DEPURACIÓN Y EVOLUCIÓN DE APLICACIONES DISTRIBUIDAS Y CONCURRENTES USANDO UN MODELO DE EVENTOS BASADO EN AUTÓMATAS CAUSALES SÍNCRONOS Y ASÍNCRONOS

PROYECTO DE GRADO

DAVID DURÁN GIRALDO
JHONNY ANDRÉS OCAMPO

Asesor
HUGO ARBOLEDA Ph.D.

DANIEL BENAVIDES Ph.D.

FACULTAD DE INGENIERÍA
DEPARTAMENTO ACADÉMICO DE TECNOLOGÍAS DE INFORMACIÓN Y COMUNICACIONES
INGENIERÍA DE SISTEMAS
SANTIAGO DE CALI
2012

DEPURACIÓN Y EVOLUCIÓN DE APLICACIONES DISTRIBUIDAS Y CONCURRENTES USANDO UN MODELO DE EVENTOS BASADO EN AUTÓMATAS CAUSALES SÍNCRONOS Y ASÍNCRONOS

DAVID DURÁN GIRALDO
JHONNY ANDRÉS OCAMPO

Trabajo de grado para optar al título de
Ingeniero de Sistemas

Asesor
HUGO ARBOLEDA Ph.D.

DANIEL BENAVIDES Ph.D.

FACULTAD DE INGENIERÍA
DEPARTAMENTO ACADÉMICO DE TECNOLOGÍAS DE INFORMACIÓN Y COMUNICACIONES
INGENIERÍA DE SISTEMAS
SANTIAGO DE CALI
2012

Nota de Aceptación

_____
_____
_____
_____
_____
_____

_____
Firma del Presidente del Jurado

_____
Firma del Jurado

_____
Firma del Jurado

Santiago de Cali, <Fecha>

Tabla de Contenido

Tabla de Figuras

Resumen

Este documento presenta una propuesta de un lenguaje que será la base para un *framework* de eventos que soporta la detección de patrones complejos en sistemas distribuidos, utilizando autómatas para modelar los patrones complejos de interacción entre los nodos que participan en el sistema distribuido. Por medio de la presentación de diferentes errores concurrentes comunes que ocurren o han ocurrido en aplicaciones industriales de esta índole, como el *deadlock* o los *dataraces*, se proponen soluciones a este tipo de inconvenientes utilizando el lenguaje propuesto, demostrando su utilidad y aplicabilidad. En concreto, se presentan las siguientes contribuciones: *i)* el diseño de un lenguaje de programación orientado a eventos con soporte para declaración, ejecución, detección y coordinación de patrones de eventos complejos en sistemas distribuidos, *ii)* propuestas de definición de autómatas utilizando el lenguaje propuesto para detectar los errores concurrentes comunes identificados en aplicaciones distribuidas industriales y *iii)* la implementación de un *kernel* para soportar las abstracciones del lenguaje por medio de una extensión a la librería KETAL, la cual define mecanismos de sincronización de eventos.

Introducción

Contexto

Un sistema distribuido de acuerdo a [1] y a [2], consiste en una colección de computadores autónomos llamados nodos o hosts que están conectados a través de una red y un middleware de distribución, los cuales permiten a los nodos coordinar sus actividades y compartir los recursos del sistema, de modo que los usuarios perciban el sistema como uno solo (un único computador). Ejemplos de estos sistemas son los llamados sistemas intensivos de software, los cuales han tenido un impacto bastante grande en la sociedad en los últimos años. De acuerdo al informe "Tecnologías software orientadas a servicios" elaborado por la plataforma INES (Iniciativa Española de Software y Servicios) [3], estos sistemas dominarán el mercado informático en las próximas décadas. La mayoría de estas aplicaciones son sistemas distribuidos que han evolucionado y aumentado su complejidad a medida que avanzan las tecnologías de computación y comunicaciones. Servicios provistos sobre plataformas móviles y sobre la nube (internet), han implicado el desarrollo de tecnologías capaces de soportar la complejidad resultante de las comunicaciones e interacciones generadas entre los componentes que permiten implementar este tipo de servicios. Para manejar la complejidad de estas aplicaciones se han propuestos catálogos de mejores prácticas y estrategias de implementación; por ejemplo, se han propuesto los patrones de diseño en [4] y [5], y los patrones de integración sobre middleware de mensajería [6]. Sin embargo, Investigaciones recientes han demostrado que el uso de estas técnicas no reduce la complejidad de código resultante, como se puede observar en [7]; donde la implementación de patrones complejos de comunicación en granjas de servidores de aplicaciones JEE resulta en código confuso y disperso, difícil de entender y mantener.

Como resultados de esta complejidad, las actividades de depuración y evolución del sistema se hacen muy complicadas. Considere, por ejemplo, errores comunes en las aplicaciones concurrentes como el *deadlock* y los *data races* o condiciones de carrera. Estos problemas se ven exacerbados en las aplicaciones distribuidas que se caracterizan por utilizar varios hilos para su funcionamiento, y que además, generan crecimientos exponenciales en los hilos utilizados para atender múltiples usuarios concurrentes. A pesar de que esta complejidad es conocida, las herramientas actuales no proponen nuevas abstracciones para atacar este inconveniente. En la actualidad, la mayoría de depuradores distribuidos manejan puntos de interrupción simples que son notificados a una entidad centralizada que se encarga de detener la ejecución del programa. Estas herramientas no logran identificar la serie de eventos que produjeron el error y que se encuentran dispersos en los diferentes nodos de la aplicación distribuida. Si deseamos

detectar los problemas que ocurren en una aplicación distribuida, es necesario poder detectar patrones complejos de eventos. Dentro de los enfoques que proponen herramientas y mecanismos para tratar con la definición, coordinación, detección y ejecución de patrones de eventos complejos se encuentra EventJava [8], donde se propone un modelo de eventos para tratar la interacción por medio de eventos entre sistemas distribuidos. Aunque este enfoque propone una solución bastante completa, carece de mecanismos para la sincronización de eventos (entendiendo por sincronización la capacidad de esperar por la respuesta o resultado del proceso de uno o más nodos para realizar una tarea), además de la falta de retornos en *broadcast* de eventos y de control de excepciones en ambientes distribuidos.

Sumado a los inconvenientes anteriores se debe lidiar con la administración de eventos, la cual es un reto que deben enfrentar diseñadores y programadores de sistemas distribuidos. La generación y recepción de mensajes entre los nodos de un sistema distribuido se conoce como eventos. Un evento representa el cambio de estado en un componente específico de un sistema. El uso de estos en los programas de computador son muy comunes; modernos lenguajes y frameworks proponen un modelo de programación orientado a eventos para el GUI (Graphical User Interface, por sus siglas en inglés). Un ejemplo es SWING API en Java y Java Server Faces en JEE. Un evento simple [9] consiste en enviar o recibir un único mensaje que viaja a través del sistema; un evento complejo [9] es entonces la unión de varios eventos simples. Otro concepto importante es el que se refiere a patrones de eventos distribuidos, los cuales son secuencias de eventos complejos de interés que pueden ocurrir en condiciones específicas del sistema y que permiten analizar el comportamiento de este. La detección de estos patrones no suele ser sencilla, debido a que factores como el retraso y la congestión de la red afectan el orden de llegada de dichos mensajes; lo que en últimas afecta al patrón de llegada esperado y puede generar acciones no deseadas en el sistema [10].

Problema

Actualmente, las interacciones de múltiples componentes distribuidos y la presencia de múltiples procesos concurrentes hacen la programación de aplicaciones distribuidas una tarea muy compleja. El código resultante en la implementación de estas aplicaciones es difícil de depurar y evolucionar. La programación orientada a eventos se ha presentado cómo un paradigma idóneo para el desarrollo de estas aplicaciones, sin embargo, varios trabajos han mostrado las limitaciones de las herramientas actuales. En particular, se ha mostrado que estas no reducen la complejidad del código resultante ni proponen nuevas abstracciones para atacar estos inconvenientes.

Contribución General

En este trabajo estudiamos la programación orientada a eventos como abstracción fundamental para la implementación de aplicaciones distribuidas y concurrentes. Estudiamos en la literatura las limitaciones de las herramientas actuales, revisando concretamente los problemas y errores de concurrencia encontrados en aplicaciones distribuidas. Finalmente, proponemos nuevas abstracciones que complementan la programación orientada a eventos, facilitando la depuración y evolución de estos sistemas. En particular, proponemos un modelo de detección y manipulación de eventos complejos basado en autómatas deterministas, con soporte para el ordenamiento causal de eventos y para el manejo síncrono y asíncrono de interacciones entre los eventos.

Contribuciones Específicas

1. Estudiar el estado del arte en la programación orientada a eventos para aplicaciones distribuidas.

2. Estudiar como modelar los problemas de concurrencia de *Liveness* y *Data Races* en sistemas distribuidos y concurrentes por medio de autómatas finitos deterministas.

3. Diseñar un lenguaje de programación orientado a eventos con soporte para declaración, ejecución, detección y coordinación de patrones de eventos complejos en sistemas distribuidos utilizando autómatas, ordenamiento de eventos basados en la causalidad y la composición síncrona y asíncrona de eventos.

4. Mostrar como se puede utilizar este lenguaje para detectar y depurar problemas de concurrencia en JBoss Cache, una herramienta de software distribuida y concurrente.

5. Presentar la implementación de un kernel para soportar las abstracciones del lenguaje, por medio de una extensión a la librería KETAL [10].

Estructura del Documento

Este documento busca motivar y explicar el tipo de investigación asociada con este desarrollo de la siguiente manera. Como primera instancia se enuncian el problema, el objetivo general y los objetivos específicos. Luego, en la sección del

marco teórico se muestra la caracterización de un modelo de eventos y un ejemplo concreto de esto, conocido como EventJava. Además se tratan los diferentes tipos de errores de concurrencia que se pueden encontrar en el middleware distribuido disponible actualmente en las soluciones empresariales. Concretamente, se muestra como los errores de *deadlock*, problemas de *liveness* y *data races* ocurren en Jboss Cache, el cual es un módulo esencial en el servidor de aplicaciones JBoss. Luego se presentan diferentes aproximaciones sobre cómo modelar concurrencia por medio de autómatas y se resaltan las similitudes y diferencias de estas con nuestro enfoque. En la siguiente sección se discuten las características que presenta el lenguaje propuesto para permitir la definición, coordinación, detección y ejecución de patrones de eventos complejos. En seguida se presentan propuestas implementadas con el lenguaje que permiten la detección de los problemas tratados anteriormente; luego se presenta una implementación de un *kernel* para soportar las abstracciones del lenguaje por medio de una extensión a la librería KETAL [10], concretamente se describen los métodos que permiten la sincronía en la librería y por último, se presentan las conclusiones.

Marco Teórico

1. Máquinas de Estado

Las máquinas de estado mencionadas anteriormente son la base del proyecto para la detección de patrones complejos. A continuación se presenta un ejemplo básico que permite el entendimiento de este mecanismo.

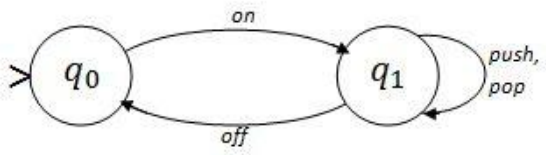

Figura 1. Autómata de una pila

En la figura 1 se puede observar un autómata que se encarga del manejo de los eventos de una pila. Inicialmente la pila se encuentra apagada (*qo*). Para que se puedan ingresar y retirar datos sobre esta estructura, primero debe ocurrir un evento de encendido (*on*). Luego de encender la pila pueden ocurrir varias inserciones (*push*) o extracciones (*pop*) de los elementos de la estructura, en este momento el autómata se encuentra en el estado (*q1*). Una vez se termina de modificar esta estructura, ocurre el evento de apagado (*off*) y la máquina regresa a su estado original.

Se ha decidido extender la librería KETAL ya que esta librería permite generar, manejar y controlar autómatas de sistemas distribuidos. En este contexto los autómatas serán utilizados para el reconocimiento de eventos distribuidos; eventos que son generados o llamados por máquinas dentro de un sistema distribuido.

2. Lenguajes y Modelos de Programación de Eventos

2.1. Caracterización de un Modelo de Eventos

2.1.1. Eventos, Patrones y Reacciones

A continuación se explica con más detalle qué caracteriza a un modelo de eventos. Un sistema basado en eventos es un conjunto de componentes de software que interactúan entre sí a través de la notificación de eventos, por lo general un cambio en el estado de un componente. Ejemplo de esto son clicks del ratón, las pulsaciones de un teclado, temporizadores, las interrupciones del sistema operativo, lecturas de sensores, etc.; este cambio de estado se conoce como un evento, a veces referenciado como un evento simple para no generar ambigüedades de los eventos *complejos* o *compuestos*. Los eventos simples tienen nombres (tipos de eventos) y atributos de datos. Un ejemplo de un evento simple es la cotización de una acción de la organización google en el mercado: *CotizarAccion* (organización: “Google”, precio: 58.22), donde *CotizarAccion* es el tipo de evento y donde organización y precio son los atributos. Los eventos se pueden escribir como métodos, de los cuales los tipos de eventos se representan por la cabecera de los métodos y los eventos se representan por las invocaciones de estos métodos. El tipo de evento *CotizarAccion* se puede generalizar representándolo como *CotizarAccion* (String organización, float precio). Un evento *compuesto* es cualquier *patrón* de eventos simple, ósea un conjunto de eventos satisfaciendo un *predicado* especifico. Un ejemplo es el promedio de 100 lecturas consecutivas de temperatura tomadas a partir de un sensor [9].

Una *reacción* es un fragmento de programa (generalmente el cuerpo de un método), ejecutado por un componente de software tras la recepción de un evento simple o compuesto. En el caso de un evento compuesto, todos los eventos simples que forman parte de él comparten la misma *reacción* (cuerpo del método).

2.1.2. Definición Formal de los Eventos Compuestos

La Detección de Eventos Compuestos (Composite Event Detection o CED) [9] tiene como tarea identificar combinaciones de eventos que son significativas en relación con los patrones de un programa definido. Un patrón de eventos [9] describe un evento compuesto como un conjunto de eventos con un predicado que debe satisfacer. La sintaxis BNF de un patrón de evento P se presenta en el Listado 1.

$$
\begin{array}{lll}
\text{attributes} & t & ::= T\ \ a \mid t,\ T\ \ a \\
\text{join} & j & ::= e(t)[v] \mid j, e(t)[v] \\
\text{condition} & b & ::= true \mid e[\text{i}].a\ op\ v \mid e[\text{i}].a\ op\ e[\text{i}].a \\
\text{predicate} & p & ::= \forall \text{i} \in [v, v]\ p \mid p\ \&\&\ p \mid p\ \|\ p \mid (p) \mid !p \mid b \\
\text{boolean operator} & op & ::= > \mid < \mid >= \mid <= \mid == \mid != \\
\text{pattern} & \mathcal{P} & ::= j\ \textbf{if}\ p
\end{array}
$$

Listado 1. Sintaxis BNF de un patrón de eventos. Tomada de [9]

En el Listado 1 *e* representa el nombre del tipo de evento, *a* el atributo, *T* el tipo de dato del atributo y *v* se utiliza para representar un valor que puede ser String, Integer o Float. La forma general de un patrón de evento *P* con *m* tipos de eventos, donde e_i contiene r_i atributos es la siguiente:

$$\mathcal{P} \ = \ e_1(T_{1,1}\ a_{1,1}, \ldots, T_{1,r_1}\ a_{1,r_1})[k_1], \ldots, e_m(T_{m,1}\ a_{m,1}, \ldots, T_{m,r_m}\ a_{m,r_m})[k_m] \ \ \textbf{if} \ \ p$$

Nos referimos a k_i como el tamaño de la ventana del tipo de evento e_i en *P*, esto significa la cantidad de eventos de tipo e_i que forma parte del evento compuesto *P*. Un evento en la ventana puede ser referenciado en el predicado mediante el uso de los índices 1, …,k_i. Existen varios tipos de condiciones definidos a continuación:

Una condición *unaria* compara un atributo de un evento con un valor, una condición binaria compara dos atributos de eventos. Una condición *intra-event* son condiciones unarias o binarias comparando dos atributos del mismo tipo de evento. Las condiciones *inter-event* son condiciones binarias comparando atributos de dos tipos de eventos distintos. De forma más general, un predicado es *n-ario* si el mayor conjunto de tipos de eventos relacionados transitivamente por condiciones *inter-event* es de tamaño *n*. Para un predicado que consiste solamente de condiciones *intra-event*, *n* es trivialmente 1 [9].

