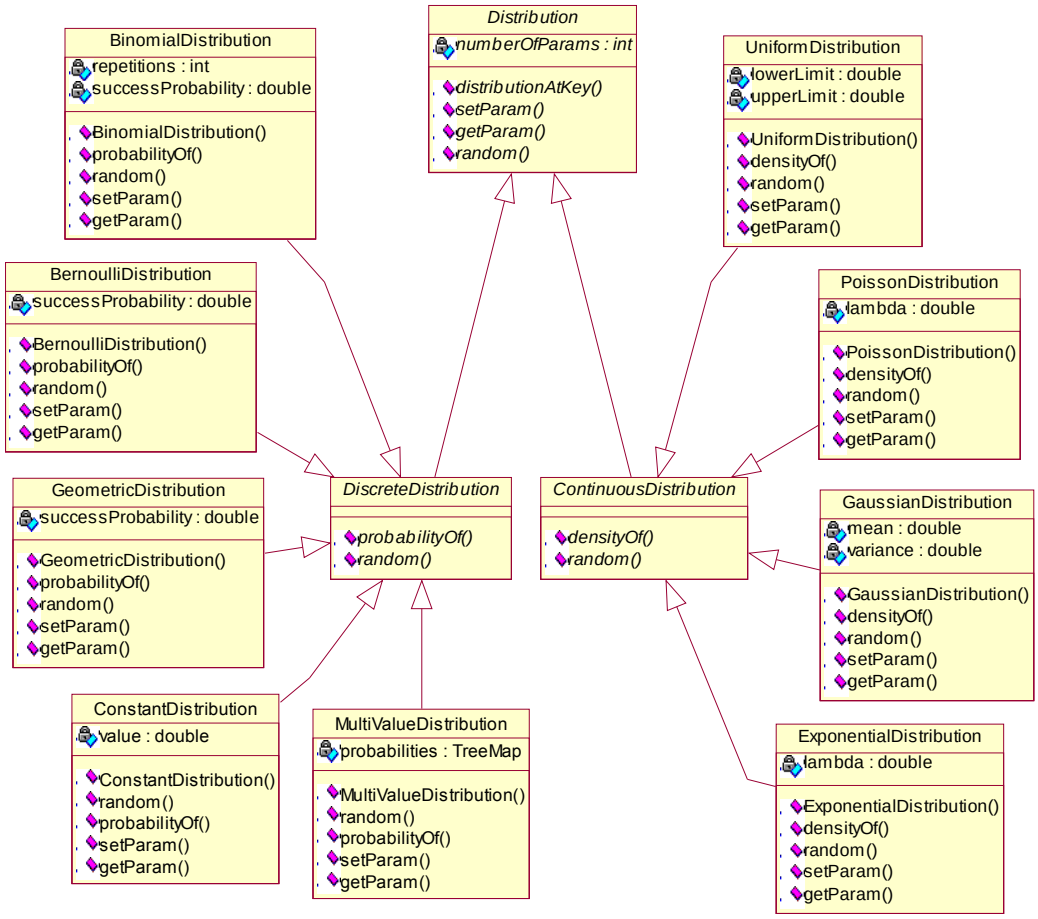


Figura 15: Clases de modelado de distribuciones

Todas las distribuciones heredan de la clase *Distribution*, por lo que se instanciar sus parámetros con *setParam*() y generar un número aleatorio con una distribución acorde a lo indicado por dichos parámetros, con *random*(). Distintas distribuciones tienen distinta cantidad de parámetros, por lo que la asignación de parámetros se realiza de a uno por vez.

Las clases abstractas *DiscreteDistribution* y *ContinuousDistribution* diferencian las distribuciones cuyos valores tiene una probabilidad de las que tienen una función de densidad acumulada. Cualquiera de los tipos de distribución puede generar un número aleatorio con la distribución indicada por sus parámetros.

El método *distributionAtKey*() de la clase *Distribution* toma el nombre de una variable del archivo de configuración y devuelve una instancia de la distribución indicada en dicha variable, obteniendo los parámetros de la distribución a partir de las variables asociadas a la mencionada, que comienzan con el mismo nombre y lo completan agregando "*_param1*", "*_param2*", etc.

Clases auxiliares

Las siguientes clases implementan funcionalidades auxiliares que no pueden ser agrupadas con las clases vistas hasta ahora.

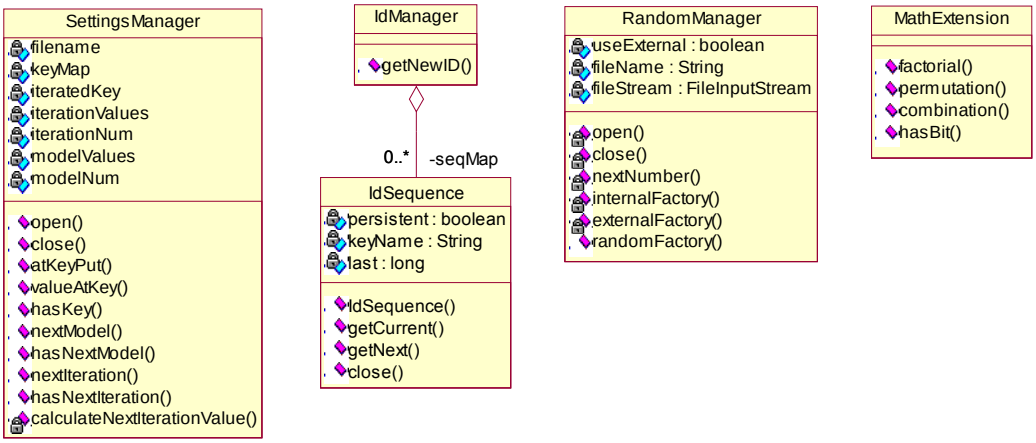


Figura 16: Clases auxiliares

La clase *SettingsManager* administra el archivo de configuración. Es capaz de leerlo con *open*(), devolver el valor de un parámetro con *valueAtKey*() y determinar si un parámetro está definido. Al cerrar con *close*() graba el archivo de configuración, persistiendo de esta forma los valores que pudieron haber sido cambiados con *atKeyPut*().

Adicionalmente, *SettingsManager* administra las iteraciones definidas en las variables iteradas. Al leer una variable iterada, calcula todos los valores que deberá tomar con repetidas invocaciones al método *calculateNextIterationValue*() y los almacena en el atributo *iterationValues*. El simulador utiliza entonces los métodos *hasNextIteration*() y *nextIteration*() para que la variable iterada asuma sucesivamente los valores calculados y repite la simulación con cada nuevo valor (ver <1.7 Formato del archivo de configuración>).

Lo mismo sucede con la lista de modelos en la variable `CurrentModel`. Al leer esta variable del archivo de configuración, *SettingsManager* registra los valores que deberá asumir en el atributo *modelValues* y permite que el simulador repita las simulaciones con los distintos valores invocando *hasNextModel*() y *nextModel*().

Los eventos, las plataformas y las transacciones se identifican por medio de números secuenciales que obtienen del *IdManager*. Esta clase, junto con *IdSequence*, implementan dos tipos de secuencias: una secuencia genérica que no se persiste y que comienza en cero con cada nueva simulación, y secuencias persistentes identificadas con un nombre que corresponde al nombre de una variable del archivo de configuración. Al inicializar la clase *IdSequence* el último valor persistido se lee del archivo de configuración, luego se obtienen nuevos valores de la secuencia mediante *getNext*(), y finalmente el último valor generado se graba en el archivo de configuración cuando la simulación termina y se invoca el método *close*().

La clase *RandomManager* implementa dos formas de obtener números aleatorios con distribución uniforme entre cero y uno: el generador pseudo-aleatorio provisto por Java y un archivo externo de bits aleatorios. Su único método público, usado por las clases que implementan las distintas distribuciones, es *randomFactory*(), que obtiene el siguiente número aleatorio. La elección de uno u otro método depende del valor del parámetro del archivo de configuración "`UseExternalRandomGenerator`". Si se utiliza el archivo externo, la clase *RandomManager* consume, al inicializar, una cantidad aleatoria números, para evitar que todas las simulaciones obtengan siempre la misma secuencia de números. La cantidad de números consumidos se determina con un número pseudo-aleatorio utilizando la función nativa de Java.

Finalmente, la clase *MathExtension* implementa algunas funciones matemáticas de uso común no provistas por Java, principalmente para generar distribuciones aleatorias: el factorial de un número, permutaciones y combinaciones, y el estado de un bit en un entero.

1.3 Descripción de los eventos

En esta sección explicaremos cada evento del sistema en detalle. En cada caso, indicaremos qué función cumple, quién lo fabrica, quién es el destinatario y qué distribución utiliza. Con respecto a esto último es de notar que, aunque en nuestra implementación permitimos asignar cualquier distribución de probabilidad a los eventos, utilizamos las funciones de densidad mencionadas en [Liu96] [Lij96] [roc2000].

NetNewMH: Este evento, fabricado y atendido por la entidad Red, causa la aparición de un nuevo *MH* en el sistema. El simulador le asigna un nuevo *MSS*, crea las nuevas conexiones y lo inicializa. Como parte de la inicialización, el nuevo *MH* se registra como fábrica de los eventos *TxBegin*, *NetHandoff* y *NetDisconnect*.

La generación de este evento está determinada por las siguientes dos variables aleatorias `NetNewMH_Distribution` y `NetNewMH_Delay_Distribution`. La primera determina la cantidad de *MH*s que serán simulados y sigue una distribución Gausiana. La segunda determina el tiempo que debe transcurrir entre dos creaciones sucesivas de *MH*s y sigue una distribución Exponencial.

NetDestroyMH: Cuando un *MH* recibe la noticia del fin de la última transacción pendiente, genera el evento *NetDestroyMH* y, al recibirlo, se elimina del sistema. Como este evento depende de la evolución de las transacciones, no sigue ninguna distribución ni es reproducido.

NetHandoff: Este evento corresponde a un salto de celda de un *MH*. Es fabricado por el *MH* que cambia de celda y atendido por su *MSS*, quien realiza el salto de celda creando un nuevo *MSS*, vinculándolo con las *DBS* y con el *MSS* anterior, desconectando luego el *MH* del *MSS* anterior y reconectándolo al nuevo. La generación de este evento responde a la variable aleatoria `NetHandoff_Distribution`, que determina el intervalo de tiempo entre dos saldos de celda para un *MH*, y sigue una distribución Exponencial.

NetDisconnect: Este evento corresponde a la Interrupción temporaria de la comunicación radial entre un *MH* y su *MSS*. Es fabricado por el *MH* que sufre la desconexión y atendido por su *MSS*, quien realiza la desconexión y crea el evento de reconexión. La generación del evento *NetDisconnect* responde a la variable aleatoria `NetDisconnect_Distribution`, que determina el intervalo de tiempo entre dos desconexiones para un *MH*, y sigue una distribución Exponencial.

NetReconnect: Al recibir el evento *NetDisconnect*, el *MSS* genera su correspondiente evento de reconexión dirigido al *MH* desconectado y utilizando la variable aleatoria `NetReconnectDelay`, que sigue una distribución Exponencial.

A diferencia de los demás eventos, tanto *NetDisconnect* como *NetReconnect* viajan por la red en forma instantánea, es decir que no se ven afectados por la velocidad de transmisión. Esto tiene sentido ya que la desconexión es un hecho físico, no un mensaje transmitido bit a bit por la red.

TxBegin: Un *MH* fabrica este evento para dar comienzo a una nueva transacción en el *DBS* al que va dirigido. La transacción se crea en el mismo momento que el evento *TxBegin*, pero está "pendiente" hasta llegar a destino. Este evento está controlado por las variables aleatorias `MH_TransactionsPerMH` y `DBMS_TxSuperpositionFactor`. Ambas siguen distribuciones Gaussianas. La primera dicta cuántas transacciones lanzará el *MH*; la segunda indica en qué medida se superponen dos transacciones sucesivas: si vale 0.5 la segunda se lanza mientras la primera está por la mitad; si vale 2, la segunda se lanza luego de esperar un tiempo igual a la duración de la primera; etc.

Cuando el evento es atendido por el *DBS*, éste lo envía a su *DBMS*, quien calcula la duración de la transacción tomando en cuenta un posible bloqueo, y genera el evento de fin de la transacción: un *TxCommit* o un *TxAbort*. En otras palabras, el destino de una transacción se decide cuando esta comienza.

TxCommit: Este evento representa el fin exitoso de una transacción. Es generado por el *DBMS* y dirigido al *MH* dueño de la transacción, al atender el evento *TxBegin*, siempre que la variable aleatoria `DBMS_txSuccess`, que sigue una distribución de Bernoulli, indique que la transacción debe ser exitosa y siempre que la transacción no aborte por bloqueo. El evento es programado para ocurrir en un tiempo que depende de la duración intrínseca de la transacción y de la duración del posible bloqueo. Cuando el *DBMS* atiende este evento cambia el estado de la transacción para indicar el fin exitoso.

TxAbort: Al atender el evento *TxBegin*, el *DBMS* genera este evento si la variable aleatoria `DBMS_txSuccess` indica que la transacción debe abortar, o si la transacción debe abortar por bloqueo. El evento es programado para ocurrir en un tiempo que depende de la duración intrínseca de la transacción y, si termina por bloqueo, del momento en el que ocurre dicho bloqueo. Cuando el *DBMS* atiende este evento cambia el estado de la transacción para indicar su fracaso.

TxLock: Este evento representa el bloqueo de una transacción por parte de otra. Es generado por el *DBMS* al momento de atender el *TxBegin*, siempre que los parámetros que lo controlan así lo indiquen. El principal de estos parámetros es `DBMS_txLockProbability`, que indica con qué probabilidad ocurrirá un bloqueo habiendo exactamente dos transacciones activas. El *DBMS* toma en cuenta este valor y la cantidad de transacciones activas para determinar si debe producirse un bloqueo. Luego utiliza otros parámetros para calcular al duración del bloqueo y si puede causar que la transacción aborte, lo cual se explica detalladamente en la sección correspondiente.

TxSynch: Algunos modelos utilizan este evento para determinar si una transacción local al *MH* puede publicar sus cambios en la base de datos del *DBS*. El evento es generado por el *MH* con el *DBS* como destinatario, quien lo pasa al *DBMS*, y el *DBMS* responde con el mismo evento agregando únicamente el resultado de la sincronización, para lo cual utiliza la variable aleatoria `DBMS_SynchSuccess`, que sigue una distribución de Bernoulli.

TxTransfer: Este es otro evento de uso opcional, cuyo propósito es representar la transferencia del contexto de una transacción entre dos plataformas, típicamente *MSSs*. Es generado por un *MSS* y atendido por otro *MSS*, y usa la variable aleatoria `DBMS_TxTransferOverhead`, de distribución Exponencial, para determinar el tiempo que lleva hacer la transferencia.

1.4 Distribuciones utilizadas

Para encontrar las distribuciones asociadas a cada aspecto de la simulación fue necesario consultar en muy diversas fuentes, entre ellas la literatura referente a telefonía móvil [Liu96] [Cor2000], la referente a sistemas de transacciones [Grey93], la referente a simulación [Bag94] y, no menos importante, a nuestra experiencia personal en el uso de distintos dispositivos similares a los simulados en este trabajo. En nuestra modelización permitimos asignar cualquier distribución de probabilidad a los distintos eventos y a otras variables que definen diversos aspectos del sistema simulado. [Liu96] [Lij96] [roc2000]

En la Tabla 16 mostramos los distintas variables del simulador que fueron modeladas con distribuciones de probabilidad. Si bien estas no son todas las que se pueden modelar, son aquellas que son independientes de los modelos transaccionales móviles. Cada modelo agrega sus propias variables definidas en base a distribuciones (por ejemplo, Moflex usa una distribución para modelar la probabilidad de haber alcanzado un estado aceptable).