Patrones en CED (Composite Event Detection) [9] pueden ser considerados a lo largo de tres dimensiones, según las cuales los lenguajes de programación y sistemas para CED se pueden clasificar:

- El máximo tamaño *k* de la ventana para los flujos de tipos de eventos individuales (*k-size windows*)
- El máximo número *m* de tipos de eventos correlacionados (*m-way joins*)

- El máximo número de tipos de eventos *n* involucrados en predicados (*n-ary predicates*)

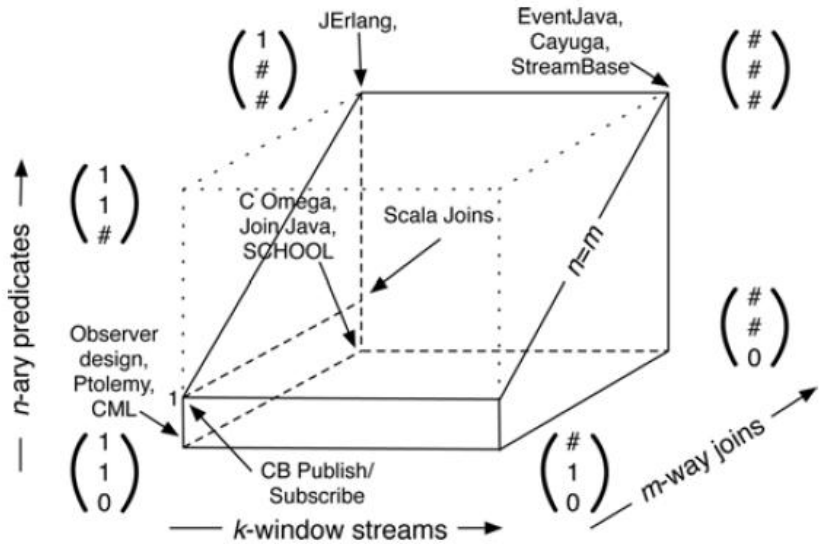

Figura 2. Descripción de las características de los lenguajes de programación y sistemas basados en eventos. El # representa la ausencia de un límite. Tomado de [9]

Debido a la clasificación anterior, se procede a describir EventJava; un modelo de eventos que cumple con las tres características de la CED que se describieron anteriormente.

2.2. Un Ejemplo Concreto de un Framework de Eventos: EventJava

2.2.1. Modelo de Eventos

Como se mencionó anteriormente, EventJava es una extensión de Java que define un modelo de eventos que se acopla al contexto de sistemas distribuidos. Este modelo de eventos se caracteriza por los siguientes atributos:

✓ Permite la definición de métodos de eventos (*event methods*); para esto presenta las palabras reservadas *event* y *when*, que definen el encabezado y predicado o patrón del evento respectivamente. A continuación se presenta la estructura básica de estos:

```
1. event methodName(parameters){
```

```
2.    when (predicates){
3.          reaction…
4.    }
5.  }
```

Los parámetros del método (*parameters*) pueden ser cualquier tipo de dato primitivo o que implemente la interfaz Serializable, esto para permitir la notificación de esta información a través de la red. Los predicados (*predicates*) son las condiciones que debe cumplir un determinado evento para su ocurrencia. La reacción (*reaction*) son los pasos que se deben ejecutar una vez se cumplan las condiciones definidas en los predicados. Esta se ejecuta de manera asíncrona, es decir, en un hilo aparte, cada vez que la ocurrencia de un evento cumpla con los predicados.

- ✓ Soporta el envío de notificaciones de mensajes por medio de *unicast* y *broadcast*. Esto es útil cuando se quiera informar a un nodo en específico o a todos sobre la ocurrencia de un evento. Suponiendo que se cuenta con una clase ClassX y una instancia de esa clase denominada ClassXInstance, el envío de *unicast* se define de la siguiente manera:

  ```
  1. ClassXInstance.methodName(parameters);
  ```

  La línea anterior notifica únicamente a la instancia ClassXInstance la ocurrencia del evento methodName. En caso de que se quisiera realizar un *broadcast* para notificar a todas las instancias (locales y/o remotas) de la clase sobre la ocurrencia de un evento, la manera de hacerlo es la siguiente:

  ```
  1. ClassX.methodName(parameters);
  ```

- ✓ Permite la definición de eventos complejos, es decir, conjunciones o disyunciones entre varios eventos simples, además de que permite relacionar eventos entre sí, es decir, comparar sus atributos explícitos (información específica de la aplicación) e implícitos (tiempo, coordenadas, etc.) en la definición del patrón que se desee reconocer.

  Cuando se quiera validar la ocurrencia de dos o más eventos, la declaración del encabezado de estos se escribe uno detrás del otro separados por coma, de la siguiente manera:

  ```
  1. event methodA(int x1, int y2), methodB(int x2, int y2, int
     z){
  2.    when (methodA < methodB && x1 == x2){
  3.          reaction…
  4.    }
  5. }
  ```

Como se puede observar en el ejemplo anterior se declaran dos métodos. En el predicado se compara el tiempo de ocurrencia de dichos métodos (argumento implícito) y además se comparan dos argumentos, cada uno perteneciente a un evento diferente; esto es lo que se conoce como patrón de correlación, lo que permite comparar información relativa a diferentes fuentes o eventos para determinar su grado de relación.

- ✓ Por medio de relojes lógicos se puede determinar la causalidad entre dos o más eventos, como se muestra en el ejemplo anterior
- ✓ Se pueden definir varios eventos como un arreglo de objetos, teniendo en cuenta las siguientes dos restricciones. Este mecanismo se conoce como Flujos:
  - Todo evento en la posición [i] es menor que el evento en la posición [i+1]
  - La anterior regla no significa que todos los eventos sean necesariamente consecutivos

  La declaración de un flujo se muestra a continuación:

```
1. event methodName[2](int x){
2.    when (methodName[1].x == methodName[2].x){
3.          reaction…
4.    }
5. }
```

  Como se puede observar, en el ejemplo anterior el flujo es un arreglo de dos eventos methodName. En el predicado se pueden obtener los atributos relativos a cada evento indicando su posición, como se realiza usualmente en java, para definir la condición de ejecución del evento seleccionado.

La lista anterior muestra las capacidades actuales de EventJava para el manejo de eventos en sistemas distribuidos. A continuación se listan algunos de los problemas que presenta EventJava, los cuales apuntan al problema principal descrito anteriormente:

- ✓ No es posible la sincronización en el modelo de eventos. La sincronización en este contexto hace referencia a la capacidad de un nodo de esperar por la respuesta de uno o varios nodos para realizar una acción específica. Esto no es posible porque la definición de los eventos no contempla la posibilidad de retornar un valor.

- ✓ No se manejan tipos de retorno en *broadcast* de eventos. No se recibe una respuesta (inmediata o prolongada) luego de la ocurrencia de un evento.
- ✓ El manejo de excepciones en modelos de eventos distribuidos no es controlado. Este problema hace referencia, al igual que los anteriores, a la carencia de valores de retorno de los eventos. En caso de que ocurra una excepción de procesamiento en alguno de los nodos, no hay posibilidad de informar a los demás este hecho.

3. Depuración de Errores de Concurrencia en Middleware Distribuido

3.1. Deadlock en Middleware Distribuido

De acuerdo a [11] y a [12] , un *deadlock* es una condición en un sistema donde un proceso no puede proceder por que necesita obtener un recurso retenido por otro proceso, pero el mismo esta teniendo un recurso que el otro proceso necesita. Por lo tanto ambos procesos quedan bloqueados.

Las mismas condiciones anteriores para *deadlocks* aplican en sistemas distribuidos; desafortunadamente, estos son más difíciles de detectar, evitar y prevenir. En el siguiente ejemplo se utilizan dos caches distribuidos, las acciones en el primer cache son replicadas en el segundo cache por medio de un framework de replicación. A continuación se presenta gráficamente esta situación:

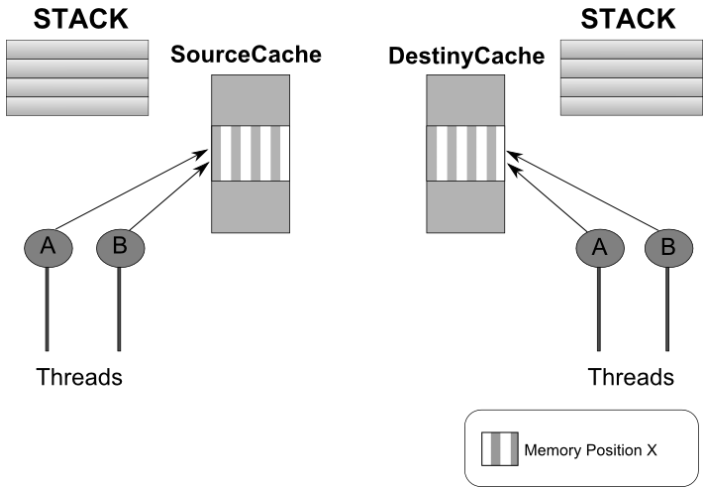

Figura 3. Estado inicial de los cachés

En la figura 3 se muestra el escenario de nuestro ejemplo para representar un caso de *deadlock* en un sistema distribuido, en donde hay dos nodos cada uno con un caché. Estos están compartiendo una misma posición de memoria *X*, y hay dos hilos *A* y *B* en cada nodo ejecutándose paralelamente que interactuarán con esta memoria compartida.

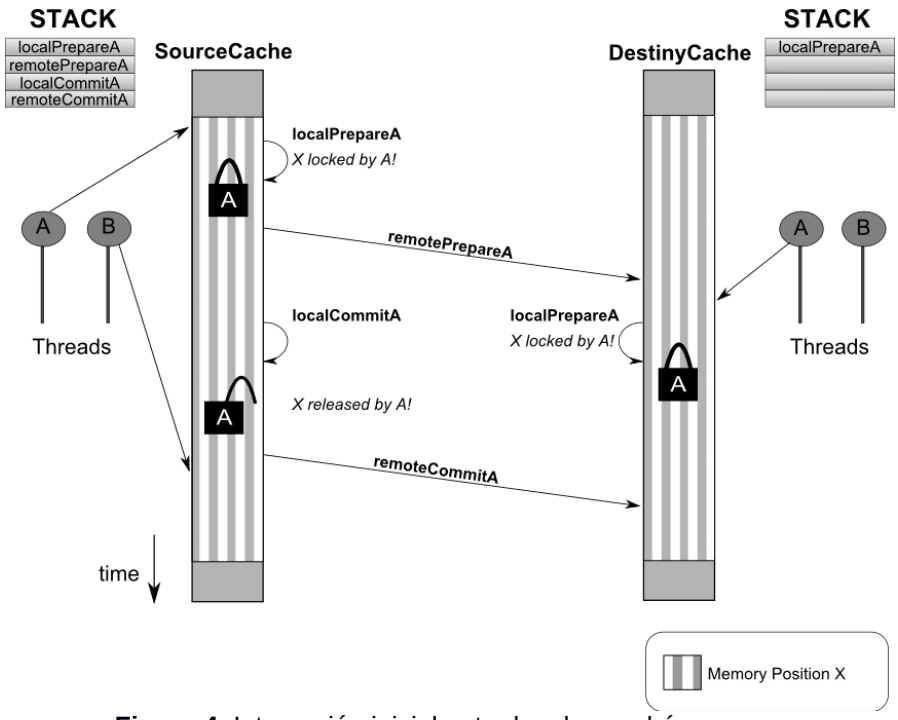

Figura 4. Interacción inicial entre los dos cachés

Los dos hilos que se están ejecutando en el nodo SourceCache quieren editar una variable que se encuentra en la posición de memoria X, para esto se utiliza el protocolo *two phase commit*; que es un algoritmo distribuido que coordina todos los procesos que participan en una transacción atómica distribuida. Como se observa en la figura 4, primero el hilo *A* del nodo fuente realiza de forma completa un *prepare phase* del protocolo, luego se pasa a la fase *commit* donde se libera el candado del caché fuente inmediatamente después de ejecutar el *commit* local.

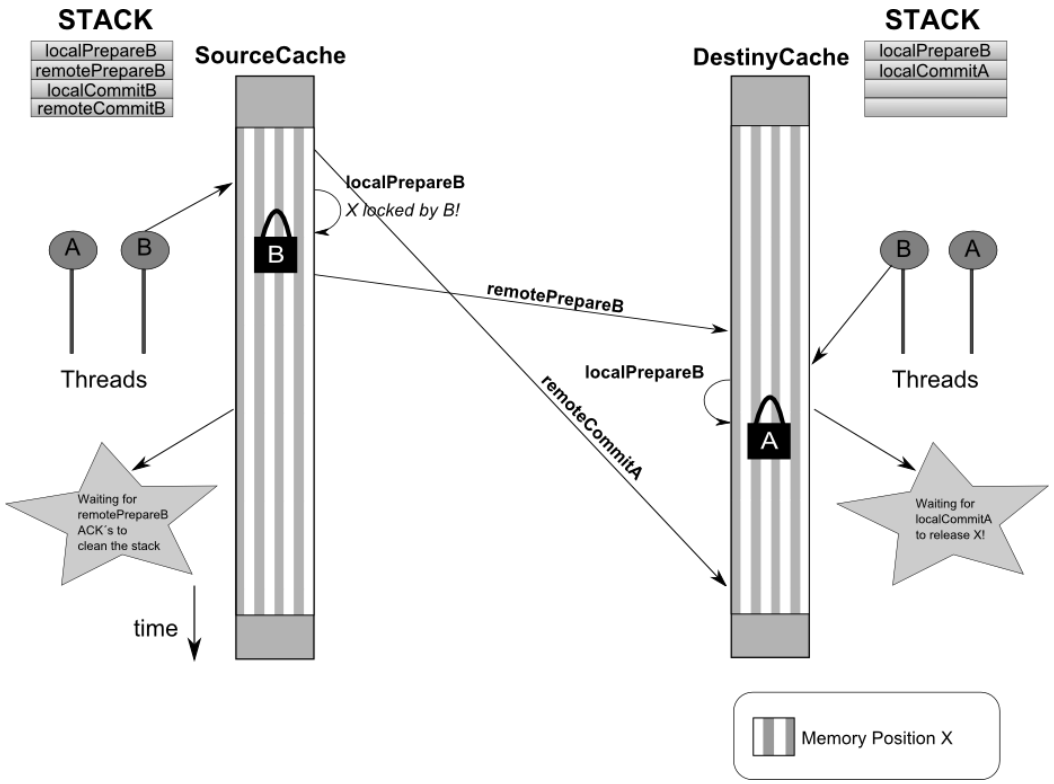

Figura 5. Situación de *deadlock*

Antes de terminar la fase final del *commit* con el cache remoto (*DestinyCache*), el hilo *B* interrumpe sus operaciones de transacción, en particular adquiriendo el candado en la misma posición compartida en el cache debido a que este hilo ejecuta un *local prepare*. Luego, el hilo *B* prosigue para terminar su fase de *prepare* pero la posición de memoria compartida en el host destino esta bloqueada por el candado del hilo *A,* por lo tanto este *local prepare* se guarda en la pila quedando a la espera de que se libere el candado. Suponiendo que por problemas en la red como el *delay*, el *remote commit* del hilo *A* (que fue enviado antes de que la fase *prepare* del hilo *B* comenzara) se demoró en llegar a su destino, este *remote commit* se guarda en la pila esperando a que el tope de la pila sea consumido (*local prepare* del hilo *B*). Aquí se genera un situación de *deadlock,* debido a que el caché fuente está esperando por un ACK del *remote prepare* del hilo *B* y el caché destino está esperando por un *local commit* del hilo *A* para liberar el candado de la memoria compartida, entonces ambos necesitan un recurso del otro bloqueándose mutuamente.

3.2. Usando Ordenamiento Causal para la Corrección del Deadlock

Por lo general, los modelos existentes para la detección de patrones de eventos se basan en la suposición implícita de que los eventos son consumidos en el mismo orden en que son producidos. Sin embargo, las notificaciones de los eventos son realizados a través de mensajes enviados en la red, y la red no es un recurso confiable por los problemas que se pueden generar, como se mencionó en secciones anteriores.

Generalmente, el autómata finito nos permite modelar los patrones de eventos que llegan pero se enfrenta a dos problemas [13]:

- No podría coincidir una secuencia valida de eventos que llegaron en un orden incorrecto al host donde la secuencia va a hacer consumida.

- Podría coincidir una secuencia incorrecta que se derivan de eventos que ocurren en diferentes hosts en el orden equivocado, pero cuyo orden ha sido invertido, por ejemplo, por el retraso en la red de los mensajes que van hacia el host donde la secuencia va a hacer consumida.

Varios algoritmos y soluciones se han propuesto para resolver este problema; una primera aproximación es sincronizar los relojes de cada nodo y forzar el ordenamiento de los mensajes. Esta aproximación ha mostrado ser muy costosa y compleja de implementar y mantener. Otras soluciones se han presentado que dependen de órdenes parciales como los relojes lógicos [14] y los relojes vectoriales [15]. La solución más adecuada se utiliza en la librería KETAL, la cual se basa en los relojes vectoriales de Mattern [15] para soportar el ordenamiento causal de los eventos.

En el ejemplo anteriormente expuesto, el problema radica en que por inconvenientes en la red se genera un *delay* o retraso en la llegada del *remote commit* de *A*, causando que toda la fase *prepare* de *B* sea completada antes y el hilo *B* adquiera el candado. Nuestro modelo para capturar los patrones de eventos utilizando un autómata nos permite detectar la ocurrencia de este error; en capítulos siguientes se presenta la solución utilizando nuestro lenguaje de eventos para detectarlo. Sin embargo, este caso en particular de *deadlock* se puede corregir utilizando el ordenamiento causal, para esto se creó un constructor *seqCausalOrder,* que asegura que todas las relaciones causales son consumidas por el autómata si se ordenan todos los eventos entrantes. Su semántica asegura que cada evento es retrasado para esperar por el evento que lo precede causalmente. Con esto, el *remote prepare* de *B* cuando llegue al nodo destino, debe esperar a que el evento que lo precede casualmente llegue, ósea en este caso el *remote commit* de *A*; cuando este último evento llegue, con ayuda de los

relojes vectoriales se podrán ordenar estos eventos correctamente según el orden de salida desde el nodo fuente, asegurando así que la fase *commit* del hilo *A* termine y libere el candado para que hilo *B* lo pueda adquirir, evitando así la generación del *deadlock*.