Variable modelada	Distribución de probabilidad	Explicación de los parámetros
Frecuencia de salto de celda de un *MH*	Exponencial(**t**)	**1/t** representa el tiempo de permanencia en la celda actual, en milisegundos
Frecuencia de desconexión de un *MH*	Exponencial(**t**)	**1/t** representa el tiempo entre desconexiones, en milisegundos
Duración de desconexión	Exponencial(**t**)	**1/t** representa el tiempo de desconexión, en milisegundos
Frecuencia de ingreso de nuevos *MH* al sistema	Exponencial(**t**)	**1/t** representa el tiempo entre apariciones de nuevos *MH*, en milisegundos
Cantidad de *MHs*	Gauss(**μ, σ**)	**μ** y **σ** representan la media y la varianza de la cantidad de *MH* simulados en el sistema
Transacciones por *MH*	Gauss(**μ, σ**)	**μ** y **σ** representan la media y la varianza de la cantidad de transacciones lanzadas por un *MH*
Subtransacciones por	Gauss(**μ, σ**)	**μ** y **σ** representan la media y la varianza de la cantidad

Transacción		de subtransacciones en las que se descompone una transacción
Superposición de transacciones	Gauss(μ, σ)	μ y σ representan la media y la varianza del grado en el que se superponen las sucesivas transacciones de lanzadas por un *MH*
Cantidad de accesos a la base por transacción	Gauss(μ, σ)	μ y σ representan la media y la varianza de la cantidad de accesos a la base realizados por una transacción
Tiempo entre accesos a la base	Constante	El tiempo de acceso a la base es despreciado por ser una de las simplificacioens asumidas, su modelado queda como trabajo futuro
Tiempo de procesamiento por acceso a la base	Constante	El tiempo de procesamiento de las consultas es despreciado por ser una de las simplificaciones asumidas, su modelado queda como trabajo futuro
Resultado de una subtransacción	Bernoulli(**p**)	**p** representa la probabilidad de éxito de una subtransacción
Bloqueos entre transacciones	Bernoulli(**p**)	**p** representa la probabilidad de que se produzca un bloqueo entre dos transacciones cualesquiera
Resultado de una sincronización	Bernoulli(**p**)	**p** representa la probabilidad de que la sincronización de una transacción en el motor de base de datos central tenga éxito (para los modelos que la usan)

Tabla 16: Distribuciones utilizadas en el simulador

1.5 Políticas predefinidas

Si el modelo especificado en el archivo de configuración no tiene ninguna política, el simulador asume que el modelo subyacente es Flat. Según nuestra interpretación del comportamiento de este modelo en un ambiente móvil, el simulador aborta sus transacciones ante cualquier salto de celda o desconexión. Este comportamiento está definido por las políticas predefinidas (exceptuando a las dos últimas), que se detallan en la Tabla 17. Los eventos a los que responden son evidentes por sus nombres.

Política	PolicyHandler	Descripción
TxBeginPolicy	*DBMS*	Invoca el método *begin*() de la transacción asociada al evento
TxCommitPolicy	*DBMS*	Invoca el método *commit*() de la transacción asociada al evento
TxAbortPolicy	*DBMS*	Invoca el método *abort*() de la transacción asociada al evento
TxLockPolicy	-	Ignora el evento
TxSynchPolicy	-	Ignora el evento
TxTransferPolicy	-	Ignora el evento
NetNewMHPolicy	-	Ignora el evento
NetDestroyMHPolicy	-	Ignora el evento
NetHandoffPolicy	*MSS*	Reenvía el evento a cada *DBS* conectado al *MSS*
	DBS	Recorre las transacciones locales al *DBMS* y genera un *TxAbort* para cada transacción perteneciente al *MH* que inició el Salto de Celda
NetDisconnectPolicy	*MSS*	Reenvía el evento a cada *DBS* conectado al *MSS*
	DBS	Recorre las transacciones locales al *DBMS* y genera un *TxAbort* para cada transacción perteneciente al *MH* que inició el Disconnect
NetReconnectPolicy	-	Ignora el evento
IgnoreDisconnectPolicy	-	Ignora el evento
IgnoreHandoffPolicy	-	Ignora el evento

Tabla 17: Políticas predefinidas

Puede verse que las políticas *NetHandoffPolicy* y *NetDisconnectPolicy* funcionan en dos fases: cuando el evento llega al *MSS*, el método *handleMSSevent*() de la política lo redirecciona al *DBS* (primiera fase). El evento luego llega al *DBS* y el método *handleDBSevent*() de la política genera los *TxAbort* para las transacciones del *MH* (segunda fase). Ambas fases son necesarias, porque el *MSS* no conoce las transacciones

del *MH*, y el *DBS*, que si las conoce, no recibiría el evento si el *MSS* no se lo hace llegar, ya que el destinatario final de los eventos *NetHandoff* y *NetDisconnect* es siempre el *MSS* al que está conectado el *MH*. Este tipo de colaboración es típico en la implementación de los modelos.

1.6 Políticas de los modelos implementados

Flat (tradicional)

El modelo plano es implementado por las políticas predefinidas (ver <1.5 Políticas predefinidas>). Dado que todos los modelos deben construirse en base a estas políticas (redefiniendo las que sea necesario), es importante comprenderlas para poder implementar nuevos modelos, por lo que decidimos profundizar en su descripción.

TxBeginPolicy: Esta política actúa únicamente cuando el evento *TxBegin* llega al *DBMS* destinatario, cambiando el estado de la transacción de *NEW* a *RUNNING* y devolviendo como resultado *USED* para que el evento sea contabilizado en las estadísticas.

TxCommitPolicy: Al igual que la política de *TxBegin*, la de *TxCommit* sólo actúa cuando el evento llega al *DBMS* que aloja a la transacción asociada, cambiando su estado a *COMMITED* y devolviendo *USED* para que sea contabilizado.

TxAbortPolicy: Esta política es idéntica a la anterior, excepto que pasa la transacción al estado *ABORTED*.

NetHandoffPolicy: Como el modelo Flat no es móvil, debe abortar ante eventos que no maneja, por lo los eventos *NetHandoff* y *NetDisconnect* implementan la cancelación de las transacciones del *MH* que los produce. La política de *NetHandoff* para *MSS* reenvía el evento al *DBS* y devuelve *USED* para contabilizar dicho evento.

A su vez, la política de *NetHandoff* para *DBS* recorre las transacciones locales a su *DBMS* y selecciona las que pertenecen al *MH* que inició el salto de celda, y que se encuentran en estado *RUNNING*. Para cada una de dichas transacciones, elimina su finalización de la cola de eventos y genera un evento *TxAbort* a ser atendido de inmediato. El *MH* es el destinatario final de dicho evento y el *DBS* actúa como receptor inicial.

NetDisconnectPolicy: Esta política es idéntica a la anterior. Desde el punto de vista del funcionamiento del evento, la única diferencia con el evento de salto de celda es que la notificación del *abort* sólo alcanzará al *MH* cuando se reestablezca la conexión, es decir luego del evento *NetDisconnect*.

Kangaroo

Kangaroo_BeginPolicy: Si el evento *TxBegin* corresponde a una *Kangaroo Transaction* (*KT*), se determina la cantidad de *Joey Transactions* (*JT*) necesarias[10], que son generadas como subtransacciones de la *KT* y dirigidas al *DBMS* Si el *begin* de transacción corresponde a una *JT*, únicamente se agrega a la lista de transacciones del *MSS*

En caso de que el *MSS* que atiende el evento no sea el actual debido a un salto de celda, el evento se retransmite al próximo *MSS* en la cadena y eventualmente alcanzará al actual.

Kangaroo_CommitPolicy: El evento de *commit* es atrapado en dos plataformas: en el servidor de base de datos (*DBS*) se realiza efectivamente salvo que sea el *commit* de la transacción principal (*KT*).

En el *MSS* se determina si el *commit* corresponde a una *JT* En caso afirmativo, si la *KT* está en estado *RUNNING* y no está pendiente, se decide mediante una distribución si deben generar nuevas *JT*s En este último caso se generan los eventos correspondientes utilizando sucesivos *TxBegin* como subtransacciones de la original, enviados a sí mismo, para el *DBMS*.

[10] Si se desea que la KT sea serializable este valor debería ser siempre 1 .

En caso de no generar mas subtransacciones (*JT*) y siempre y cuando no queden mas *JT*s activas de la *KT* padre pueden ocurrir a su vez dos casos:

Si el *MH* que dio origen a la *KT* original está conectado se envía un evento de *TxCommit* a la *KT* correspondiente al *MSS* actual y se setea *KT* que generó la premier transacción en estado *Pending Commit*.

En caso de que el *MH* que dio origen a la *KT* no esta conectado se decide abortar enviando un *TxAbort* a la *KT* actual y se pone la *KT* original en *PendingAbort*.

Kangaroo_AbortPolicy: El evento de *abort* es parecido al *commit* explicado en el inciso anterior. Al igual que en este último en el servidor de base de datos (*DBS*) se realiza efectivamente salvo que sea el *commit* de la transacción principal (*KT*).

En el *MSS* se discrimina si se trata de un *abort* para una *JT* perteneciente a una *KT* en estado *RUNNING* (y no *pending*) en cuyo caso se envía un evento de *TxAbort* para la *KT*, cuyo estado pasa a *PendingAbort.*

Si el *MSS* recibe un *abort* por política para la *KT* y el destinatario del *abort* no es el *MSS* que lo recibe se crea otro evento de *abort* para la *KT* pero destinado al *MSS* anterior (si es que existe).

Si el modelo funciona en modo compensador[11] se corren las transacciones compensatorias para todas las *JT*s del mismo *MSS* que estén en estado *COMMITED*. Además se abortan las *JT*s que estén *RUNNING*, eliminando el evento de fin original.

Si por el contrario, el modelo funciona en modo *split* se decide por distribución para cada *JT* en estado *COMMITED* si la va a compensar o no. En caso afirmativo se corre la transacción compensatoria.

IgnoreDisconnectPolicy: Kangaroo no implementa una polìtica propia de desconexión, sino que utiliza una polìtica predefinida que ignora el evento de desconexión. De no especificar el uso de esta polìtica, el simulador asumiría que implementa la política por defecto, NetDisconnectPolicy, que aborta todas las transacciones ante una desconexión.

Moflex

Moflex_BeginPolicy: El evento se modela únicamente en el *MSS*[12]. Si el evento corresponde a la transacción original se generan según una distribución las subtransacciones cuyas precondiciones se cumplan para ser ejecutadas a través de *TxBegin* como subtransacciones de la original, enviadas a sí mismo, para el *DBS*. Si es el *begin* de una subtransacción únicamente se la agrega a su lista de transacciones.

Moflex_CommitPolicy: El comportamiento en el *DBS* es similar al de Kangaroo.

En el *MSS* se realiza un comportamiento especial para las subtransacciones (no debería ser el *commit* de política generado por el salto de celda).

Si la transacción padre está en estado *RUNNING*, se decide mediante una distribución si se llegó a un estado aceptable en cuyo caso se envía un *TxCommit* a la transacción principal en caso de estar conectado o un evento de *TxAbort* a la transacción principal (que llegará cuando se reconecte) en caso de estar desconectado el *MH*. En ambos casos de finaliza la subtransacción.

En caso de no haber arribado a un estado aceptable se determina (mediante otra distribución) si quedan subtransacciones por lanzar. Para todas aquellas se repite un comportamiento similar al *TxBegin* explicado anteriormente.

Si no se arribó a un estado aceptable ni quedan transacciones por generar se envía un evento de *TxAbort* a la transacción principal y termina. (salvo que esta este no este *RUNNING* en cuyo caso se la deja seguir).

[11] Esto se define en el archivo de configuración.

[12] Recordar que se heredan las propiedades del modelo plano por lo que este evento tiene también su procesamiento en la DBMS

Moflex_AbortPolicy: En el *DBS* se repite una vez mas el mismo mecanismo que en el *commit*.

En el *MSS* el mecanismo es similar el descrito para el evento de *commit* determinándose de la misma manera según se haya arribado a un estado aceptable y queden (o no) subtransacciones por lanzar cómo se continua con el conjunto de subtransacciones disponibles.

Moflex_HandoffPolicy: El procesamiento ocurre en el *MSS*.

Para cada una de sus transacciones se aplica una probabilidad indicada como parámetro para definir su regla de salto de celda:

- Si es *Restart*: se envía un *TxAbort* para la subtransacción y un *TxBegin* para el nuevo *MSS*.y se elimina el evento de fin original (programado).
- Si es *Continue*: se pasa el contexto de la subtransacción al nuevo *MSS* y se le cambia el dueño.
- Si es *Split-Restart*: se dirige un *TxCommit* a la subtransacción y se vuelve a lanzar un *TxBegin* para la subtransacción en el nuevo *MSS*.además de eliminar el evento de fin original.
- Si es *Split-Resume*: se envía un *TxCommit* a la subtransacción y se vuelve a lanzar un *TxBegin* para la subtransacción en el nuevo *MSS*, con una longitud igual a lo que le restaba a la original. Nuevamente se elimina el evento de fin original.

IgnoreDisconnectPolicy: Al igual que Kangaroo, Moflex no implementa una polìtica propia de desconexión, sino que utiliza la polìtica predefinida que ignora el evento de desconexión.

IOT

IOT_BeginPolicy: En nuestra implementación de este modelo, decidimos que la transacción local del *MH* fuera la transacción principal. Sin embargo, el evento *TxBegin* fabricado por el *MH* tiene como destinatario al *DBS*, por lo que la primera transacción que se genera es la secundaria que se aloja en la base de datos central. Por lo tanto, la política de *TxBegin* para *MH* genera una nueva transacción local como *partner* de la global, para lo cual genera un nuevo evento *TxBegin* que tiene como emisor al *MH* y destinatario al *MH*, creando la *DBMS* local si no estaba previamente. Dado que el nuevo *TxBegin* también será atendido por esta política, estas las acciones sólo se llevan a cabo si el evento no fue generado por política, evitando así un ciclo infinito.

IOT_CommitPolicy: Como en todos los modelos que utilizan transacciones adicionales a la principal, esta política debe atender los sucesos de finalización de la transacción principal en el *MH* y la de su asociada en el *DBS*.

Para comprender esta política es necesario tener en cuenta el resto de las políticas de este modelo, ya que funcionan en estrecha colaboración y por lo tanto no puede ser considerada en forma aislada. Presentamos a continuación su funcionamiento en forma algorítmica:

En el *MH*:

Si el *MH* no está conectado:

Si se trata del evento de *commit* natural de la transacción local del *MH* (el generado por el *DBMS* del *MH* al atender el *TxBegin*), el evento es absorbido (no será enviado al *DBMS*) y la transacción pasa al estado "*PENDING_COMMIT*". Permanecerá en este estado hasta que el *DBS* envíe, luego de la reconexión, el evento de *commit* o de *abort*.

Si el *MH* está conectado:

Si es el *commit* natural de la transacción local del *MH*, el evento es bloqueado y se genera un nuevo *commit* para la transacción global asociada (alojada en el *DBS*).

Si es el *commit* enviado por el *DBS* (que puede ser por sincronización exitosa si estuvo desconectado) es aceptado y procesado.

En el *DBS*:

Si se trata del *commit* natural de la transacción global, es absorbido (no se envía al *MH* ni al *DBMS*).

Si es el *commit* enviado desde el *MH*:

Si no hubo desconexión (la transacción global está *RUNNING*) contesta con un *commit* para el *MH*.

Si hubo desconexión (la transacción fue abortada por desconexión o salto de celda) entiende que el evento es el resultado de la reconexión y que debe sincronizar la transacción con la base de datos global. Determina el resultado de la sincronización con una variable aleatoria y, según sea exitosa o no, envía un *commit* o un *abort* al *MH*.

IOT_AbortPolicy: Esta política es similar a la de *TxCommit*, pero en este caso no es necesario responder a una sincronización después de una reconexión, ya que no es necesario sincronizar un *abort*.

En el *MH*:

Si el *MH* no está conectado:

El *abort* natural de la transacción local queda como un *ABORT* pendiente..

Si se trata del evento de *commit* natural de la transacción local del *MH*, el evento es absorbido y la transacción pasa al estado "*PENDING_ABORT*".

Si el *MH* está conectado:

Si es el *abort* natural de la transacción local, el evento es bloqueado y se genera un nuevo *abort* para la transacción global.

Si es el *abort* enviado por el *DBS* (que puede ser por una sincronización fallida si estuvo desconectado), es aceptado y procesado.

En el *DBS*:

Si es el *abort* natural de la transacción global, es absorbido.

Si es el *abort* enviado desde el *MH* para la transacción global, responde con otro *abort* para la transacción local del *MH*.