4. Soluciones no Distribuidas para la Detección de Data Races

Este tipo de problemas parte de la utilización de varios hilos o procesos trabajando de manera sincronizada para acceder y manipular un recurso compartido. Los estudios realizados en [16] determinan que existen cuatro diferentes técnicas para la detección de *data races*: *static*, *dynamic*, *"on-the-fly"* y *postmortem.*

Los detectores que utilizan la técnica *static* analizan el código fuente de un programa por medio de anotaciones. Los detectores *dynamic* analizan secuencias de pasos específicos o rastros de un programa en ejecución. La técnica *on-the-fly* realiza el procesamiento de los eventos del programa en paralelo con la ejecución de estos. Por último, la técnica *postmortem* en llevar el registro de estos eventos en un archivo temporal para luego analizar este después de que el programa que se va a analizar se ejecute.

El detector en particular propuesto en [16] llamado ThreadSanitizer, es un detector dinámico que utiliza relaciones de causalidad (denominadas *happens-before*) y análisis de utilización de *locks* o candados de cada proceso para conformar lo que se denomina *the hybrid state machine.* Para la implementación de este detector es importante diferenciar dos conceptos fundamentales, los cuales son utilizados para definir la relación de causalidad. Estos se presentan a continuación:

Dados dos hilos *Tx, Ty* y dos recursos compartidos *Ax, Ay*, pueden ocurrir las siguientes relaciones entre estos:

Happens-Before Arc: Un par de eventos *X=Signal (Tx, Ax)* y *Y=Wait (Ty, Ay)* tal que *Ax=Ay*, *Tx!=Ty* y *X* ocurre primero.

Happens-Before: Un orden parcial en el set de eventos, donde dados dos eventos *X=TypeX (Tx, Ax)* y *Y=TypeY (Ty, Ay),* el evento *X* precede al evento *Y* (representado como *X<Y*) si *X* a ocurrido antes que *Y* y al menos una de las siguientes condiciones se cumple:

- *Tx=Ty*
- *{X, Y}* es una relación *happens-before arc*

- Existen dos eventos *E1, E2* tal que *X<=E1<E2<=Y.* Esto determina que la relación *bappens-before* es transitiva

Teniendo claros los conceptos anteriores, se puede determinar el orden de ocurrencia de un set de eventos y las relaciones que existen entre ellos. La siguiente figura presenta un ejemplo que permite ilustrar los conceptos anteriores:

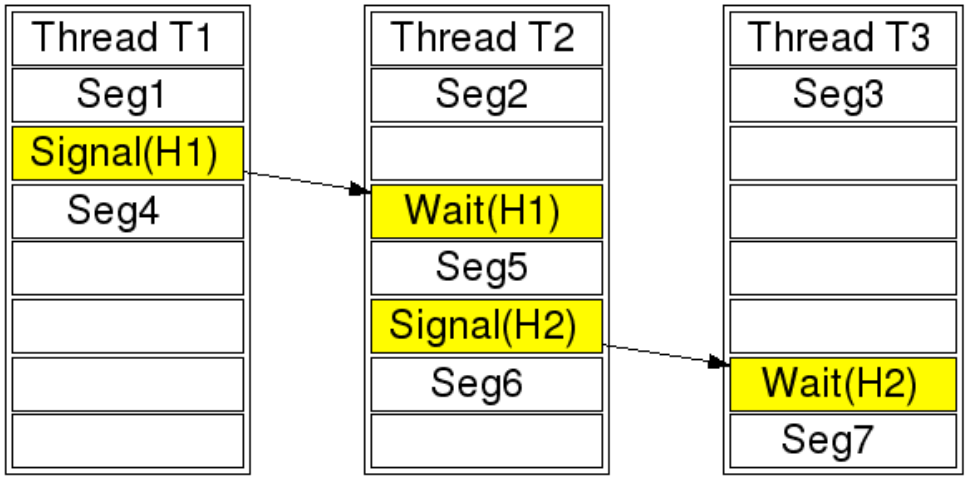

Figura 6. Relación *happens-before* entre diferentes hilos. Tomado de http://code.google.com/p/data-race-test/wiki/ThreadSanitizerAlgorithm

En la figura anterior se observan tres diferentes hilos divididos en segmentos y se pueden obtener las siguientes conclusiones:

1. *Seg1 < Seg4* porque estos segmentos se encuentran en el mismo hilo y *Seg1* ocurre primero que *Seg4*
2. *Seg1 < Seg5* debido a que hay una relación de *happens-before arc* entre los eventos *Signal (T1, H1)* y *Wait (T2, H2)*
3. *Seg1 < Seg7* ya que *Seg1 < Seg5* y además *Seg5 < Seg7* (Relación transitiva)
4. *!(Seg4 < Seg2)* porque no hay una relación de tipo *happens-before* entre estos

Como se mencionó en párrafos anteriores, el detector denominado *hybrid machine* utiliza tanto relaciones *happens-before* como análisis de utilización de candados para su implementación. La necesidad de implementar este detector híbrido parte

del hecho de que detectores que solo evalúan relaciones *happens-before* pueden presentar inconvenientes al momento de utilizar como método de sincronización la utilización de candados. Estos inconvenientes son ajenos al programador ya que son originados por factores como el planificador de hilos del sistema operativo. Un ejemplo de esta situación se presenta a continuación:

El siguiente código muestra el acceso de dos hilos *Thread1, Thread2* a una variable compartida *X* utilizando como método de sincronización el uso de candados.

```
1. //Código tomado de http://code.google.com/p/data-race-
   test/wiki/ThreadSanitizerAlgorithm
2. void Thread1() {
3.   X = 1;
4.   mu.Lock (); // E1
5.   mu.Unlock(); // E2
6. }
7. void Thread2() {
8.   mu.Lock(); // E3
9.   mu.Unlock(); // E4
10.   X = 2;
11.}
```

Las líneas 4, 5, 8 y 9 hacen referencia a los eventos de sincronización que ocurren en cada hilo. Estos se denominan *E1, E2, E3* y *E4* respectivamente. El orden de ejecución de los eventos, es decir, de qué hilo inicia primero su ejecución depende completamente del planificador de hilos; luego si el hilo *Thread1* se ejecuta primero, ocurre una relación *happens-before arc* entre *E2* y *E3*. Esta situación se muestra en la siguiente gráfica:

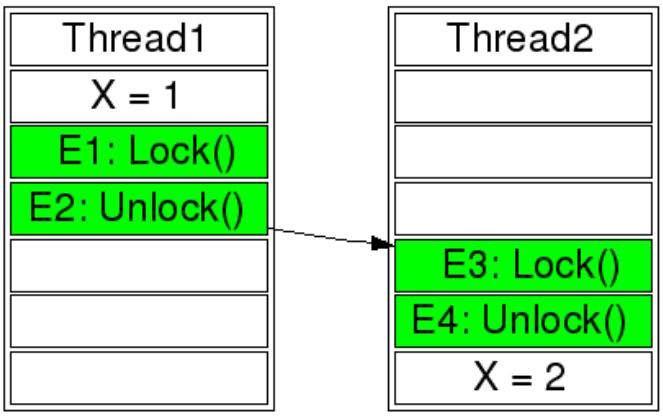

Figura 7. Relación *happens-before arc* entre *E2* y *E3.* Tomado de http://code.google.com/p/data-race-test/wiki/ThreadSanitizerAlgorithm

Debido a que *E2* ocurre antes que *E3*, se crea una relación *happens-before arc* entre los segmentos *Seg1 (X=1)* y *Seg2 (X=2)*. Los eventos *E1, E2, E3* y *E4* no hacen parte de ningún segmento debido a que estos son eventos de bloqueo y los segmentos se conforman únicamente de eventos de acceso a memoria (*read* y *write*). La relación *happens-before arc* se crea porque el evento *E1* envía una notificación al hilo *Thread2,* el cual esta esperando a que el hilo *Thread1* sea atendido. Si ocurre el caso contrario y se ejecuta el hilo *Thread2* primero, se presenta la siguiente situación:

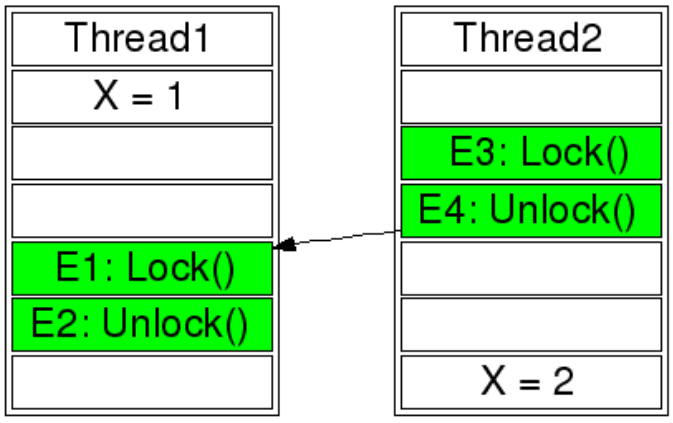

Figura 8. Relación *happens-before arc* entre *E4* y *E1.* Tomado de http://code.google.com/p/data-race-test/wiki/ThreadSanitizerAlgorithm

En este caso, un *data race* ocurre porque justo después de que el hilo *Thread2* desbloquea el candado sobre *X* (evento *E4*), este mismo hilo asigna el valor 2 a *X*, pero en el momento en el que el hilo *Thread1* es notificado, este asigna el valor 1 a *X.* Esto se traduce en una condición de *data race* ya que se tienen dos hilos accediendo a la misma posición de memoria y al menos uno de ellos esta realizando modificaciones sobre dicha posición (*X*).

Como se puede observar, los ejemplos anteriores ilustran el hecho de que la capacidad de un detector que sólo analiza relaciones de tipo *happens-before* depende completamente del planificador de hilos; lo que indica que este tipo de detectores es menos previsible. Aunque los detectores de candados e híbridos son más confiables y más rápidos que los detectores *happens-before*, estos ocasionan

mayores falsos positivos, luego la utilización de un tipo de detector o de otro depende de diferentes factores como la aplicación a analizar, el método de sincronización a utilizar, entre otros.

5. Modelando Concurrencia Utilizando Autómatas

Existen diferentes propuestas y estudios realizados sobre el tema de modelamiento de concurrencia por medio de autómatas. Libros como [12] presentan de manera clara y concisa mecanismos para realizar este tipo de tareas, teniendo en cuenta los diferentes problemas y restricciones que se deben considerar para trabajar con este tipo de aproximaciones. A continuación, se presentan los problemas más relevantes que deben ser considerados en este ámbito y los mecanismos o métodos que se utilizan para dar solución a estos:

El primer inconveniente hace referencia a cómo modelar la velocidad a la que un proceso se ejecuta de respecto a otro. Esta velocidad depende de diferentes factores como el número de procesadores y la estrategia de planificación implementada por el sistema operativo para la asignación de recursos a los hilos que se deben ejecutar. Debido a que el propósito de la utilización de autómatas para modelar concurrencia es poder mostrar la interacción de los procesos, independientemente del número de procesadores y estrategias de planificación, se asume que cada proceso se ejecuta a velocidades arbitrarias respecto a los demás. Esto presenta la ventaja de poder modelar la interacción de aplicaciones concurrentes sin considerar la(s) máquina(s) sobre la(s) que se ejecuta.

El siguiente problema apunta a cómo modelar concurrencia o paralelismo. Para poder realizar esto se asume que una acción *a* es concurrente con una acción *b*, si el modelo permite que estas acciones puedan ocurrir en cualquier orden (*a*-*b, b*-*a*). El hecho de que no se tiene en cuenta el tiempo de ejecución de las acciones o eventos, permite evitar consideraciones de si la ocurrencia de *a* se da exactamente al mismo tiempo que *b.*

La última consideración tiene foco en cómo modelar el orden de ejecución de los eventos ocurridos. Aunque se sabe que los eventos de un proceso se ejecutan en una secuencia específica, el hecho de modelar varios procesos ejecutándose paralelamente significa diferentes interrelaciones en la ocurrencia de eventos relativos a diferentes procesos. La manera en la que se trata este inconveniente es modelar todas las posibles permutaciones de la ocurrencia de los eventos de múltiples procesos, lo que permite una abstracción completa de como los procesadores cambian en la ejecución de los procesos debido a interrupciones externas. Este modelo es conocido como *asíncrono*.

Diferentes métodos han sido utilizados para la representación y el modelado de procesos en paralelo. Una de las estrategias utilizadas para este fin es la que se presenta en [12], la cual define la concurrencia como sigue:

“Si P y Q son procesos entonces (P||Q) representa la ejecución concurrente de P y Q. El operador || es el operador de composición paralela”

Dentro de las propiedades el operador || se encuentra la propiedad conmutativa y la asociativa, ya que como este relaciona procesos concurrentes, el orden de ejecución de estos es irrelevante.

Un ejemplo de aplicación de este operador se puede observar en el siguiente ejemplo presentado en [12]:

El proceso *ITCH* se compone del paso *scratch* y termina su ejecución. Este se representa de la siguiente manera:

ITCH = (scratch->STOP)

El proceso *CONVERSE* se representa como sigue:

CONVERSE = (think->talk->STOP)

Por último, la composición de estos procesos se representa de la siguiente manera:

||*CONVERSE_ITCH = (ITCH* || *CONVERSE)*

Como se mencionó anteriormente, para modelar la concurrencia con autómatas es necesario plantear todas las permutaciones de los procesos involucrados. Para este caso particular, las interrelaciones de los eventos son las siguientes:

think->talk->scratch

think->scratch->talk

scratch->think->talk

Los autómatas para cada uno de los procesos descritos anteriormente, incluyendo la composición de ambos procesos se presentan a continuación:

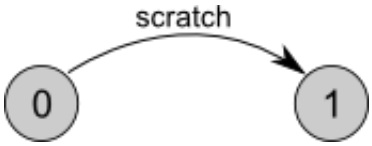

Figura 9. Autómata del proceso *CONVERSE.* Tomada de [12]

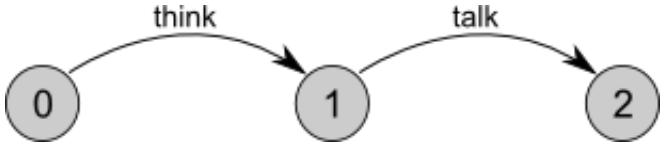

Figura 10. Autómata del proceso *ITCH*. Tomada de [12]

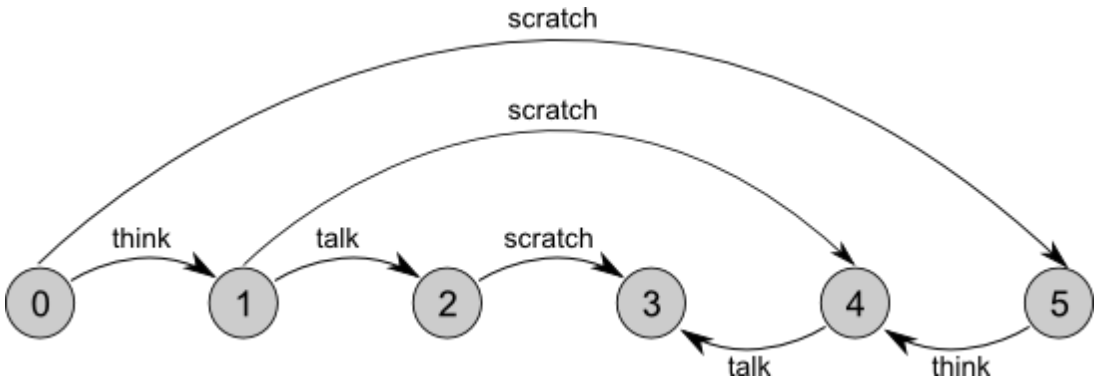

Figura 11. Autómata del proceso ||*CONVERSE_ITCH.* Tomada de [12]

El ejemplo anterior representa la ejecución concurrente de dos procesos independientes. Este no contempla el caso en que los procesos que se ejecutan comparten el acceso a un recurso por medio de un evento en común. Para modelar este tipo de situaciones basta con reducir el número de permutaciones a las permitidas, dependiendo de las restricciones de la aplicación que se desea modelar. El siguiente ejemplo, tomado de [12] ilustra esta situación:

Existen dos procesos *BILL* y *BEN*, los cuales se componen de los siguientes eventos:

BILL = (play->meet->STOP)

BEN = (work->meet->STOP)

||BILL_BEN = (BILL || BEN)

Como se puede observar, ambos procesos comparten el evento *meet*. Teniendo en cuenta esta restricción, las posibles interrelaciones de estos procesos son las siguientes:

play->work->meet

work->play->meet

Como se puede observar, la acción compartida *meet* se encarga de sincronizar la ejecución de estos procesos. El autómata que modela esta situación es el siguiente:

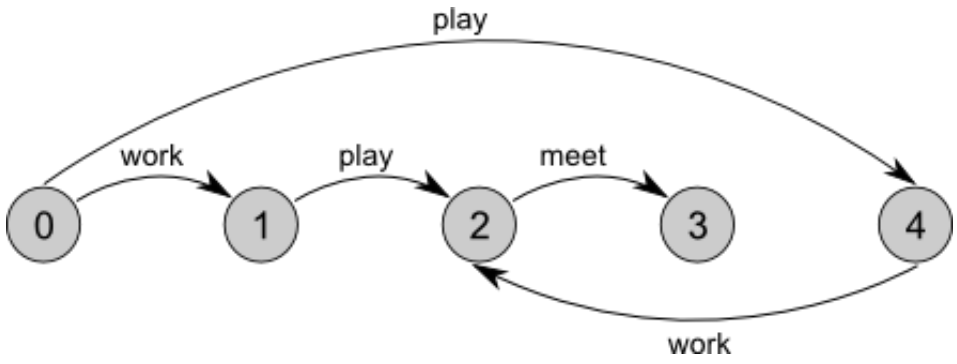

Figura 12. Autómata del proceso ||*BILL_BEN.* Tomada de [12]

Otra de las consideraciones importantes a la hora de modelar concurrencia con autómatas es la coincidencia de nombramiento de los eventos que hacen parte de diferentes procesos, es decir, la situación en la que un proceso contiene uno o más eventos *Xset = {X1, X2,..., Xn}* y existe otro proceso que contiene al menos un evento llamado exactamente igual como uno de los eventos pertenecientes a *Xset*.