Si es el *abort* causado por un salto de celda o una desconexión, deja que aborte la transacción global pero no reenvía el evento al *MH* (el salto de celda imita al evento de desconexión).

IOT_HandoffPolicy: Esta política simplemente absorbe el evento en el *MH* si éste no está conectado. El motivo es evitar que el evento *TxAbort* de la transacción global que genera la política predefinida *NetHandoffPolicy* sea interpretado por la política *IOT_AbortPolicy* como el resultado de la sincronización posterior a la reconexión.

IOT_ReconnectPolicy: Al reconectar debe revisar si, durante el tiempo que el *MH* estuvo desconectado, alguna transacción local terminó con *commit*. Si es así, debe sincronizar con la base de datos central. Para hacer esto, recorre todas las transacciones locales del *MH*. Si el estado de una transacción es "*PENDING_COMMIT*" o "*PENDING_ABORT*", manda el evento correspondiente (*TxCommit* o *TxAbort*) a la transacción global para que esta responda con el evento de finalización definitivo (que puede ser resultante de la sincronización la transacción local terminó exitosamente). Luego cambia el estado de la transacción a "*WAITING_COMMIT*" o "*WAITING_ABORT*", simplemente para que no vuelva a aplicar este algoritmo a las mismas transacciones si vuelve a ocurrir una desconexión y posterior reconexión.

WeakStrict

WeakStrict_BeginPolicy: El evento se procesa en el *MH*. El *MH* crea una base de datos local y le cambia el destinatario para que sea el *MH*.

Al momento de crear la transacción se determina mediante una distribución si todos sus datos están disponibles en le ambiente local en cuyo caso se categoriza a la transacción como débil. Si la transacción es estricta y el *MH* se encuentra conectado se crea un nuevo *partner* (a través del evento *TxBegin* correspondiente dirigido al *DBS)*. El modelo asume que si el *MH* esta desconectado y se desea realizar una transacción estricta esta se aborta, por lo tanto el simulador hace lo propio indicando el estado de la

transacción como pendiente de *ABORTED*, de modo que el DBMS, al recibir el evento, abortará la transacción inmediatamente.

WeakStrict_CommitPolicy: Esta política, al igual que la anterior, procesa el evento en el *MH.*Para las transacciones remotas (*partner*) y estrictas se genera el *TxCommit* de la transacción local.

Para las transacciones locales débiles conectadas se absorbe el evento, se cambia el estado a *PendingCommit* y se manda un evento de sincronización (*TxSynch*) al *DBS*. Mientras que para las no conectadas se limita a agregarlas a la lista de transacciones pendientes de sincronización.

Si el evento es el *commit* para una transacción local estricta, lo absorbe sólo en caso de que se trate del evento natural, generado por el *DBMS* al recibir el *begin*. En otro caso, es decir si es el *commit* enviado por el *DBS*, deja que el *DBMS* local lo trate para que el *commit* se haga efectivo (a través de la política *TxBeginPolicy* heredada del modelo *Flat*).

WeakStrict_AbortPolicy: El evento es tratado en el *MH* y tiene un comportamiento muy similar al *commit* explicado en el ítem anterior. Para las transacciones remotas (*partner*) y estrictas se genera el *TxAbort* de la transacción local. Si es la transacción es local y estricta se absorbe del mismo modo en que se explicó en el punto anterior.

WeakStrict_HandoffPolicy: Este evento, tratado en el *MH*, genera un *abort* para las transacciones locales de tipo *strict* al recibir el evento NetHandoff. Las transacciones de tipo *weak* siguen ejecutando localmente sin cambios.

WeakStrict_DisconnectPolicy: En el *MH* si la transacción es estricta se genera un *abort* para la local. No se transforma la transacción local en *débil* porque si inicialmente decidió que debía ser *estricta* es porque no tenía los datos en el cluster local (entendemos que el objetivo del modelo es usar *débil* siempre que se pueda). Por lo tanto, no funcionaría como *débil*.

En el *MSS* se utiliza el comportamiento heredado de *NetDisconnect*: se eliminan todas las transacciones del remotas que pertenecen al *MH* que se desconecta.

WeakStrict_ReconnectPolicy: En el *M.H* se genera un evento de sincronización y se envía al correspondiente *DBS*, donde el DBMS decidirá si la sincronización es exitosa o no según una variable aleatoria, luego de lo cual re-envía el evento a su origen con el resultado de la decisión.

WeakStrict_SynchronizePolicy: En el MH, si la sincronización resultó exitosa y si existe alguna transacción en estado *RUNNING* con *Pending Commit* se genera un *commit* local para dicha transacción.

En caso de fracaso y si existe una transacción que cumpla las mismas condiciones del caso anterior, se genera un *abort* local para dicha transacción.

Para cada transacción que se encuentre pendiente de sincronización se decide si se debe compensar (utilizando otra distribución de probabilidad). Al finalizar dicho ciclo se limpia la lista de pendientes de sincronización.

1.7 Formato del archivo de configuración

El archivo de configuración define los valores de los parámetros y de las variables aleatorias a ser usados en las simulaciones. Es un archivo de texto con el nombre "*simulator.ini*" que debe estar ubicado en el mismo directorio que el simulador. La instanciación de este archivo se detalla en el apartado correspondiente a la configuración (ver <2.2 Configuración>).

El archivo de configuración puede contener comentarios en cualquier lugar excepto en la misma línea en la que se define un parámetro o variable. Si bien no es necesario, se recomienda preceder los comentarios con un punto y coma, un asterisco, una doble barra o cualquier otro carácter que facilite su identificación. Las definiciones de parámetros y variables aleatorias deben comenzar al principio de una línea.

Parámetros

Los parámetros se definen de la siguiente manera:

```
<nombre> = <valor>
```

El valor puede ser numérico o alfanumérico. Un parámetro "*CurrentModel*" utiliza una sintaxis especial que permite indicar los nombres de todos los modelos que se desean simular en la misma corrida, usando el punto y coma como separador. Por ejemplo:

```
CurrentModel = Flat;IOT;Moflex;Kangaroo
```

Variables aleatorias

Las variables aleatorias se definen de la siguiente manera:

```
<nombre> = <distribución>
<nombre>_param1 = <valor>
<nombre>_param2 = <valor>
...
<nombre>_param<N> = <valor>
```

La cantidad de parámetros (el valor de *N*) depende de la distribución. Las distribuciones permitidas son las siguientes:

Distribuciones continuas:

`UniformDistribution`: 2 parámetros (límites inferior y superior)
`GaussianDistribution`: 2 parámetros (media y varianza)
`ExponentialDistribution`: 1 parámetro (frecuencia)
`PoissonDistribution`: 1 parámetro (frecuencia)

Distribuciones discretas:

`BernoulliDistribution`: 1 parámetro (probabilidad de éxito)
`BinomialDistribution`: 2 parámetros (repeticiones y probabilidad de éxito)
`GeometricDistribution`: 1 parámetro (probabilidad de éxito)

Adicionalmente definimos dos pseudo-distribuciones para comodidad del usuario:

`ConstantDistribution`: 1 parámetro (valor constante)
`MultiValueDistribution`: N parámetros (probabilidades relativas)

La distribución "*Constant*" permite fijar el valor de la variable aleatoria (en rigor deja de ser aleatoria) y es generalmente utilizada para realizar pruebas.

La distribución "*MultiValue*" define una variable aleatoria que tomará los valores discretos 1..N, con frecuencias relativas definidas en los parámetros. La frecuencia relativa que define cada parámetro se calcula en referencia a la suma de los parámetros. Por ejemplo, si los parámetros son 3 y 2, el valor "1" ocurrirá 3 de cada 5 veces y el valor "2" ocurrirá 2 de cada 5 veces, y es equivalente a definir los parámetros 6 y 4. Como esta distribución tiene una cantidad variable de parámetros, es necesario indicar explícitamente el número de parámetros de la siguiente forma:

```
<nombre> = MultiValueDistribution
<nombre>_params = <N>
<nombre>_param1 = <valor>
<nombre>_param2 = <valor>
...
<nombre>_param<N> = <valor>
```

Variables iteradas

Las variables iteradas se definen de la siguiente manera:

```
<nombre> = <valor inicial>;<cantidad de iteraciones>;<fórmula>
```

Los sucesivos valores que debe tomar la variable en cada iteración, luego del valor inicial, se calculan utilizando la fórmula dada. Esta fórmula, en notación polaca inversa, calcula el siguiente valor de la variable a partir del anterior. El cálculo se basa en el uso de una pila de valores, sobre los cuales se aplican los operadores. La pila contiene inicialmente el valor de la variable en la iteración anterior. Los operadores permitidos en la fórmula son los siguientes (para la explicación se asume que la pila contiene los valores **`[a,b,c]`**, siendo ***c*** el tope de la pila):

`'+'`: extrae b y c de la pila y deja el resultado de la suma, quedando **`[a,b+c]`**
`'-'`: extrae b y c de la pila y deja el resultado de la resta, quedando **`[a,b-c]`**
`'*'`: extrae b y c de la pila y deja el resultado de la multiplicación, quedando **`[a,b*c]`**
`'/'`: extrae b y c de la pila y deja el resultado de la división, quedando **`[a,b/c]`**
`'^'`: extrae b y c de la pila y deja el resultado de la potencia, quedando **`[a,b^c]`**
`'d'`: duplica el tope de la pila, quedando **`[a,b,c,c]`**
`'s'`: intercambia los dos últimos valores de la pila, quedando **`[a,c,b]`**

Al final del cálculo, la pila debe contener un único valor, que será el valor de la variable en la siguiente iteración. En la Tabla 18 se muestran ejemplos de definiciones de variables iteradas. La fórmula se muestra en notación infija y usa el símbolo X para representar el valor de la variable en la iteración anterior.

Definición	Fórmula (*)	Valores resultantes
`iterador = 0;6;1+`	`X + 1`	`0, 1, 2, 3, 4, 5`
`iterador = 10;5;2/`		`10, 5, 2.5, 1.25, 0.625`
`iterador = 2;5;10*5+`	`10 X + 5`	`2, 25, 255, 2555, 25555`
`iterador = 1;4;d2^+`		`1, 2, 6, 42`
`iterador = 0.1;4;1s/10+1s/`		`1/10, 1/20, 1/30, 1/40`

Tabla 18: Ejemplos de iteradores

1.8 Interfaz de usuario

Para facilitar el uso del simulador, si bien puede usarse directamente desde la línea de comandos, desarrollamos una pequeña interfaz de usuario utilizando las facilidades de Access para tal fin. La pantalla principal de la interfaz puede verse en la Figura 17.

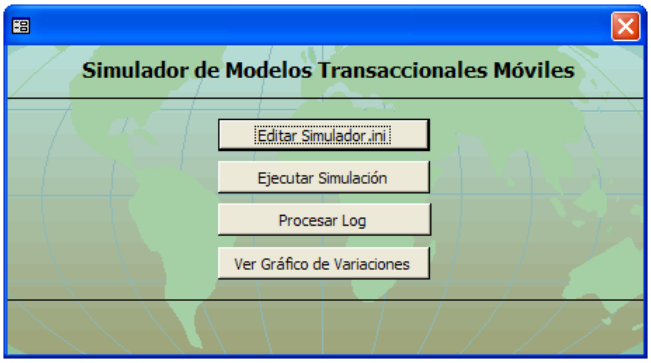


Figura 17: Pantalla inicial

El primer botón simplemente abre el archivo de configuración con el anotador de Windows (notepad).

El segundo botón ejecuta el simulador y muestra el avance de la ejecución con una barra en la parte inferior de la ventana, como puede observarse en la Figura 18.

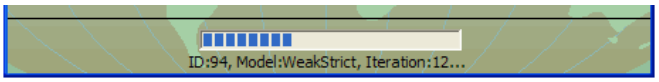


Figura 18: Progreso de la ejecución del simulador

El tercer botón inicia el análisis del log generado por el simulador, mientras muestra el avance del análisis con una barra similar a la anterior, como se ve en la Figura 19.

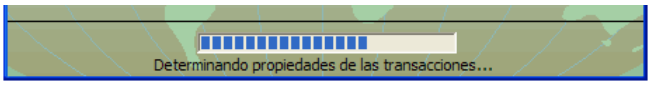


Figura 19: Progreso del análisis del log

El cuarto y último botón abre una ventana con los gráficos resultantes del análisis. Estos pueden imprimirse o exportarse. Los gráficos presentados en este trabajo fueron exportados al formato "*snp*", propietario de Access, que si bien es el único formato al que Access puede exportar sin perder los gráficos, no es el ideal debido a que es un "*bitmap*" con resolución apropiada para su presentación en pantalla pero insuficiente para su impresión en medios de alta calidad como una impresora láser.

Apéndice 2: Guía de instalación y configuración

En este apéndice explicaremos brevemente los componentes del simulador y la secuencia de pasos necesarios para ponerlo en marcha.

2.1 Instalación

1. Instalar el ambiente Runtime de Java

   El entorno de ejecución (runtime) de Java es indispensable para ejecutar el Simulador. El CD provisto con esta tesis contiene la versión 1.3.1 del Java 2 Standard Edition, o bien puede descargarse en forma gratuita desde el sitio web de Sun Microsystems (http://java.sun.com). Para instalarlo, alcanza con seguir las instrucciones incluidas en el producto y verificar que el directorio *bin* que contiene las herramientas de Java, en especial el intérprete *java.exe*, esté en el *path* del sistema.

2. Instalar Microsoft Access

   El motor de base de datos utilizado para el Analizador es parte integral de Microsoft Office y puede encontrarse en la mayoría de las instalaciones de Windows. La versión usada para este trabajo es MS-Access 2000 y es la que recomendamos para evitar problemas de compatibilidad. Una vez instalado, es necesario verificar las referencias. Para hacer esta verificación, entrar al editor de código (abriendo un módulo o creando uno nuevo), seleccionar el ítem *referencias* del menú *herramientas* y controlar que las siguientes referencias estén marcadas:

   - Visual Basic For Applications
   - Microsoft Access 9.0 Object Library
   - OLE Automation
   - Microsoft ActiveX Data Objects 2.1 Library
   - Microsoft DAO 2.5/3.5 Compatibility Library
   - Microsoft Visual Basic For Applications Extensibility 5.3
   - EuGraph ActiveX Control module
   - Microsoft Windows Common Controls 6.0 (SP4)

   Recomendamos también instalar el visualizador de archivos ("*Snapshot Viewer*") de Microsoft, disponible gratuitamente en el sitio web de Microsoft (http://www.microsoft.com). Este programa permite visualizar los archivos *.snp* incluidos en la instalación, generados por el Analizador.

3. Instalar el Simulador

   Para instalar el simulador se debe copiar del CD el directorio ModMov al directorio raíz del disco C. En la Tabla 19 detallamos el contenido de los principales directorios del Simulador:

Directorio / Archivo	Descripción
`C:\ModMov\ModMov.mdb`	Programa principal, desde el que se ejecutan las simulaciones y se analizan y visualizan los resultados
`C:\ModMov\Simulador\Simulador.jar`	Programa simulador en formato empaquetado de Java
`C:\ModMov\Simulador\simulador.ini`	Archivo de configuración del simulador
`C:\ModMov\Simulador\50megs`	Archivo de bits aleatorios (opcional)
`C:\ModMov\Simulador\Fuentes\`	Directorio de fuentes del simulador
`C:\ModMov\Simulador\Fuentes\make.bat`	Script para generar el Simulador.jar
`C:\ModMov\Simulador\Fuentes\modelo.mdl`	Modelo de clases del simulador, en formato Rational Rose 98, incluido en el CD
`C:\ModMov\Aplicaciones\`	Directorio con los archivos de configuración y los gráficos resultantes para cada una de las aplicaciones estudiadas en este trabajo

Tabla 19: Principales directorios del simulador

2.2 Configuración

Una vez instalados todos los componentes, es necesario configurar el simulador. En el directorio de aplicaciones proveemos los archivos de configuración de las aplicaciones estudiadas en este trabajo, y son un buen ejemplo para analizar y comprender sus posibilidades. En la Tabla 20 detallamos el significado de cada parámetro del archivo de configuración, dejando de lado únicamente los parámetros específicos de los modelos implementados en este trabajo.