Para solucionar el problema anterior, se propone en [12] el etiquetamiento de los procesos por medio de la utilización de prefijos. Para ilustrar esta situación, se contempla el siguiente ejemplo tomado de [12]:

Existen dos procesos llamados *SWITCH* que se componen de las siguientes acciones:

SWITCH = (on->off->SWITCH)

Para poder diferenciar la ocurrencia de un evento e identificar a qué proceso pertenece este, se utilizan los siguientes prefijos:

a:SWITCH = (a.on->a.off->a:SWITCH)

b:SWITCH = (b.on->b.off->b:SWITCH)

||TWO_SWITCH = (a:SWITCH || b:SWITCH)

Los autómatas que modelan estos procesos, incluyendo la máquina de estados que contiene las interrelaciones entre estos dos procesos se muestran a continuación:

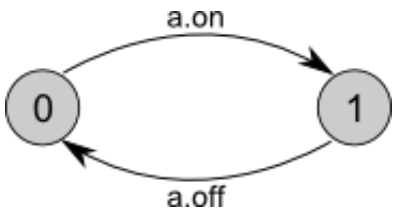

Figura 13. Autómata del proceso *a:SWITCH*. Tomada de [12]

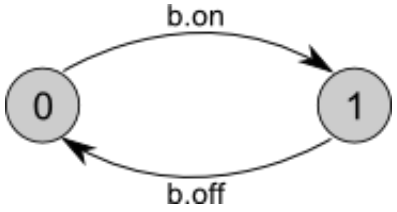

Figura 14. Autómata del proceso *b:SWITCH*. Tomada de [12]

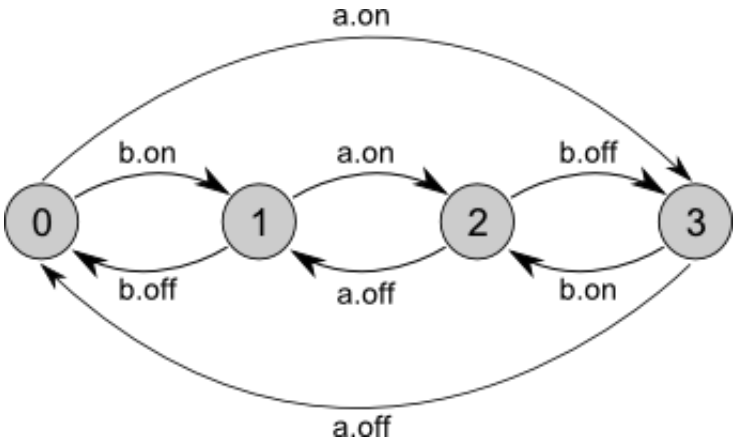

Figura 15. Autómata del proceso ||*TWO_SWITCH.* Tomada de [12]

Los ejemplos anteriores permiten ilustrar cómo el enfoque dado en [12] permite el modelamiento de concurrencia por medio de autómatas. Nuestra propuesta coincide con los planteamientos anteriores en cuanto al modelamiento de los eventos sin contemplar el tiempo de ejecución de estos y la manera arbitraria en que pueden ocurrir estos. Sin embargo, debido a que nuestro objetivo de estudio plantea la utilización de autómatas deterministas para secuencias específicas de la ocurrencia de eventos, los autómatas que nosotros plateamos no necesitan contemplar todas las posibles interrelaciones entre los eventos de los procesos relacionados de la aplicación, sino aquellas en las que su ocurrencia genera fallos o errores sobre la ejecución del programa. Adicional a esto, el inconveniente de nombramiento de los eventos es solucionado utilizando diferentes etiquetas sobre las transiciones o eventos; esto con el fin de dar un identificador único a cada evento que permita describir la acción que se está generando.

Los ejemplos anteriores son una demostración sencilla del uso de autómatas para el modelado de concurrencia; a continuación se utiliza este mecanismo para abordar los problemas tratados en este documento de *liveness* y *deadlock*. Antes de proceder a la utilización de los autómatas es necesario resaltar cuatro condiciones suficientes y necesarias identificadas por Coffman, Elphick y Shoshani (1971) para la ocurrencia de un *deadlock*:

- ✓ Los procesos involucrados comparten recursos que son utilizados por estos bajo exclusión mutua (concepto explicado en [1] y en [12])
- ✓ Los procesos retienen los recursos conseguidos mientras esperan por la adquisición de recursos adicionales
- ✓ Una vez un recurso es adquirido por un proceso, este es liberado únicamente por voluntad propia del proceso que lo posee
- ✓ Existe un ciclo entre procesos tal que cada proceso retiene un recurso que es necesario para la ejecución de su sucesor

Una vez conocidas estas condiciones, exponemos un ejemplo sencillo de deadlock, tomado de [12], el cual consiste en un sistema conformado por dos procesos P y Q. Cada uno de estos procesos realiza la misma tarea: escaneo de un documento y la impresión de este, utilizando un scanner e impresora compartidos. Cada proceso adquiere tanto la impresora como el scanner, realiza el escaneo y la impresión y por último, libera los recursos. La única diferencia entre estos dos procesos es el orden de adquisición de los recursos, dado que P adquiere la impresora primero y Q adquiere el scanner primero. Como se puede observar, las cuatro condiciones anteriores se cumplen totalmente: cada proceso utiliza los recursos de manera independiente, cada proceso retiene o la impresora o el scanner mientras espera por la adquisición del segundo recurso, los recursos

son liberados exclusivamente por cada proceso y ambos procesos necesitan de los recursos retenidos. El autómata que describe el proceso P es el siguiente:

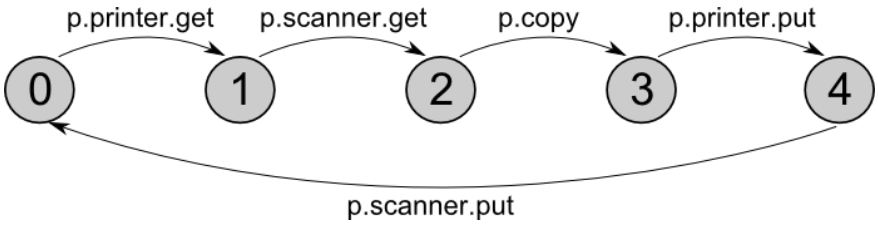

Figura 16. Autómata del proceso *P.* Tomado de [12]

El autómata del proceso Q sería igual a la figura anterior pero invirtiendo las transiciones entre los estados 0, 1 y 2. La ocurrencia de un *deadlock* en este sistema ocurre por la siguiente razón: ambos procesos comienzan su ejecución al mismo tiempo y el proceso P solicita y obtiene la impresora, mientras el proceso Q obtiene el scanner. El siguiente paso de ambos procesos es obtener el recurso que esta siendo retenido por el proceso paralelo, por lo que ocurre un bloqueo y no se ejecutan más transiciones. Un autómata sencillo que detecte esta condición se presenta a continuación:

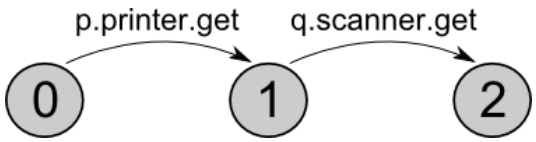

Figura 17. *Deadlock* del sistema *P* y *Q*

En la figura anterior, cuando se alcanza el estado 3 los procesos *P* y *Q* van a detenerse hasta que uno de ellos libere el recurso que tiene retenido, pero como esta situación nunca ocurre, se presenta el *deadlock*. Este autómata puede invertir sus transiciones y seguirá detectando el problema. Esta situación es un ejemplo trivial del problema de *deadlock*, a continuación trataremos un ejemplo un poco más complejo referente a procesos lectores y escritores accediendo a una base de datos compartida.

El problema de los lectores y escritores, tomado de [12], presenta las siguientes características: Existen dos lectores y dos escritores. Los lectores ejecutan transiciones que examinan la información almacenada en la base de datos,

mientras que los escritores examinan y además modifican la base de datos. Para que la base de datos sea actualizada correctamente, los escritores deben tener acceso exclusivo a esta mientras realizan la modificación. Si no hay escritores accediendo a la base de datos, cualquier número de lectores puede acceder de manera concurrente a esta. Aunque en este documento los autómatas son utilizados para la detección de errores, estos pueden servir de herramientas para el modelado correcto de una aplicación, es decir, que no existen transiciones erróneas que detengan la ejecución del programa; esto es conocido como *safety.* Los estados que son alcanzados y no tienen transiciones salientes son conocidos como estados de *deadlock.* El siguiente autómata presenta el problema descrito con todos los estados necesarios para que cumpla con la propiedad *safety* y además, se incluye un estado de *deadlock* o sumidero para resaltar las transiciones erróneas.

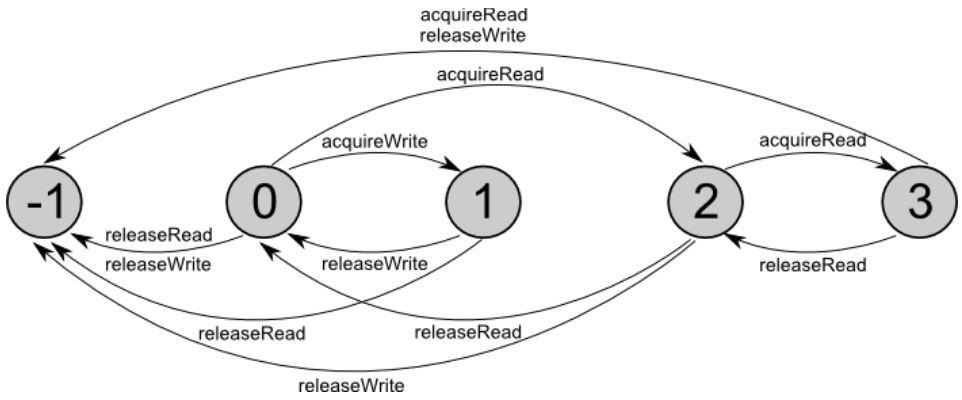

Figura 18. Autómata con propiedad *safety.* Tomada de [12]

Como se puede observar, en la figura anterior se declara un estado -1, al cual desembocan todas las transiciones erróneas. Los demás estados representan todas las transiciones válidas en el orden correcto en el que deberían ejecutarse. Entre los estados 0 y 1 se ejecutan los eventos de los procesos escritores, y se ve claramente que el candado de escritura es liberado luego de ser adquirido. Los estados 2 y 3 permiten la adquisición de hasta 2 candados de lectura por los procesos lectores, cumpliendo el límite establecido. Es importante resaltar que si algún candado de lectura ha sido adquirido, los procesos escritores no se pueden ejecutar.

Esta máquina de estados introduce un concepto adicional denominado *progress,* el cual contempla que si se define una serie de eventos *P = {a1, a2,…, an}*, los cuales pueden ser ejecutados de manera indefinida en cualquier estado de un sistema, al menos uno de estos eventos también será ejecutado de manera

indefinida. Este concepto es lo opuesto a la situación de *starvation*, la cual evita que alguna acción esperada se ejecute. Existen dos posibles problemas con la máquina de la figura 18: puede ocurrir una condición de *data races* entre los procesos lectores y escritores y, dependiendo del planificador, este autómata puede incurrir en una situación de *starvation;* a continuación se explican y modelan estos.

El primer problema está directamente relacionado con el mecanismo de sincronización utilizado para el acceso a la base de datos. Debido a que los lectores y escritores son procesos concurrentes, si el programador no implementa una correcta sincronización de los procesos, se puede presentar la siguiente situación:

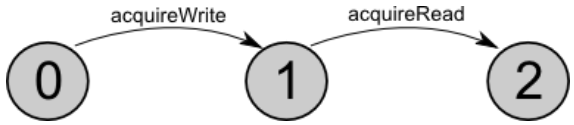

Figura 19. Detección posible *data race* del problema de lectores y escritores

Si algún proceso escritor adquiere el candado de escritura, e inmediatamente un proceso lector adquiere el candado de lectura sin comprobar que la base de datos esté siendo accedida por un escritor, se tiene que el lector va a estar modificando la información de los datos y el lector puede estar consultando la información que está siendo actualizada, lo que puede generar inconsistencias en los datos de cada proceso. Este es un típico caso de *data race,* el cual es detectado con el autómata anterior. El orden de las transiciones de este autómata puede ser invertido y se seguirá corriendo el mismo riesgo de incurrir en una condición de *data race.* El segundo problema contempla la situación de *starvation.* Consideremos el siguiente autómata, el cual modela un momento específico de ejecución de la máquina de estados de la figura 18.

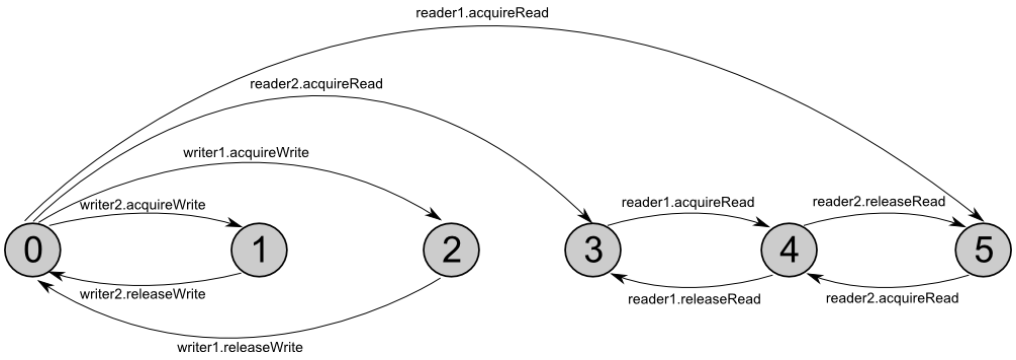

Figura 20. *Starvation* del problema de lectores y escritores. Tomada de [12]

Los estados 0, 1 y 2 de la figura anterior referencian a los estados 0 y 1 de la figura 18. Surge un estado adicional porque se tratan transiciones individuales para cada proceso escritor. Esta misma situación ocurre con los estados 3, 4 y 5 de la figura anterior, los cuales referencian a los estados 2 y 3 de la figura 18 y modelan los dos procesos lectores. En el autómata anterior ocurre la siguiente situación: el planificador de tareas está asignado una prioridad baja a los eventos de *release*, lo que ocasiona que cuando ocurra alguno de los eventos de adquisición del candado de alguno de los lectores, se elimine el evento de *release* de este candado que retorna al estado 0 (figura 18), ya que si alguno de estos eventos ocurre, el autómata se posiciona en el estado 3 o 5, y como dichos estados presentan transiciones de salida que referencian a eventos de adquisición, estas son las que permanecen activas. Considerando este autómata, si ocurre un evento de un proceso lector, el planificador va a seguir dando prioridad a procesos lectores, lo que bloquea a los procesos escritores para que puedan acceder a la base de datos, generando esto una condición de *starvation.* Una posible alternativa de detección de este problema es adicionar un estado 6 al que pueden llegar los estados 3, 4 y 5 cuando ocurra un evento de *timeout* de alguno de los procesos escritores que se ha iniciado, como se muestra en la siguiente figura:

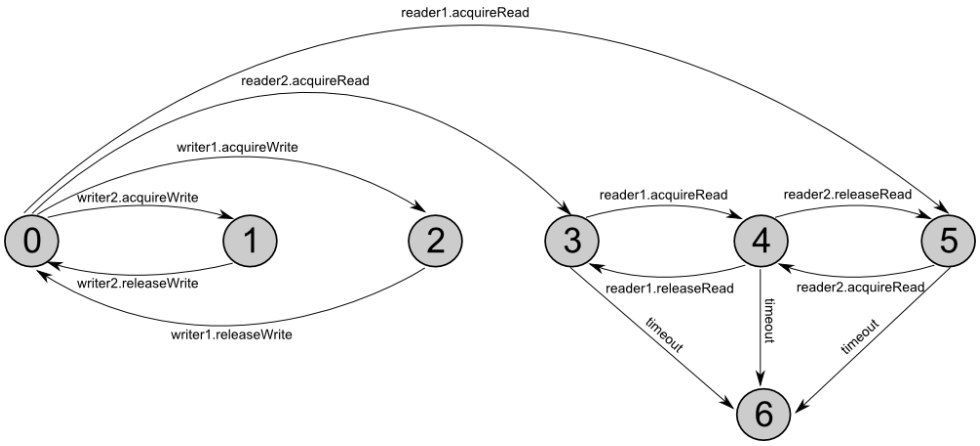

Figura 21. Detección de *starvation* del problema de lectores y escritores

Los ejemplos anteriores permiten demostrar la utilidad de los autómatas para el modelado de concurrencia, además de la detección de patrones específicos de eventos, en ambientes locales. El alcance del enfoque de este documento se expande hasta ambientes distribuidos, como se ha resaltado en secciones anteriores, aprovechando los niveles de abstracción que brinda esta metodología para la definición de patrones de detección de eventos complejos.