Definición de nuevos modelos

Si bien para implementar un modelo nuevo alcanza con definir las políticas correspondientes, no es una tarea trivial. Es fundamental tener un conocimiento cabal de las posibilidades y limitaciones del simulador. Antes de codificar un nuevo modelo es recomendable escribir en pseudocódigo las políticas y su interacción con el simulador porque, dado lo compleja que puede resultar dicha interacción, es muy fácil perder el hilo. En la Tabla 20 detallamos el procedimiento a seguir para diseñar y probar un nuevo modelo.

1. Para cada evento se debe decidir qué hará cada elemento que lo recibe (*MH*, *MSS*, *DBS*, *DBMS*).

2. Si el tratamiento que se desea difiere del comportamiento por defecto, se debe implementar una política. La política siempre debe subclasificar la política por defecto para el tipo de evento, nunca se debe subclasificar directamente EventPolicy.

3. Cuando se programa una política hace falta tener en cuenta los siguientes factores:
   - Quién produjo el evento
   - A quién va dirigido el evento
   - Si fue producido por política o no
   - Si la Transacción pertenece al *DBS* local o no
   - El estado de la Transacción
   - El *pendingState* de la Transacción
   - Si se trata de una transacción principal o de una subtransacción
   - El estado de las transacciones relacionadas (*parent*, *partner*, etc)
   - Si el *MH* está conectado
   - Cuando se trata de una política de *MSS*, si está en el *MSS* correcto (puede ser uno de una cadena). Esto se puede verificar con el destinatario del evento (si el destinatario es el *MSS*) o viendo si está conectado al *MH*
   - Si hay que devolver NO_USED, USED, NO_FORWARD y/o NO_DBMS (empezar con NO_USED y ver cómo contabiliza)

   Cada uno de estos factores puede derivar en una condición que debe ser verificada en política para tomar diferentes acciones.

4. Dado que los eventos que genera una política también serán atendidos por las políticas (posiblemente la misma que lo generó), es necesario repasar las condiciones del punto anterior para cada nuevo evento.

5. Someter el modelo a situaciones sucesivamente más complejas. A medida que se aumenta la complejidad pueden ir apareciendo errores, generalmente detectados por el algún *assert*, o por resultados inexplicables en las estadísticas. Un conjunto de condiciones de prueba fue presentado en el apartado <5.6 Validación y verificación del simulador>.

Clave	Significado
CurrentModel	Lista de modelos a utilizar en la simulación.
Repetitions	Cantidad de repeticiones de las simulación. Las repeticiones se promedian, por lo que aumentando este número, las curvas resultantes deberían ser más suaves.
SeqRunID	Número de secuencia de las simulaciones, de uso interno.
ConcatenateLog	Indica si se deben concatenar los logs de las distintas simulaciones. Normalmente debe estar en *true*.
ReproduceSimulation	Indica si debe reproducir una simulación a partir del log de otra.
ReproduceSimulationLogFile	Ubicación del archivo log de la simulación que debe reproducirse (opcional)
UseExternalRandomGenerator	Indica si los números aleatorios se obtienen de un archivo externo o con el generador pseudo-aleatorio de Java.
RandomNumberFileName	Ubicación del archivo de bits aleatorios (opcional)
CheckAssertions	Indica si se controlan las invariantes. Normalmente en *true*.
ReportTime	Indica si se reporta el tick de la simulación. Normalmente en *false*.
StatsDisplayInterval	Cantidad de eventos entre sucesivos reportes de estadísticas. Normalmente 50.
LogFilter	Tipos de eventos que deben ser reportados en el archivo de log.
NumberOfDatabases	Cantidad de servidores de bases de datos (*DBSs*) simulados (distribución). Normalmente 1.
NetNewMH_Distribution	Cantidad de elementos móviles (*MHs*) a simular (distribución).
NetNewMH_Delay_Distribution	Tiempo entre creaciones de MHs en milisegundos (distribución).
NetDisconnect_Distribution	Frecuencia de desconexión en 1/ms (distribución).
NetReconnectDelay	Tiempo entre desconexion y reconexión en ms (distribución).
NetHandOff_Distribution	Tiempo entre saltos de celda en ms (distribución).
NetLinkDelay_Radio_Symetrical	Indica si la latencia de transmisión por radio es la misma en ambas direcciones (del MH al MSS y del MSS al MH).
NetLinkDelay_Radio_MHtoMSS	Latencia de transmisión del MH al MSS (distribución).
NetLinkDelay_Radio_MSStoMH	Latencia de transmisión del MSS al MH (distribución).
NetLinkDelay_Wire	Latencia de transmisión en la red fija (distribución).
MH_TransactionsPerMH	Cantidad de Transacciones que lanza un MH (distribución).
DBMS_AccessesPerTx	Cantidad de accesos a la base por transacción (distribución)
DBMS_MaxNumberOfSubTx	Cantidad de subtransacciones por transacción (distribución).
DBMS_TxSuperpositionFactor	Factor de superposicion de dos Tx sucesivas (distribución).
DBMS_TimeBetweenAccesses	Tiempo entre accesos a la base, en ms (distribución).
DBMS_AccessesProcessingTime	Tiempo de procesamiento de cada acceso en la base, en ms (distribución).
DBMS_txSuccess	Probabilidad de éxito de una transacción (distribución).
DBMS_txLockProbability	Probabilidad de que se produzca un boqueo habiendo exactamente dos transacciones activas (distribución).
DBMS_txLockLenProportion	Relación entre la duración de un bloqueo con respecto a la duración de la transacción.
DBMS_txLockLenPopulationFactor	Factor que determina cuánto influye la cantidad de transacciones activas en la longitud de un bloqueo.
DBMS_txLockTimePopulationFactor	Factor que determina cuánto influye la cantidad de transacciones activas en la determinación del momento de inicio de un bloqueo.
DBMS_txLockTimeOut	Tiempo máximo que una transacción puede estar bloqueada antes de que aborte, en milisegundos.
DBMS_SynchSuccess	Probabilidad de éxito de una sincronización de cambios en el DBS en modelos que soportan transacciones locales al MH (distribución).
DBMS_TxSynchOverhead	Tiempo que lleva sincronizar, en milisegundos (distribución).
DBMS_TxTransferOverhead	Tiempo que lleva transferir una transacción entre dos MSS, en milisegundos (distribución).

Tabla 20: Parámetros de configuración

Reproducción de simulaciones

Para reproducir una simulación es necesario hacer dos corridas: la primera genera un log de referencia, la segunda lo utiliza para reproducir dicho log con otros modelos. A continuación presentamos una guía que indica cómo modificar los parámetros de configuración para reproducir simulaciones:

Para grabar la corrida de referencia:

1. CurrentModel = Flat
2. LogFileName = c:\ModMov\Simulador\logr.txt
3. ReproduceSimulation = false
4. LogFilter = 1

Para reproducir la corrida de referencia:

1. CurrentModel = *(todos)*
2. LogFileName = c:\ModMov\Simulador\log.txt
3. ReproduceSimulation = true
4. LogFilter = 127

Apéndice 3: Glosario

A continuación realizamos una enumeración de términos frecuentemente utilizados en esta tesis con la descripción de su significado. Es de notar que existe una gran cantidad de sinónimos utilizados en la bibliografía leída (especialmente en lo que respecta a las definiciones de entornos móviles) por lo que este capítulo tendrá también la intención de unificar los nombres utilizados.

3.1 Definiciones de bases de datos

- ***Base de datos***: Representación de un estado de un sistema abstracto (ver <2.1 Sistemas de procesamiento de transacciones>).
- ***Deadlock:*** Situación que ocurre cuando dos o más transacciones esperan indefinidamente la liberación de recursos que ellas mismas bloquean. Por ejemplo, una transacción necesita un recurso que está tomado por otra, que a su vez espera otro recurso tomado por la primera.
- ***Commit:*** decisión de hacer visibles para el mundo exterior las actualizaciones de una transacción. Cuando una transacción hace *commit* todos sus efectos son publicados en forma durable. Después de un *commit* los cambios de un transacción no pueden ser revertidos de manera automática.
- ***D.B.M.S (Database Management System):*** Motor de Base de datos. Entidad encargada del acceso y la administracion de las tablas y otros objetos de la base de datos.
- ***LLT (Long Lived Transactions):*** Transacciones que se espera duren tiempos apreciables (por ejemplo una actualización masiva).
- ***Lock (Bloqueo):*** Mecanismo de semaforiazación para obtener aislamiento entre transacciones.
- ***Modelos transaccionales tradicionales***: Son aquellos modelos que respetan las propiedades *ACID*
- ***Propiedades ACID:*** Propiedades que cumplen las transacciones tradicionales: Atomicidad, Consistemcia, aislamiento (Isolation), Durabilidad
- ***Rollback:*** (Sinónimo: *abort*) Terminar un transacción deshaciendo todas sus acciones.
- ***Constraint:*** restricción de integridad impuesta por el DBMS.
- ***SQL*** (Structured Query Language): lenguaje estándar de consulta y manipulación de datos sobre un DBMS.
- ***Sistema de procesamiento de transacciones (TP System)****:* sistema que provee herramientas para facilitar o automatizar tareas de programación, ejecución y administración de bases de datos.
- ***Transacción***: es una transformación de la base de datos, realizada por la ejecución de un programa (ver <1.3 Transacciones y las propiedades ACID>).

3.2 Definiciones de entornos móviles

La explicación comprehensiva de muchos de estos conceptos así como las aclaraciones con respecto a su interacción se desarrollan en <>

- ***Celda:*** (Sinónimo**:** ***célula***) Área geográfica controlada por un *MSS* en la que se mueven los componentes móviles.
- ***Handoff:*** (Sinónimo: *salto de celda*) Evento que ocurre cuando un móvil (*MH*) se desplaza de una celda a otra y en consecuencia pasa a ser atendido por un *MSS* diferente.
- ***MH (Mobile Host):*** Componente móvil cuya comunicación con la red fija se realiza a través del *MSS* que lo atiende.
- ***MSS (Mobile Support Station):*** (Sinónimo: *Base Station,* Sinónimo *Home Agent*) Estación encargada de mantener un vínculo entre los componentes móviles y la red fija.
- ***Red Fija***: Parte de la red interconectada por medio de cables.

- ***Roaming***: Acuerdo entre operadoras que permite al usuario de un teléfono móvil utilizarlo en una red celular fuera de la cobertura de la red a la que pertenece. Permite realizar llamadas con el mismo teléfono desde cualquier país.
- ***Location-dependent data:*** (Datos Dependientes de la ubicación) Resultados de una consulta que dejan de tener sentido si el componente móvil modifica su ubicación geográfica. [Re99]

3.3 Definiciones de simulación

- ***Estado***: Conjunto de variables que caracterizan el estado del sistema simulado. Por ejemplo la posición de un móvil, la cantidad de transacciones de un *MH* etc.
- ***Evento***: Una ocurrencia en algún punto del tiempo que puede desencadenar un cambio en el estado del sistema. Por ejemplo un salto de celda, un *abort* de transacción etc.
- ***Entidad***: Un objeto simulado dentro del sistema, que tiene un estado y un comportamiento. Por ejemplo un MH, una transacción, etc.
- ***Cola de eventos***: Lista ordenada por tiempo de eventos a la espera de ocurrir. Un simulador toma de esta cola el siguiente evento e inserta en la misma los nuevos eventos a medida que son programados por las entidades. Una cola de simulación no necesariamente representa una cola en el ambiente real (los *MH*s no se encolan para pedir un ingreso a una celda). Puede ser una lista de acciones a seguir, una serie de recursos listos para su utilización, los eventos que utiliza el etc..
- ***Creación***: La creación en un ambiente de simulación representa el arribo de una nueva entidad al sistema en algún punto en el tiempo (por ejemplo la aparición de un nuevo *MH* o un split que genera la aparición de una nueva subtransacción)
- ***Scheduler***: Administrador de la cola de eventos. Selecciona el próximo evento a ocurrir y permite al resto del simulador agregar nuevos eventos o manipular los eventos ya programados, por ejemplo para eliminar un evento, desplazarlo en el tiempo, etc.
- ***Variable aleatoria***: Representa un valor desconocido pero cuyo valor sigue una distribución de probabilidad conocida. Por ejemplo el tiempo de permanencia de un *MH* dentro de una celda, la cantidad de transacciones en un *MH*, etc.
- ***Variación aleatoria***: Consiste en la generación artificial de una variable aleatoria utilizando números aleatorios o pseudo-aleatorios y siguiendo alguna distribución de probabilidad. Por ejemplo la ocurrencia de un evento como el salto de celda puede estar dada por una variable con una distribución de Poisson.
- ***Distribución de probabilidad***: Es una función matemática que determina la probabilidad relativa de cada posible valor de una variable aleatoria.
- ***Reacción ante eventos***: Un simulador define un conjunto de acciones que se ejecutan como consecuencia del disparo de algún evento. Por ejemplo un salto de celda puede desencadenar un *split* de la transacción activa.

3.4 Otras definiciones

- ***Broadcast***: Transmisión de datos en las que el emisor envía un mensaje y todos los receptores lo reciben aunque no sean el destinatario del mismo, quedando a criterio de los receptores determinar si deben tomar en cuenta el mensaje o no.
- ***Ancho de banda:*** Característica de los lazos de comunicación que define el volumen de datos que pueden ser transmitidos por unidad de tiempo. La medida típica utilizada es *Mb/s*.
- ***PDA (Personal digital assistant):*** computadora personal móvil.

Apéndice 4: Prueba de aleatoriedad

Para complementar las pruebas de aleatoriedad que aplicamos a nuestra fuente de números aleatorios, desarrollamos nuestra propia versión de la prueba de Chi-cuadrado.

Esta prueba consiste en dividir el intervalo (0,1) en n subintervalos para luego comparar para cada subintervalo la frecuencia esperada con al frecuencia observada. Si estas frecuencias son bastante parecidas, entonces la muestra proviene de una distribución uniforme. El estadístico que se usa en esta prueba es X^2_0 el cual se obtiene de acuerdo a la siguiente expresión:

$X^2_0 = \Sigma_{i=1..n}$ (Frecuencia Observada $_i$ – Frecuencia Esperada $_i)^2$ / Frecuencia Esperada $_i$

donde Frecuencia Esperada $_i$ = Tamaño de la muestra / número de subintervalos

Este estadístico se compara con $X^2_{\alpha,n-1}$ la cual representa a una variable aleatoria Chi-Cuadrado con n-1 grados de libertad y un nivel de confianza de α. Si $X^2_0 < X^2_{\alpha,n-1}$ entonces no se puede rechazar la hipótesis de que la muestra proviene de una distribución uniforme.

Generadores aleatorios evaluados

Utilizando el método descrito arriba evaluamos tres generadores de números aleatorios distintos con un nivel de confianza α = 0,05 % obteniendo los resultados detallados en la Tabla 21.

Generador	N (tamaño de la muestra	N (cantidad de subintervalos)	X^2_0	$X^2_{\alpha,n-1}$	¿Cumple con la Hipótesis?
Java	2000	100	112,9	123,23	Si
Java	2000	200	194,6	232,91	Si
Java	2000	500	200	552,07	Si
Microsoft Excel	2000	100	82,6	123,23	Si
Microsoft Excel	2000	200	211,6	232,91	Si
Microsoft Excel	2000	500	195,75	552,07	Si
Random.org	2000	100	86,7	123,23	Si
Random.org	2000	200	184,4	232,91	Si
Random.org	2000	500	250,5	552,07	Si

Tabla 21: Generadores de números aleatorios

Nota: Para determinar si cumple con la hipótesis se utilizó una tolerancia del 10%

De los resultados arriba expuestos se desprende que no se puede rechazar la hipótesis de que los numeros pseudo-aleatorios generados por las tres aplicaciones provienen de una distribución uniforme.

Implementación del algoritmo de prueba de aleatoriedad:

```
final long tamaño = 200000;
final int cantSubintervalos = 100;
final int tamanioSubintervalo = (int) (tamanio / cantSubintervalos);
double suma = 0;

double[] frecuenciaObservada = new double[cantSubintervalos];
for(int i = 1; i < tamanio;i++)
  (frecuenciaObservada[((int) (Math.random()*tamanio))/tamanioSubintervalo]) ++;

for(int i = 0; i < cantSubintervalos; i++)
  suma += (Math.pow((frecuenciaObservada[i] - tamanioSubintervalo),2)/ tamanioSubintervalo);
```