Caracterización de Errores Concurrentes en Sistemas Distribuidos

1. Problemas de Liveness

El término *liveness,* de acuerdo a [11] y a [12], hace referencia a la capacidad de una aplicación concurrente de ejecutarse de manera oportuna. Dentro de las aplicaciones concurrentes existen diferentes problemas típicos que causan que la aplicación no se ejecute de la manera esperada. Dichos problemas se caracterizan por ocurrir únicamente en estados específicos del sistema y por ocasionar fallas aparentemente inexplicables, lo que determina un alto grado de dificultad para su detección (existen otros problemas referentes a la sincronización de procesos concurrentes denominados *data races* tratados en secciones siguientes debido a que estos no figuran como problemas de *livness).* Estos problemas se dividen en tres: *Deadlock* (tratado en la sección anterior), *Livelock* y *Starvation*. A continuación se explican estos dos últimos problemas y se presentan casos en los cuales estos problemas se presentan y cuáles son sus consecuencias.

1.1. Livelock

Un hilo a menudo actúa como respuesta a la acción de otro hilo; si la acción del otro hilo es también una respuesta a la acción del primer hilo, entonces se puede producir un *livelock*. Al igual que con los *deadlock*, cuando ocurre un *livelock* los hilos no son capaces de seguir avanzando. Sin embargo, los hilos no están bloqueados, ellos están simplemente demasiado ocupados respondiendo al otro hilo para reanudar el trabajo.

En un ejemplo del mundo real, un *livelock* ocurre cuando dos personas, al encontrarse en un pasillo angosto avanzando en sentidos opuestos, cada una trata de ser amable moviéndose a un lado para dejar a la otra persona pasar, pero terminan moviéndose de lado a lado sin tener ningún progreso, pues ambos se mueven hacia el mismo lado, al mismo tiempo.

1.2. Starvation

Este problema hace referencia a la situación en la que un hilo es incapaz de obtener acceso regular a un recurso compartido y por ende no puede terminar sus

tareas. Esto ocurre cuando los recursos compartidos se encuentran deshabilitados por largos periodos de tiempo por otros hilos.

2. Definiciones de Condiciones de Carrera (Data Races)

De acuerdo a [16], una condición de carrera es aquella situación en la que dos hilos acceden de manera concurrente a un mismo recurso compartido, y al menos uno de esos hilos ejecuta alguna modificación a dicho recurso. Esta definición aplica en un contexto local, es decir, dos o más hilos en un mismo nodo accediendo de manera concurrente al mismo recurso.

Debido a que el objetivo de este documento es presentar una propuesta que solucione problemas como este en ambientes distribuidos, es necesario realizar una modificación a la definición anterior. Será entendido por condición de carrera entonces una situación donde para un mismo recurso compartido (entendiendo como recurso compartido un objeto o estructura que presenta una instancia en cada nodo), dos o más nodos acceden a dicho objeto (o a una posición X de la estructura en la instancia local que sea la misma para cada instancia de cada nodo) y al menos uno de los nodos realiza alguna modificación.

3. Errores de Concurrencia en Aplicaciones Distribuidas Reales

Los ejemplos que se tratan a continuación son situaciones comunes de errores concurrentes en sistemas distribuidos. En esta sección se pretende describir cada uno partiendo por contextualizar el ambiente en el que se generaron o se pueden generar estas situaciones, se plantea el problema que representa cada ejemplo, se ilustra dicho problema con una imagen con el fin de clarificar la situación descrita y se plantean los requerimientos que debería poder solucionar el lenguaje propuesto. En las secciones siguientes se propone una máquina para cada ejemplo que permita detectar la ocurrencia de estos errores.

3.1. Starvation

3.1.1. Hilos OOB - Versión EAP 5.0.0.CR1 de JBoss

Antes de abordar este problema es importante aclarar que existen dos tipos de mensajes en JBoss:

1. Mensajes Regulares
2. Mensajes *OOB* (*Out of Band Messages)*. Ignoran cualquier política de ordenamiento que tenga la pila de mensajes de JBoss. Estos son procesados por un pool de hilos dedicado, conocido como pool OOB. Utilizados en caso de que no se quiera que el procesamiento de los mensajes espere hasta que todos los mensajes enviados del mismo nodo sean procesados.

El problema de *starvation* del pool de hilos *OOB* que ocurre en esta versión de JBoss ocasiona *timeouts* y fallas de actualización de candados (*locks*). Este tipo de mensajes *OOB* son comúnmente utilizados en situaciones como el *Hertbeat*[1]. Suponiendo un ambiente donde un nodo P envía 5 mensajes y luego inicia una respuesta a una solicitud de *hearbeat* que envió otro nodo, puede ocurrir que el tiempo de procesamiento necesario para procesar los 5 mensajes enviados sea mayor al *timeout* del *heartbeat*, luego P puede ser marcado como inactivo en falso; en cambio, si la respuesta al *heartbeat* es marcada como *OOB*, esta va a ser procesada por el pool de hilos *OOB* y entonces será concurrente con el procesamiento de los mensajes enviados y el nodo no será marcado como inactivo.

Este problema surge como resultado de una solución a otro problema que ocasionaba que los algunos mensajes regulares no fueran procesados hasta que un nuevo mensaje regular era enviado. Este problema se ilustra con el siguiente ejemplo. Supongamos que se tiene un ambiente distribuido con las siguientes condiciones:

- Existen dos nodos *A* y *B*
- *B* envía tres mensajes por *multicast*: *B1* (*regular*), *B2* (*OOB*) y *B3* (*regular*)
- *A* recibirá los mensajes en el siguiente orden: *B1, B3, B2*
- *A* procesa *B1*

[1] *Heartbeat*. Mensaje que determina el estado de un nodo (Si se mantiene "vivo" o no).

- Como *B3* debe esperar a que el mensaje *B2* llegue, el hilo encargado de su procesamiento se detiene y el mensaje de *NAKACK* (confirmación) no es enviado

- *B2* llega al nodo *A* pero como este es *OOB*, una vez es procesado y eliminado de la pila de mensajes, el hilo se termina. Esto implica que *B3* no puede ser procesado hasta que llegue otro mensaje regular

La solución dada a este problema fue implementar un contador que es incrementado cada vez que un mensaje regular es adicionado a la pila de mensajes. Cuando más de un mensaje es removido el contador se reduce. Siempre que un hilo *OOB* vea el contador mayor a cero, este procesa el mensaje *OOB* al que se asocia y además procesa los mensajes regulares que existan en la pila; si no hay mensajes regulares el hilo se termina.

Como resultado de la solución anterior los hilos *OOB* pueden bloquear el procesamiento del protocolo *NAKACK*, pero las posibilidades de que ocurra esto son menores. Cuando se presenta una configuración síncrona de envío de mensajes hay bastantes mensajes *OOB* (uno por cada *commit* de dos fases y uno por cada respuesta *RPC*).

Abordando el problema principal, supongamos que hay un pool de cinco hilos *OOB* y dos nodos *A*, *B* que se comunican como indica la siguiente figura:

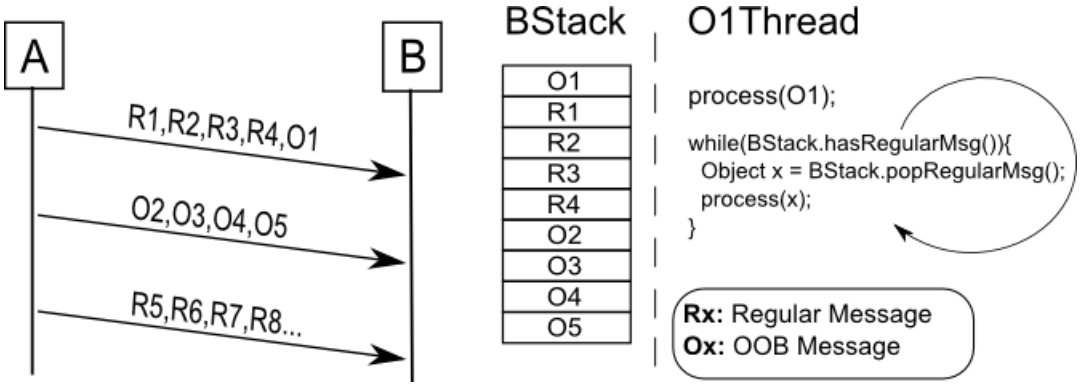

Figura 22. Comunicación entre los nodos *A* y *B* con mensajes *OOB*

Como se puede observar en la figura anterior, el nodo *A* envía como primera instancia cuatro mensajes regulares *R1, R2, R3, R4* y un mensaje *OOB O1* al nodo *B*. En la pila de *B* se observa el orden de llegada de estos mensajes, donde se indica que el mensaje O1 llega primero que los cuatro anteriores. Partiendo de la solución descrita anteriormente, el hilo *O1Thread* va a procesar el mensaje *O1*

que tiene asociado y luego va a procesar todos los mensajes regulares que existan en la pila.

Mientras esto ocurre supongamos que A envía otros cuatro mensajes *OOB: O2, O3, O4 y O5* hacia el nodo *B*. Aunque estos mensajes ignoran el orden de la pila, estos deben esperar a que *O1Thread* termine su procesamiento ya que este tiene el candado (*lock*) para manipular la pila. Supongamos ahora que *O1Thread* no ha terminado de procesar los primeros cuatro mensajes que llegaron y *A* decide enviar mensajes regulares a *B* de manera indefinida, como se muestra en la figura 22 (*R5, R6, R7, R8,…*); esto implica que los demás hilos *OOB* deben esperar a que *O1Thread* termine de procesar todos los mensajes regulares que existan en la pila para poder liberar el candado. Si lo anterior ocurre, cuando el nodo A quiera enviar un mensaje *OOB* a otro nodo no van a existir hilos *OOB* disponibles para atender esta solicitud, ya que el pool de hilos *OOB* está siendo totalmente utilizado, luego existen dos posibilidades: desechar esta solicitud (lo cual no es recomendado) o esperar hasta que un hilo *OOB* esté disponible; lo cual es un típico caso de *starvation*, tanto para esta última solicitud como para los hilos *OOB* que ya fueron asignados para atender solicitudes, pero, que están esperando a que se libere el candado para poder procesarse.

Continuando con el ejemplo anterior, esta situación actualmente se solucionó agrandando el tamaño del pool de hilos *OOB*, lo cual funciona porque permite atender más solicitudes para este tipo de mensajes sabiendo que, eventualmente el nodo A va a dejar de enviar mensajes regulares a B, permitiendo al hilo *O1Thread* liberar el candado.

Como se puede observar, la situación anterior presenta una serie de pasos que deben ocurrir en un orden determinado para que esta condición de *starvation* ocurra. Aprovechando esta característica, se puede proponer una máquina de estados determinista que permita modelar la ocurrencia de este problema. A continuación se presenta un autómata que permite capturar esta condición de *starvation*:

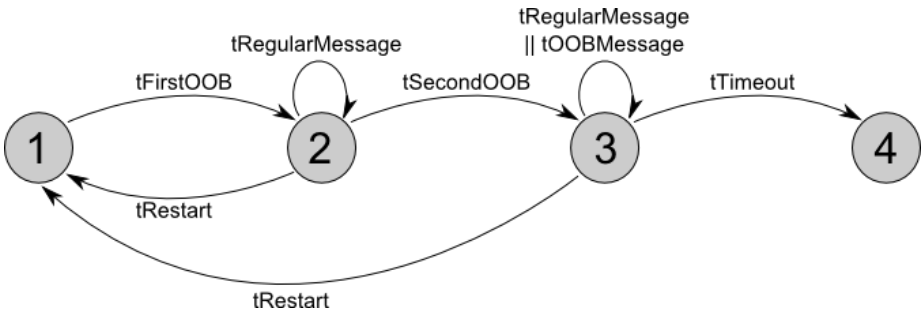

Figura 23. Autómata para la detección de la situación de *starvation*

El autómata anterior permite modelar la detección de la situación de *starvation* tratada anteriormente, de la siguiente manera:

La primera transición *tFirstOOB* hace referencia al evento en el que la primera solicitud de mensaje *OOB* ha ocurrido, implicando esto que el primer hilo *OOB* obtiene el candado para manipular la pila de mensajes.

En el estado 2, el primer hilo *OOB* denominado *O1Thread* se encarga de procesar tanto las solicitudes pendientes de mensajes regulares, como los nuevos mensajes regulares (*tRegularMessage*). Lo anterior implica que el candado sigue siendo posesión de *O1Thread.*En este mismo estado puede ocurrir el caso de que no hayan mensajes regulares por procesar en la pila de mensajes, luego sólo se procesa el mensaje *OOB* relacionado a *O1Thread;* en este caso, no hay posibilidad de que ocurra la condición de *starvation* y por eso se define la transición *tRestart* que reinicia el autómata para que esté listo a detectar una nueva posible situación de *starvation*. En caso de que en el estado 2 llegue otro mensaje *OOB*, ocurre la transición *tSecondOOB;* al ocurrir esta transición *O1Thread* no ha liberado el candado.

En el estado 3 también se incluye la transición *tRestart* porque puede darse el caso en el que inmediatamente luego de llegar el segundo mensaje *OOB,* el hilo *O1Thread* termine su ejecución, lo que significa que este hilo no va a retener el candado sobre la pila y por ende no hay posibilidad de situación de *starvation.* Es importante resaltar que *tRestart* hace referencia a la terminación del primer hilo *OOB* que se haya iniciado. La transición *tRegularMessage* también se define en este estado. Esta transición contempla el caso en el que luego de la llegada del segundo mensaje *OOB* aún existan mensajes regulares por procesar en la pila de mensajes. La transición *tOOBMessage* indica la llegada de más mensajes *OOB,* los cuales no van a ser procesados hasta que *O1Thread* libere el candado y el hilo relacionado a cada uno de estos mensajes sea ejecutado. Por último, la transición *tTimeOut* lleva al estado 4, el cual indica que ha ocurrido una condición de *starvation,* porque alguno de los hilos relacionados a los mensajes *OOB* que llegaron y están esperando por *O1Thread* para liberar el candado, ha llegado a su tiempo límite de espera (*tTimeout*); lo que significa que *O1Thread* ha retenido los recursos (que en este caso es el candado sobre la pila de mensajes) de manera excesiva, lo que inhabilita al menos un hilo de su mismo tipo, negándole el acceso a dichos recursos.

El autómata anterior funciona en la mayoría de los casos pero hay situaciones en las que se puede dar un estado de bloqueo, específicamente en el momento en el que ocurre la transición *tSecondOOB*, el hilo *O1Thread* al ser concurrente con el proceso de recepción de mensajes, puede terminar su ejecución en el mismo momento en el que *tSecondOOB* ocurre, lo que ocasiona una inconsistencia en la

máquina porque esta ocurrencia de eventos debería llevar al autómata al estado 1, sin embargo el autómata permanece en el estado 3, impidiendo que se analicen las secuencias de mensajes en el orden que se deberían procesar. Este inconveniente en particular necesita contemplar los tiempos de ejecución de los eventos, para que el consumo de estos sea exacto y preciso, pero como se aclara en la sección “Modelando Concurrencia con Autómatas”, en este modelo de autómatas no se contemplan este tipo de variables. Como trabajo futuro y solución alternativa se está estudiando la utilización de *push down automatas* [17] y de *lógica temporal* [18], lo que implica que el lenguaje propuesto debe soportar la implementación de este tipo de máquinas de estados.

3.2. Condiciones de Carrera

3.2.1. Ejemplo Trivial

Este es el código fuente de un ejemplo trivial de dos nodos que quieren realizar una operación de suma a una variable compartida entre ellos, para esto utilizan el protocolo *commit* de dos fases y dado una secuencia determinada se genera un *data race*. *X* es el entero compartido por dos host.

```
1. private void add(){
2.     int y = x;
3.     y++;
4.     PreparePhase();
5.     x = y;
6.     CommitPhase();
7.     System.out.println(x);
8. }
```

La siguiente figura ilustra el proceso de una determinada secuencia de eventos que generarían un *data race* cuando dos nodos ejecutan el código anteriormente expuesto.

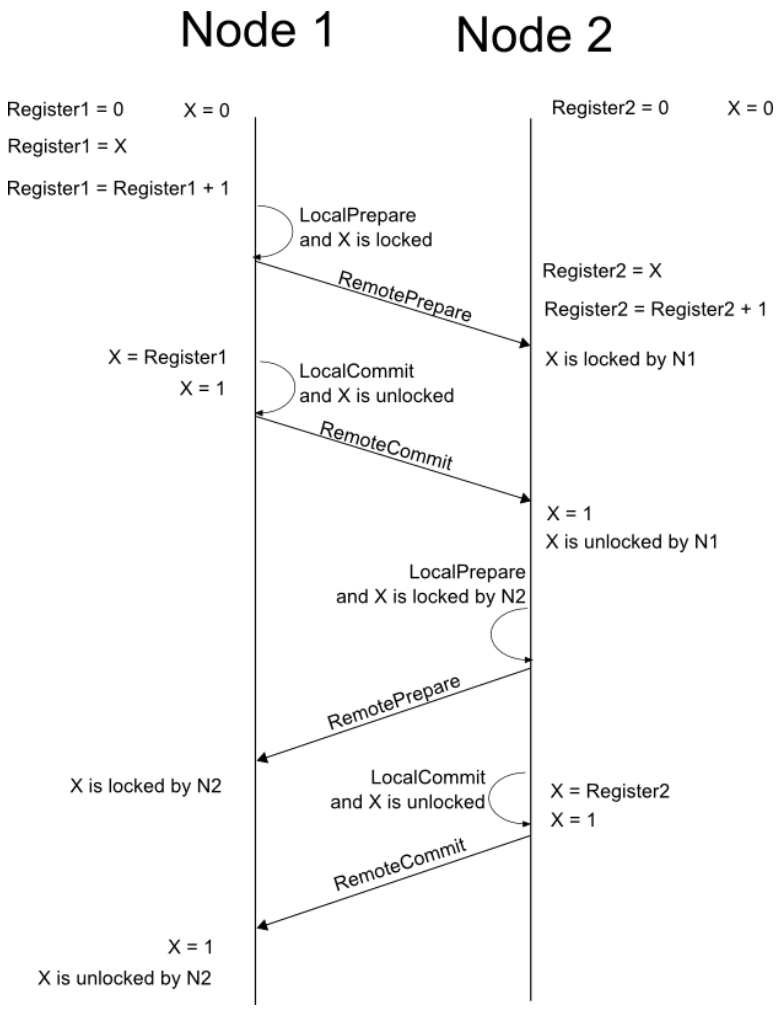

Figura 24. *Data race* en protocolo *commit* de dos faces

Teniendo en cuenta que cuando nos referimos a *N1* y *N2* son el nodo 1 y 2 respectivamente y donde el registro1 y registro2 son variables locales de cada nodo correspondiente, además el valor de la variable *X* comienza en 0 y asumimos que el candado que utiliza el protocolo *two phase commit* es solo para evitar que escriban en una posición de memoria. Los pasos que se observan en la figura 24 son los siguientes:

1. *Integer X* = 0; (objeto compartido)
2. *N1* lee el valor X de la memoria y lo guarda en el registro1: 0
3. *N1* incrementa el valor de *X* en el registro1: (contenido del registro1) + 1 = 1
4. *N1* hace un *local prepare* y bloquea el objeto compartido
5. *N1* envía un *remote prepare*
6. *N2* lee el valor de *X* de la memoria y lo guarda en el registro2: 0

7. *N2* incrementa el valor de *X* en el registro2: (contenido del registro2) + 1 = 1
8. El *remote prepare* de *N1* llega a *N2* y la variable compartida *X* es bloqueada
9. *N2* intenta obtener el candado de la variable compartida *X* pero no puede por que *X* ha sido bloqueado por otro nodo
10. *N1* hace un *local commit* y guarda el valor del registro1 en *X*
11. *N1* envía un *remote commit*
12. El *remote commit* de *N1* llega a *N2* y *X* se modifica con el valor de 1
13. El objeto compartido en *N2* ha sido desbloqueado por *N1* y *N2* que ha estado esperando, bloquea el objeto compartido
14. *N2* envía un *remote prepare*
15. *N2* hace un *local commit* y guarda el valor del registro2 en *X*

Finalmente el valor de *X* es 1 en vez del valor esperado de 2 y se genero un *data race* debido a que los dos nodos en un momento determinado accedieron a la variable *X* y uno de estos realizó una operación de escritura sobre esta variable. A continuación se presenta un autómata que modela la detección del problema anteriormente planteado.

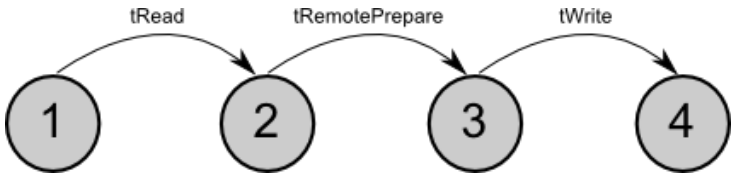

Figura 25. Detector de d*ata race* del protocolo *commit* de dos fases

Esta máquina tiene 4 estados y 3 transiciones, donde *tRead* es una guarda que valida que haya un evento de lectura de la variable *X* y este se genere en el caché remoto; luego, si se realiza un *remotePrepare()* en el host local, el autómata cambia a otro estado esperando a que la transición *tWrite* ocurra. Esta transición se cumple cuando hay un evento de escritura en la variable *X* donde el que genera este evento es el caché remoto. Cuando esta secuencia de eventos ocurre, es por que el host remoto leyó la variable y el host local escribió sobre esta, por lo tanto cuando el autómata esté en el estado 4, se ha detectado una posible condición de *data race*.

3.2.2. Proceso de Eviction en JBoss Cache Versiones 1.3.0.GA – 11.4.0.SP1

En JBoss Cache existe un proceso conocido como *Eviction* que se encarga de limpiar la información del caché, removiendo todos los datos almacenados que no son usados o referenciados por un determinado periodo. Esta política da prioridad de almacenamiento en el caché a la información más frecuentada.

Asumamos que existen dos nodos configurados para trabajar como Cachés. El primer nodo es el caché principal, el cual almacena la información más frecuentada por los otros nodos de la red, luego este nodo debe implementar lo política de *Eviction* cada tiempo t. El segundo nodo es un caché de nivel dos, el cual además de contener también información que es comúnmente frecuentada presenta una conexión directa con la base de datos. En caso de que el caché principal necesite información que no tiene almacenada, realiza una solicitud de esta al caché de nivel dos; si este tampoco presenta esta información almacenada realiza una solicitud a la base de datos. Este proceso es manejado por los *Cache Loaders*.

De acuerdo con la documentación de JBoss Cache [19], un *cache loader* es la conexión de JBoss Cache con un almacenamiento de datos. Hay dos momentos en los que se invoca un *cache loader*:

1. Para obtener datos del caché de nivel dos o de la base de datos en caso de que este último no tenga la información solicitada.

2. Cuando se efectúan cambios en la información guardada en el caché principal, se invoca un *cache loader* para almacenar dichas modificaciones en el caché de nivel dos y en la base de datos.

En las versiones 1.3.0.GA – 11.4.0.SP1 de JBoss Cache, cuando un hilo invoca a un *cache loader*, las instrucciones ocurren en el siguiente orden:

1. *Activation/CacheLoader - Passivation/CacheStore* [12]

2. *PessimisticLockInterceptor* [12]

Activation/CacheLoader hace referencia al primer caso descrito anteriormente en el que se invoca un *cache loader* y *Passivation/CacheStore* hace referencia al segundo caso de invocación descrito anteriormente.

PessimisticLockInterceptor es un interceptor que se encarga de manejar los bloqueos. Cuando se maneja un tiempo de bloqueo *Tx*, se obtienen los candados necesarios y una vez se cumple el tiempo *Tx* se liberan estos. Si no hay tiempo de bloqueo establecido, los candados son obtenidos por el método que los necesite y son liberados una vez este método retorne.

De acuerdo con lo anterior, antes de que un hilo adquiera el candado por medio del *PessimisticLockInterceptor*, se invoca el *Activation/CacheLoader* para conocer si la información solicitada se encuentra almacenada en el caché; de esta manera el *cache loader* sabrá si debe solicitar los datos al caché de nivel dos en caso de que estos no se encuentren almacenados.

Teniendo claros los conceptos anteriores, supongamos que contamos con un nodo *NodeA*, un caché principal *MainCache* y un dato *X* que se encuentra almacenado en el caché principal. Consideremos además el hecho de que existen dos hilos presentes en el caché principal; un hilo encargado del proceso de *eviction* (*EvictionThread*) y un hilo que se encarga de atender las solicitudes remotas de otros nodos para obtener información almacenada en el caché, denominado *HandlerThread*. Suponiendo que estos dos hilos inician de manera simultánea, una condición de *data race* potencial ocurrirá entre ambos hilos debido a que ambos acceden al mismo recurso compartido, que en este caso es el dato *X* almacenado en el árbol que utiliza el caché para contener la información, y al menos uno de estos hilos (*EvictionThread*) está realizando cambios a dicho dato. La siguiente figura ilustra este problema:

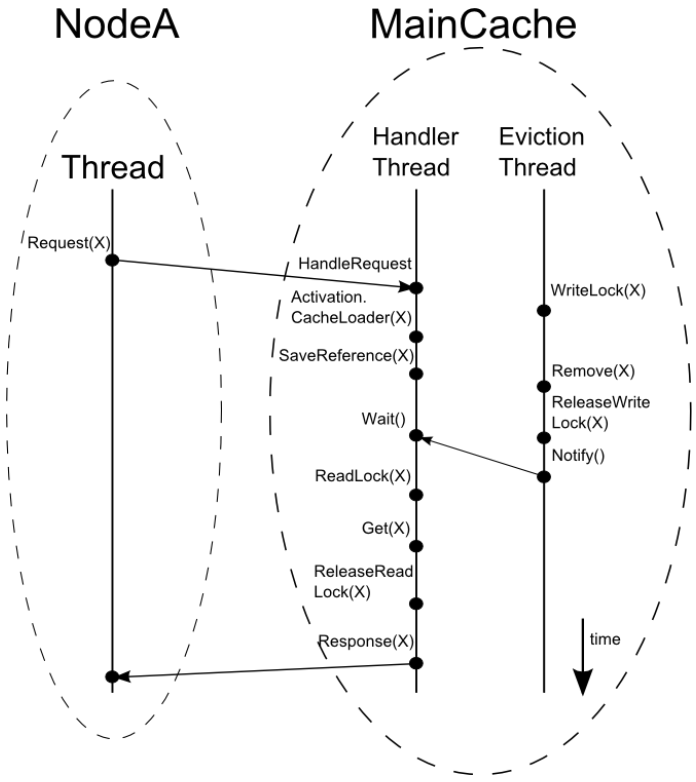

Figura 26. *Data race* en JBossCache 1.3.0.GA – 11.4.0.SP1

A continuación se explican los pasos descritos en la figura anterior:

1. Denominemos *X* como la información que va a ser eliminada y solicitada remotamente

2. *NodeA* solicita al caché principal la información *X*

3. En el cache principal, el hilo *HadlerThread* que se encarga de la solicitud de *NodeA* inicia de manera concurrente con el hilo *EvictionThread*

4. *EvictionThread* gana la carrera por adquirir el dato (*data race*) y adquiere un candado de escritura (*Write Lock*) sobre *X*

5. Aunque *HandlerThread* perdió la carrera, de acuerdo con el orden de las instrucciones de invocación de un *cache loader* descritas anteriormente, este invoca al *cache loader* de tipo *activation* antes de obtener el candado (*Activation.CacheLoader(X)*) o de consultar si este se encuentra disponible. En este momento los interceptores no tienen la necesidad de solicitar información debido a que *X* se encuentra almacenada en el caché; lo que ocurre entonces es que *HandlerThread* guarda una referencia de *X*

6. *HandlerThread* debe esperar por *EvictionThread* hasta que el candado de escritura se libere, luego este hilo se bloquea invocando el método *Wait()*

7. *EvictionThread* remueve *X* del árbol de almacenamiento de información del caché principal

8. *EvictionThread* libera el candado y notifica al *HandlerThread* este suceso

9. *HandlerThread* adquiere un candado de lectura (*Read Lock*) y como tiene una referencia a *X* no chequea si este continúa almacenado en el caché principal

10. *HandlerThread* invoca el método *get(X)* el cual retorna *null* debido a que X ha sido removido del caché principal. Este valor es enviado a *NodeA* lo cual probablemente generará una excepción en dicho nodo

    La condición de *data race* ocurre en los pasos 4 al 7 debido a que aunque *EvictionThread* adquiere un candado sobre *X*, *HandlerThread* inicia el interceptor de tipo *Activation/CacheLoader*, el cual accede al árbol contenedor para examinar si *X* se encuentra almacenada. Mientras el interceptor chequea si *X* está almacenada (evento de lectura), *EvictionThread* accede a la misma estructura para remover a *X* de esta (evento de escritura). El resultado de esto es que *HandlerThread* guarda una referencia no actualizada de *X* y por eso se retorna *null*.

Como se puede observar, la situación anterior presenta una serie de pasos que deben ocurrir en un orden determinado para que esta condición de *data race* ocurra. Aprovechando esta característica, se puede proponer una máquina de estados determinista que permita modelar la ocurrencia de este problema. A continuación se presenta un autómata que permite capturar esta condición de *data race*:

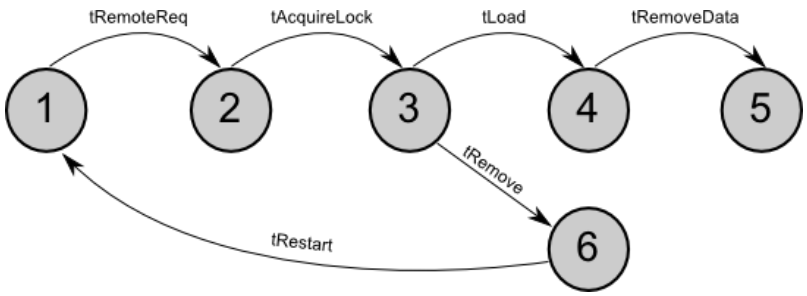

Figura 27. Detección de la condición de *data race* en JBossCache

El autómata anterior define los pasos necesarios para detectar la condición de carrera explicada anteriormente de la siguiente manera:

1. *tRemoteReq* indica que *NodeA* ha realizado la solicitud de *X* al caché principal. En este momento *HandlerThread* inicia su procesamiento

2. *tAcquireLock* es el evento que ocurre cuanto *EvictionThread* adquiere el candado de escritura sobre *X*

3. *tLoad* hace referencia a la invocación del interceptor *Activation/CacheLoader* por *HandlerThread*

4. *tRemoveData* se refiere al evento de eliminación de *X* del caché principal

Una vez ocurren los pasos anteriores se sabe que *HandlerThread* va a tener una referencia no actualizada de *X* y va a retornar *null*, lo que indica la ocurrencia de la condición de carrera. El estado 5 del autómata propuesto es aquel que captura dicho *data race*.

El estado 6 de la máquina es alcanzado cuando ocurre el evento *tRemove,* el cual indica que *EvictionThread* elimina a *X* del caché principal antes de que el interceptor invocado por *HandlerThread* guarde una referencia de este, lo cual no ocasiona un *data race*.

Por último, *tRestart* ocurre cuando *EvictionThread* libera el candado adquirido sobre *X*. Esta transición vuelve al inicio del autómata para que este quede continúe con el proceso normal de detección del *data race*. Cabe resaltar que aunque este problema ocurre en un sistema distribuido, la condición de *data race* que ocurre aquí se da sobre un mismo nodo (*Main Cache*), luego esta encaja con la definición planteada en [16] para este inconveniente.

Nuestra propuesta de lenguaje debe estar en capacidad de definir *breakpoints* complejos que permitan modelar máquinas de estados como la presentada en la figura 27, y una vez alcanzados dichos *breakpoints* se debe poder reaccionar a esta secuencia de eventos, por ejemplo, incluyendo la posibilidad de notificación a todos los nodos interesados sobre este problema o ejecutando una secuencia alternativa de eventos.

Un Lenguaje de Eventos para Aplicaciones Distribuidas

1. Sintaxis BNF del Lenguaje de Eventos Propuesto

$$
\begin{array}{lll}
Ep & ::= & \mathrm{call}(ESig) \\
 & | & \mathrm{host}(Group) \mid \mathrm{on}(Group[,\ Select]) \\
 & | & \mathrm{args}(\{Arg\}) \\
 & | & \mathrm{eq}(JExp,\ JExp) \mid \mathrm{if}(JExp) \\
 & | & Ep \mid\mid Ep \mid Ep\ \&\&\ Ep \mid !Ep \\
Group & ::= & \{\ Hosts\ \} \\
Hosts & ::= & \mathrm{localhost} \mid \mathrm{jphost} \mid \text{"}Ip{:}Port\text{"} \\
 & | & GroupId \\
GroupId & ::= & String \\
Select & ::= & JClass
\end{array}
$$

Este lenguaje soporta programación orientada a aspectos en un contexto distribuido donde un *pointcut* puede monitorear eventos en diferentes hosts. Un patrón de eventos puede involucrar diferentes hosts. Un *advice* (o *reacción*) puede ser ejecutada remotamente de forma sincrónica o asíncrona. La gramática de la anterior muestra los elementos esenciales de las definiciones de los *pointcuts* y los predicados en el lenguaje EKETAL. A continuación se explica cada símbolo terminal y no terminal de la gramática:

Ep: Es el símbolo no terminal que expresa los *pointcuts* y predicados

Call(ESig): Es la expresión para crear un joinpoint.

Host (Group): Este término define un grupo de host donde los joinpoints se originan ósea donde los *call* ocurren.

On(Group): Define en que grupo de hosts va a hacer ejecutado el *advice* (o *reacción*).

args({Arg}): Son los argumentos de un método *call*.

eq(JExp, JExp): Realiza una comparación de equivalencia utilizando expresiones Java

if(*JExp*): Se utiliza de la misma manera que el condicional if de Java

Group: Es un conjunto de hosts los cuales pueden ser construidos usando especificaciones como *localhost*, *jphost* y *direccionIp:puerto*. También los grupos pueden ser referenciados por su nombre (Los grupos con nombre son gestionados de forma dinámica dentro de un *advice* adicionando y eliminando hosts)

Los *poincuts* también pueden ser combinados usando operaciones lógicas como la negación, conjunción y disyunción. Para los eventos complejos se utiliza un autómata modelándolo con la siguiente sintaxis:

$$
\begin{array}{ll}
Seq & ::= [Id:]\ \mathrm{seq}(\{Step\}) \mid \mathrm{step}(Id,Id) \\
Step & ::= [Id:]\ Ep\ [>Target\ ] \\
Target & ::= Id \mid Id \mid\mid Target
\end{array}
$$

Secuencias (*Seq*) son definidas en términos de transiciones de un autómata finito determinista. Un *automaton* es un conjunto de transiciones *Step*, cada transición tiene una etiqueta *id* y su *pointcut Ep*. El constructor *step* identifica la transición del autómata que debe activar el *advice* (o *reacción*).

2. Reacciones

$$
\begin{array}{ll}
Rc & ::= [\mathrm{asyncex}]\ \mathrm{reaction}(\{Par\}) : EcAppl\ \text{'}\{\text{'}\ \{Body\}\ \text{'}\}\text{'} \\
EcAppl & ::= Id(\{Par\}) \\
Body & ::= JStmt \\
 & \quad \mid \ \mathrm{addGroup(Group)} \mid \mathrm{removeGroup(Group)}
\end{array}
$$

Estas son activadas cuando se cumple algún predicado o condición. La reacción (símbolo no terminal *Rc*) se define de la siguiente manera: se utiliza la palabra reservada *reaction*, un *pointcut* (*EcAppl*) que activa la reacción, un cuerpo (*Body*) construido con declaraciones Java. Por defecto, una reacción se ejecuta de forma síncrona, esto significa que la aplicación espera hasta que se complete la reacción para volver al comportamiento original o ejecutar la siguiente reacción. El programador puede también escoger una reacción asíncrona marcando la reacción con la palabra reservada *asyncex*. En el cuerpo de la reacción también

se puede agregar o eliminar un host a un grupo utilizando *addGroup*(Group) y *removeGroup*(Group) respectivamente.

3. Declaración de Clases

$$
\begin{array}{ll}
Ec & ::= \text{eventclass } Id \text{ '\{' } \{Decl\} \text{ '\}'} \\
Decl & ::= JVarD \mid EvDecl \mid Seq \mid MSig \\
EvDecl & ::= \text{event } Id(\{\text{Par}\})
\end{array}
$$

Para declarar las clases en este lenguaje se utiliza la palabra reservada *eventclass*, un nombre que identifica a esa clase y luego se escribe el bloque de código encerrado entre los símbolos “{“ y “}”. En el bloque de código se tienen declaraciones de variables en Java, declaraciones de los eventos mediante la palabra reservada *event* seguido de un identificador para el evento y un conjunto de parámetros que pertenecen al evento.

4. Significado de Cada Variable Expresada en el Lenguaje

$$
\begin{array}{ll}
MSig,\ FSig,\ ESig & ::= //\ method,\ field\ signatures\ (AspectJ\text{-}style) \\
Type & ::= //\ type\ expressions \\
Arg;Par & ::= //\ argument,\ parameter\ expressions\ (AspectJ\text{-}style) \\
Id & ::= //\ identifier \\
Ip,Port & ::= //\ integer\ expressions \\
JClass & ::= //\ Java\ class\ name \\
JExp & ::= //\ Java\ expressions \\
JStmt & ::= //\ Java\ statement \\
JVarD & ::= //\ Java\ variable\ declaration
\end{array}
$$

Para mejor entendimiento del lenguaje más adelante se utilizará este para capturar los problemas distribuidos que se han presentado en secciones anteriores.

Implementación del Prototipo del Lenguaje

1. Descripción General de la Arquitectura de KETAL

Con el fin de comprender más a fondo la utilidad de KETAL, se destacan sus características más relevantes.

1.1. The Event Model: Detecting Event Based Patterns

Es importante diferenciar los siguientes conceptos:

Evento: Acción atómica que ocurre en un proceso específico y en un nodo individual.

Mensaje: Paquetes de información enviados desde un proceso específico en un nodo individual hacia otro proceso corriendo en otro nodo.

La diferenciación anterior permite aclarar que este modelo trata las acciones específicas como envíos y/o recepción de mensajes, como eventos, y no se enfoca en la información transmitida por la red, la cual hace referencia al mensaje en si. Este modelo se caracteriza por lo siguiente:

1. Se lleva un registro de la información de localización (nodo donde ocurre el evento)
2. Se maneja una arquitectura distribuida. No hay un nodo central de control (evita los cuellos de botella). Esto permite que un nodo pueda ingresar y salir de la aplicación en cualquier momento
3. Cada nodo notifica a todos los demás la ocurrencia de sus eventos

1.2. The Pattern Model

- ✓ Se propone un modelo basado en un autómata finito
- ✓ Cada nodo implementa un *automaton* (autómata o máquina de estados). Este se encarga de recibir las notificaciones de eventos y de disparar las transiciones

1.2.1. Acciones Tomadas

- ✓ Cada nodo involucrado en la aplicación posee una instancia del autómata
- ✓ El autómata consume los eventos y dispara las transiciones. Dichos eventos pueden ser remotos
- ✓ Una acción puede estar ligada a una transición específica
- ✓ Las acciones ligadas a transiciones se ejecutan en la máquina local
- ✓ Para procesar un evento, el autómata puede establecer la localización del evento

1.3. The Dynamic Model

Con el fin de soportar orden para el manejo de relaciones causales, el modelo se basó en el vector de relojes de Mattern [15]. De acuerdo a esto, el modelo se extiende como sigue:

- Las transiciones de un autómata deben soportar *guards* (condiciones booleanas evaluadas antes de ejecutar la transición)
- La librería debe soportar los *guards causal* y *conc*. Esto para asegurar que los eventos ocurren únicamente si se cumplen estas condiciones de acuerdo al orden parcial definido por la relación de causalidad
- El modelo debe soportar constructores especiales para conceder el ordenamiento de eventos. Esto para asegurar que la relación causal no será únicamente evaluada sino concedida, reordenando los mensajes antes de transmitirlos al autómata

1.4. Implementación

Se proponen los siguientes dos constructores que manejan el ordenamiento de eventos:

Pointcuts sin reordenamiento – *SeqCausal:* Asegura que eventos causalmente relacionados ocurren pero no asegura que todas las secuencias se cumplan. Permite detectar llamadas inesperadas a métodos.

Pointcuts con reordenamiento – *SeqCausalOrder:* Asegura que cada evento es retrasado hasta que el evento que lo precede causalmente ocurra.

Una vez conocido el modelo de eventos propuesto por KETAL, se procede a hablar de la arquitectura que implementa esta librería para su funcionamiento. Esta se resume en la siguiente figura:

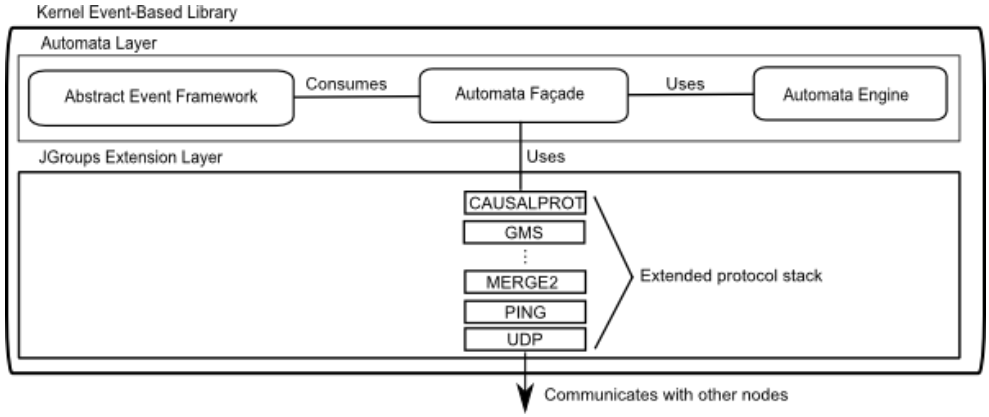

Figura 28. Arquitectura de la librería *kernel* basada en eventos

1.5. Arquitectura

La arquitectura de KETAL se divide en dos grandes bloques, *Automata Layer* y *Distribution Layer*.

1.5.1. Automata Layer

- *Abstract Event Framework*: Set de interfaces que permiten al desarrollador adaptar código existente para que sea modelado como eventos que son procesados por el autómata definido en la librería
- *Automata Facade*: Provee las abstracciones para manipular directamente la definición y ejecución de un *automaton*
- *Automata Engine*: Procesa acciones del autómata de acuerdo a la teoría de autómatas (librería de Anders Moller [20])

1.5.2. Distribution Layer

Provee las abstracciones necesarias principales para distribuir mensajes de eventos y para escuchar mensajes de eventos enviados por otros nodos. Provee una arquitectura distribuida basada en comunicación de grupo. Cada nodo necesita una instancia de esta librería. Dentro de la pila de protocolos que se manejan en esta capa se propuso la siguiente capa:

CAUSALPROT (Protocolos implementados que permiten el funcionamiento de los constructores descritos anteriormente). Esta capa cuenta con las siguientes características:

1. Adiciona información causal a mensajes de eventos; así el nodo receptor puede determinar si el mensaje llega en orden causal o concurrente. Evita falsos positivos

2. Adiciona información causal a mensajes de eventos, pero antes de entregar el mensaje al *Automata Layer* ordena los mensajes de acuerdo al orden causal parcial. Prohíbe falsos positivos y evita falsos negativos

1.6. ¿Qué Tipo de Problemas se Pueden Solucionar con la Librería KETAL?

Para entender el comportamiento de los sistemas distribuidos se presenta el siguiente ejemplo de una pila distribuida. Las líneas horizontales hacen referencia a cada host. Los eventos ocurridos son identificados por un punto y las líneas trasversales indican el tiempo que se toma un evento en llegar a un host determinado.

La figura 29 muestra el comportamiento ideal de una pila (la ocurrencia del evento *on* antes que el evento *push*, y la llamada a un evento pop luego de existir al menos un elemento en la pila).

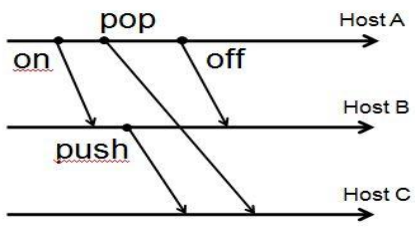

Figura 29. Eventos de una pila distribuida

En un contexto distribuido, el comportamiento descrito en la figura anterior se ve alterado por diferentes variables ajenas al programador, como el tiempo de propagación de un evento, retrasos en la red, entre otros. Esto trae como consecuencia la modificación del orden de ejecución esperado para dichos eventos, como se ilustra en la figura 30.

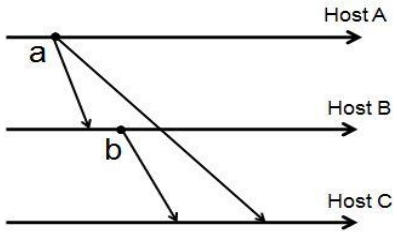

Figura 30. Eventos alterados de una pila distribuida

Como se observa en la figura anterior, el host *c* detecta el evento *b* antes que el evento *a*, cuando en realidad el evento *a* ocurrió primero que el evento *b*. Surge entonces el concepto de causalidad, el cual se define con las siguientes características. Sean *a*, *b* y *c* eventos, estos se encuentran relacionados causalmente si se cumple que:

1. *a* y *b* se encuentran en el mismo proceso, y *a* ocurre antes que *b*

2. *a* es el evento de enviar un mensaje desde un proceso y *b* es el evento de recibir el mismo mensaje por otro proceso

3. *a*, *b* y *c* son eventos tales que *a* ocurre antes que *b*, y *b* ocurre antes que *c*, entonces *a* ocurre antes que *c*. Esto quiere decir que la relación de causalidad entre dos eventos es transitiva

De aquí se derivan dos problemas comunes conocidos como falsos positivos y negativos. Un falso positivo es una secuencia de eventos que ocurre en un orden no esperado, pero que es reconocida por un autómata como una secuencia que ocurrió en el orden predestinado. Un falso negativo es una secuencia de eventos que ocurre en el orden esperado, pero que es reconocida por un autómata como una secuencia que no ocurrió en el orden predestinado. Los anteriores problemas se deben a los problemas de retrasos y propagación en la red. La librería KETAL se encuentra en capacidad de solucionar estos inconvenientes utilizando los constructores (*SeqCausal* y *SeqCausalOrder*) mencionados anteriormente.

2. Implementación de Llamados Síncronos y Asíncronos en la Librería

Como se ha mencionado en secciones anteriores, los CED existentes como EventJava carecen de mecanismos de sincronización en el envío de eventos en sistemas distribuidos. Actualmente hemos realizado una extensión a la librería KETAL para que permita a un nodo, esperar por la respuesta del procesamiento de algún evento que requiera esta acción en otros nodos que hagan parte del sistema distribuido. Específicamente, se han implementado dos métodos que permiten la sincronía de eventos: *multicastSync* y *multicastWithFutures*. La imagen siguiente permite visualizar un diagrama de clases que resume la extensión que se realizó sobre KETAL:

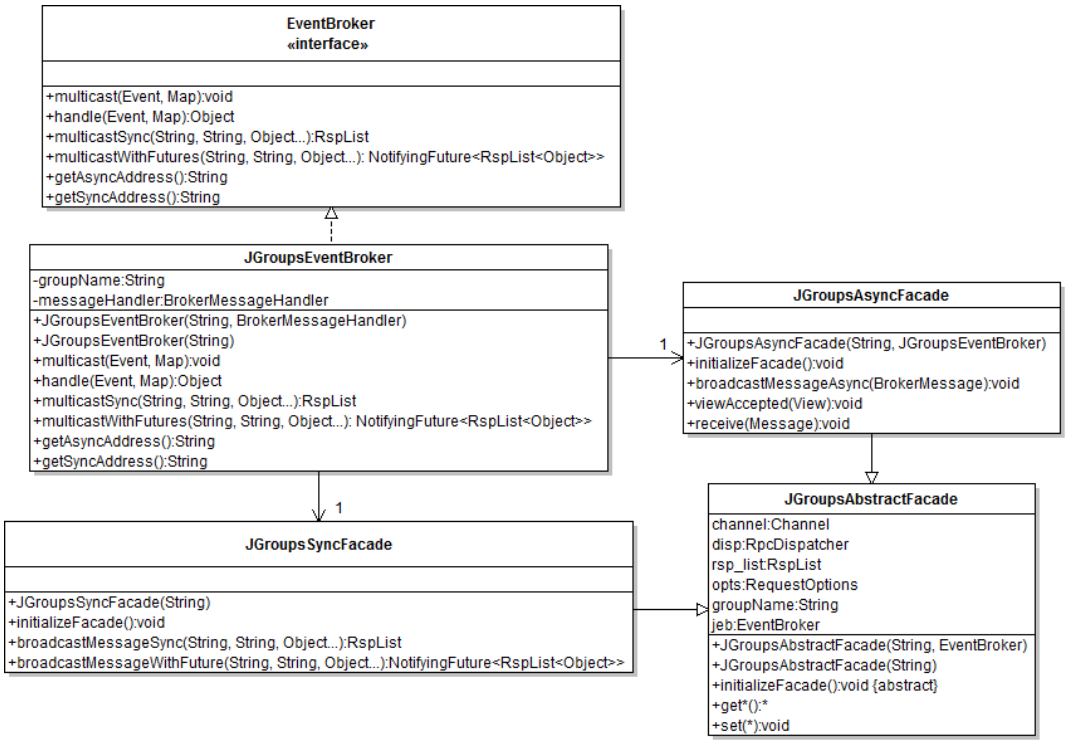

Figura 31. Extensión de KETAL para el soporte de sincronía

La interfaz *EventBroker* define los métodos que se encargan del envío de mensajes entre los nodos del sistema distribuido. El método *multicast* se encarga de la comunicación asíncrona y los métodos *multicastSync* y *multicastWithFutures* hacen referencia a la comunicación síncrona. Como se puede observar, se ha

definido una clase llamada *JGroupsEventBroker* que implementa la interfaz *EventBroker* y es la encargada de contener los servicios para el envío y recepción de mensajes. Esta clase contiene dos referencias, una hacia la clase *JGroupsAsyncFacade*, en la cual se definen los métodos que manejan la comunicación asíncrona, y la clase *JGroupsSyncFacade*, la cual contiene la definición de los métodos para controlar la sincronía. Como se tienen dos canales para enviar mensajes por cada nodo, dependiendo de la naturaleza del canal, el nombre de este se compone del grupo al que pertenece el nodo precedido por una etiqueta (SYNC o ASYNC). Dependiendo del método o servicio que se desee utilizar (síncrono o asíncrono), esta clase utiliza la fachada adecuada encargada de sobrescribir la implementación de este. Cada vez que se requiera que un nodo necesite enviar información hacia otros, Este nodo debe contener una instancia de la clase *JGroupsEventBroker*. Antes de describir en detalle la implementación de los métodos, se puede observar en la figura 31 que la clase *JGroupsAbstractFacade* define unos atributos que permiten la manipulación de los mensajes, los cuales se describen a continuación:

channel: Define el canal por el cual se van a enviar y a recibir los mensajes de un nodo específico

disp: Esta clase se encarga de manejar los llamados a procedimientos remotos (RPC, por sus siglas en inglés). Se encarga de ejecutar los métodos que sean necesarios en máquinas externas a la local.

rsp_list: Lista que contiene los resultados de las ejecuciones remotas del método ejecutado por el *disp.*

opts: Esta clase permite definir las políticas de envío de mensajes, las cuales son:

- *GET_FIRST*: Retorna la primera respuesta recibida por cualquier nodo. Esto detiene el procesamiento del nodo que envía el mensaje hasta que obtenga una respuesta.
- *GET_ALL*: Retorna todas las respuestas. Esto detiene el procesamiento del nodo que envía el mensaje, hasta que todos los nodos del grupo devuelvan la respuesta del método que se debe procesar.
- *GET_NONE*: El nodo que envía el mensaje no espera por ninguna respuesta y continúa con su procesamiento normal

groupName: Cadena que contiene el nombre del grupo al que pertenece la máquina o host

Es importante aclarar que las clases *Channel*, *RCPDispatcher*, *RspList* y *RequestOptions* hacen parte de la librería de *JGroups* [21], en este caso, la

versión 3.0.0.CR1. A continuación, se procede a explicar la implementación de los métodos para controlar la comunicación síncrona y asíncrona.

2.1. Llamados Síncronos

El primero de los métodos síncronos es *multicastSync*. Este método definido en la interfaz *EventBroker* realiza un llamado al método *broadcastMessageSync* definido en la fachada síncrona. El encabezado de este método es el siguiente:

```
public RspList broadcastMessageSync (String, String, Object…);
```

El primer parámetro hace referencia al nombre de la clase que contiene el método que se debe ejecutar. El segundo parámetro contiene el nombre del método que se debe ejecutar y el último parámetro es un arreglo de objetos que hacen referencia a los parámetros del método que se pretende ejecutar. Cuando se ejecuta este servicio, el nodo que lo ejecuta espera por los demás nodos a que procesen este mismo servicio, dependiendo de la política implementada. Las respuestas de todos los nodos se guardan y retornan en una colección de tipo *RspList*, las cuales pueden ser consultadas por el nodo originario del mensaje. La definición de este método es la siguiente:

```
1. public RspList broadcastMessageSync(String class_name, String
   method_name,
2. Object... method_parameters) {
3. MethodCall call;
4. try {
5. Class<?>[] array = new Class<?>[method_parameters.length];
6. for (int i = 0; i < array.length; i++) {
7. array[i] = method_parameters[i].getClass();
8. }
9. call                              =                              new
   MethodCall(Class.forName(class_name).getMethod(method_name,
   array));
10.   call.setArgs(method_parameters);
11.   rsp_list = disp.callRemoteMethods(null, call, opts);
12. } catch (NoSuchMethodException e) {
13. e.printStackTrace();
14. } catch (SecurityException e) {
15. e.printStackTrace();
16. } catch (Exception e) {
17. e.printStackTrace();
18. }
19. return rsp_list;
```

```
20. }
```

En línea 3 se define una clase *MethodCall call* la cual se encarga de buscar en una clase específica el método que recibe por parámetro con los argumentos indicados.

La línea 5 define un arreglo que va a contener los tipos de clases de los parámetros del método que va a ser llamado de manera remota.

Las líneas 6 a 8 se encargan de llenar el arreglo anterior obteniendo las clases de cada parámetro guardado en *method_parameters*.

La línea 9 crea la instancia del objeto *call* indicando que el método que se va a ejecutar de manera remota se encuentra contenido en la clase actual, con el nombre contenido en *method_name* y con tipos de parámetro iguales a los contenidos en *array.*

La línea 10 asigna el valor a los parámetros del método, los cuales están contenidos en *method_parameters.*

En la línea 11 se invoca el método *callRemoteMethods* del atributo *disp* explicado anteriormente, el cual recibe un arreglo de los host destino donde se debe ejecutar el método; en este caso *null,* lo que indica que se debe ejecutar en todos los host del grupo; recibe la definición del objeto *call* que ha sido configurado anteriormente y por último recibe las políticas *opts* (también descritas anteriormente) que se deben implementar para el set de host destino especificado. Este método detiene el procesamiento del método hasta que todos los hosts destino ejecuten el método y retornen el resultado de esta operación.

Por último, la línea 19 retorna el resultado de la ejecución del método en los hosts destino guardado en una lista tipo *RspList*.

2.1.1. Usando Futuros

Un futuro (*Future*) [22] es una estructura de datos utilizada para recuperar el resultado de una operación concurrente. Se proporcionan métodos para verificar si la operación esta completa, para esperar su finalización y para recuperar el resultado de esa operación. El resultado solo puede ser recuperado mediante el método *get()* cuando la operación se ha completado, si no se ha completado todavía entonces se bloquea hasta que este lista. Para cancelar esta operación se puede utilizar el método *cancel()* y además existen métodos adicionales que son proporcionados como *isDone()* y *isCancelled()* para determinar si la tarea se completo normalmente o fue cancelada. Una vez que la operación se ha completado no puede ser cancelada.

En nuestro caso particular, utilizamos futuros para esperar por una respuesta a los eventos que son enviados sin interrumpir la ejecución del hilo que envía los mensajes, para luego obtener la respuesta de estos mensajes utilizando el método *get()*; en caso tal de que el mensaje de respuesta no ha llegado, entonces el hilo se bloquea en espera de su llegada. Además, podemos utilizar varios métodos que nos proveen información acerca de si el mensaje de respuesta ha llegado o si la operación ha sido cancelada. Es importante resaltar que aunque la ejecución de los futuros es asíncrona (tiempo aleatorio y de manera independiente), estos son catalogados como mecanismos síncronos porque permiten esperar por la respuesta de procesamiento de algún evento particular en otros hosts diferentes al local. A continuación se muestra la implementación de esta característica en los métodos síncronos.

El segundo método síncrono es *multicastWithFutures*. Este método definido en la interfaz *EventBroker* realiza un llamado al método *broadcastMessageWithFuture* definido en la fachada síncrona. El encabezado de este método es el siguiente:

```
public                                NotifyingFuture<RspList<Object>>
broadcastMessageSync(String, String, Object…);
```

Al igual que el método anterior, el primer parámetro hace referencia al nombre de la clase que contiene el método que se debe ejecutar, el segundo parámetro contiene el nombre del método que se debe ejecutar y el último parámetro es un arreglo de objetos que hacen referencia a los parámetros del método que se pretende ejecutar. La diferencia crucial entre este método y el anterior, es que una vez se invoca al método de manera remota en todos los host destino, el procesamiento del nodo originario del mensaje continúa con su flujo de ejecución. La ventaja de esto es que pueden existir ocasiones que no requieran la respuesta inmediata de todos los host para continuar la ejecución, sino que esta debe ser consultada posteriormente. El funcionamiento de este método permite entonces que una vez ejecutado el llamado al método, cada vez que un nodo destino termine la ejecución del método, este retorna el resultado de la operación al nodo originario, la cual es guardada de manera asíncrona en una lista de tipo *NotifyingFuture<RspList<Object>>;* estos resultados pueden ser consultados cuando resulte conveniente. La definición de este método es la siguiente:

```
1. public                             NotifyingFuture<RspList<Object>>
   broadcastMessageWithFuture(     String     class_name,     String
   method_name, Object... method_parameters) {
2. MethodCall call;
3. NotifyingFuture<RspList<Object>> futures = null;
4. try {
5. Class<?>[] array = new Class<?>[method_parameters.length];
6. for (int i = 0; i < array.length; i++) {
7. array[i] = method_parameters[i].getClass();
```

```
8. }
9. call                                =                                new
   MethodCall(Class.forName(class_name).getMethod(method_name,
   array));
10.   call.setArgs(method_parameters);
11.   futures = disp.callRemoteMethodsWithFuture(null, call, opts);
12. } catch (NoSuchMethodException e) {
13. e.printStackTrace();
14. } catch (SecurityException e) {
15. e.printStackTrace();
16. } catch (Exception e) {
17. e.printStackTrace();
18. }
19. return futures;
20. }
```

En línea 2 se define una clase *MethodCall call* la cual se encarga de buscar en una clase específica el método que recibe por parámetro con los argumentos indicados.

La línea 3 define la lista que va a guardar los resultados de las ejecuciones remotas de manera asíncrona.

La línea 5 define un arreglo que va a contener los tipos de clases de los parámetros del método que va a ser llamado de manera remota.

Las líneas 6 a 8 se encargan de llenar el arreglo anterior obteniendo las clases de cada parámetro guardado en *method_parameters*.

La línea 9 crea la instancia del objeto *call* indicando que el método que se va a ejecutar de manera remota se encuentra contenido en la clase actual, con el nombre contenido en *method_name* y con tipos de parámetro iguales a los contenidos en *array.*

La línea 10 asigna el valor a los parámetros del método, los cuales están contenidos en *method_parameters.*

En la línea 11 se invoca el método *callRemoteMethodsWithFuture* del atributo *disp* explicado anteriormente, el cual recibe un arreglo de los host destino donde se debe ejecutar el método; en este caso *null,* lo que indica que se debe ejecutar en todos los host del grupo; recibe la definición del objeto *call* que ha sido configurado anteriormente y por último recibe las políticas *opts* (también descritas anteriormente) que se deben implementar para el set de host destino especificado. Como se explico anteriormente, este llamado permite que el nodo originario continúe con su hilo de ejecución.

Por último, la línea 19 retorna el resultado de la ejecución del método en los hosts destino guardado en la lista *futures*. En caso de que la lista no contenga resultados, esta se retorna vacía en primera instancia y es llenada a medida que los hosts destino ejecutan el método invocado. Es importante aclarar que para el llamado de estos métodos funcione, estos métodos deben ser estáticos. Esta restricción se da porque para poder identificar y ejecutar los métodos, estos no pueden ser parte de instancias de un objeto si no que deben ser instancias de clase.

2.2. Llamados Asíncronos

Uno de los métodos para realizar un llamado asíncrono es *multicast*. Este método definido en la interfaz *EventBroker* realiza un llamado al método *broadcastMessageAsync* definido en la fachada asíncrona. El encabezado de este método es el siguiente:

```
public void broadcastMessageAsync(BrokerMessage m);
```

El parámetro que tiene este método hace referencia a un objeto de la clase *BrokerMessage* el cual encapsula un evento y el tipo de mensaje. La definición de este método es la siguiente:

```
1. public void broadcastMessageAsync(BrokerMessage m) {
2.     try {
3.         channel.send(null, m);
4.     } catch (Exception ex) {
5.         logger.error("Error no message delivery: ", ex);
6.     }
7.   }
```

Se llama al método *send* de la clase *org.jgroups.Channel* que pertenece a la librería de *JGroups* [21], el cual recibe por parámetro un valor *null* indicando el envió de la información en *broadcast* (a todos los host que pertenecen al grupo en el que se encuentra el host emisor) y también se le pasa por parámetro el *BrokerMessage* que va a hacer enviado con la información necesaria. Y por ultimo en caso de error se captura la excepción y se guarda en un log.

Depuración y Pruebas Utilizando el Lenguaje

1. Detección de Condiciones de Carrera en Sistemas Distribuidos

Una vez conocida la definición formal del lenguaje propuesto de eventos, el siguiente paso consiste en definir una posible implementación de las máquinas de estado propuestas en la sección 4.3 utilizando dicho lenguaje, con el fin de demostrar la utilidad y funcionalidad de este para la detección y reacción a este tipo de problemas. Una forma para detectar el *data race* que se plantea en el ejemplo trivial utilizando nuestro lenguaje es la siguiente:

```
1.  eventclass DataRaceTestTrivial{
2.
3.  event remotePrepare();
4.  event read(FQN x);
5.  event write(FQN x);
6.
7.  s1: seq(
8.     tRead: call(read(x)) && host(!local) > tRemotePrepare
9.     tRemotePrepare:  call(remotePrepare())  &&  host(local)  >
    tWrite
10.    tWrite: call(write(x)) && host(!local)
11. ) && step(s1, tWrite);
12.
13. public void generateDataRace(){
14.    String msg = "Potencial Data Race ha sido detectado";
15.
16.    reaction(s1): showMessage(msg){
17.       System.out.println(msg);
18.    }
19. }
20.  }
```

Se declaran tres eventos *remotePrepare()*, *read(FQN x)* y *write(FQN x)*, los dos últimos con un parámetro que es la variable *X* que se comparte entre los dos nodos. Luego se declara la maquina de estados *s1* que tiene 4 estados y 3 transiciones, el primer estado tiene la transición *tRead* que se acepta si hay un evento donde se lea la variable *X* en el host remoto, luego si se genera un *remotePrepare*() en el host local este cambia a otro estado esperando a que la transición *tWrite* ocurra, para que se acepte esta transición se debe hacer un

llamado a *write*(x) en el host remoto, si se cumple esto entonces la maquina de estados detecto un posible *Data Race*.

La siguiente definición de *eventclass* tiene como objetivo detectar la ocurrencia del *data race* tratado en la sección 3.3.2., referente al proceso de *Eviction* en JBoss Cache en las versiones 1.3.0.GA – 11.4.0.SP1:

```
1.   eventclass DataRaceTest{
2.
3.   event cacheLoader(FQN x);
4.   event removeData(FQN x);
5.   event writeLock(FQN x);
6.   event request(FQN x);
7.   event writeUnlock(FQN x);
8.
9.   s1: seq(
10.  tRemoteReq: call(request(x)) && host(A) > tAcquireLock
11.  tAcquireLock: call(writeLock(x)) && host(src) > tLoad || >
tRemove
12.  tLoad: call(cacheLoader(x)) && host(src) > tRemoveData,
13.  tRemoveData: call(removeData(x)) && host(src)
14.  tRemove: call(removeData(x)) && host(src) > tRestart
15.  tRestart: call(writeUnlock(x)) && host(src)
16.
17.  ) && step(s1, tRemoveData);
18.
19.  public void generateDataRace(){
20.       String msg = “DataRace detected for node: ”+x;
21.
22.        reaction(s1): showMessage(msg){
23.       System.out.println(msg);
24.        }
25.  }
26.  }
```

Antes de proceder a la explicación del código anterior, es importante resaltar que los nombres de las transiciones de la secuencia *S1* coinciden con los nombres de las transiciones del autómata de la figura 27; esto con el fin de dar consistencia a la información presentada anteriormente. Una vez aclarado esto, recordemos que *X* es la información que va a ser eliminada por el proceso de *Eviction* y que además esta es solicitada de manera remota.

En la línea 3 se define el evento que se ejecuta cuando *HandlerThread* inicia el proceso de *Activation.CacheLoader(X)* para comprobar si *X* se encuentra almacenado en el caché principal.

La línea 4 hace referencia al evento que ocurre cuando *EvictionThread* se encuentra removiendo *X* de la estructura de almacenamiento del caché principal.

En la línea 5 se define el evento que ocurre cuando *EvictionThread* adquiere el candado (*write lock*) sobre *X* para modificarla.

La línea 6 define el evento que ocurre cuando *NodeA* solicita al caché principal la información almacenada en *X.*

La línea 7 define el evento que ocurre cuando *EvictionThread* libera el candado (*write lock*) adquirido sobre *X.*

En las líneas 9 a 17 se define la secuencia que declara la máquina de estados que permite detectar la condición de *data race.* Esta secuencia trabaja de la siguiente manera:

La línea 10 define la transición que captura la solicitud remota de *NodeA* por la información *X.*

En la línea 11 se define la transición que captura el evento *writeLock(X).* Esta es la primera transición luego de la solicitud remota de *NodeA* ya que *EvictionThread* gana la carrera de adquisición del candado sobre *X.* Si *EvictionThread* remueve la información almacenada en *X* antes de que *HandlerThread* invoca el *cache loader*, no habrá condición de *data race* y la transición de la línea 14 (*tRemove*) se ejecutará.

La línea 12 define la transición que captura el evento *cacheLoader(X).* Este evento ocurre incluso después de que *EvictionThread* ha bloqueado el acceso a *X,* porque esta instrucción se ejecuta en *HandlerThread* antes de evaluar si hay algún candado obtenido o activado sobre *X. HandlerThread* en este momento guarda una referencia de *X* y espera a que *EvictionThread* libere el candado.

La línea 13 define la transición que captura el evento *removeData(X).* Este evento es ejecutado por *EvictionThread* justo después de que *HandlerThread* ha guardado la referencia de *X.* Esta acción elimina a *X* de la estructura de almacenamiento del caché principal y no actualiza la referencia que tiene *HandlerThread* de *X,* lo que crea una inconsistencia.

En la línea 14 se define la transición que captura el evento *removeData(X).* Esta acción se ejecuta por *EvictionThread* justo antes de que *HandlerThread* invoque al proceso del *cache loader* que conlleva a guardar la referencia sobre *X.* Esta transición significa que *EvictionThread* termina su proceso de ejecución de manera normal y por ende no se crearán inconsistencias con *HandlerThread* al acceder a *X.*

La línea 15 define la transición que captura el evento *writeUnlock(X).* Esta acción ocurre justo después de que *EvictionThread* remueve *X* de la información almacenada en el caché principal.

La línea 17 se encarga de manejar la ocurrencia de la condición de *data race* cuando la transición *tRemoveData* ocurre.

En la línea 22 se define el mensaje que será mostrado cuando la condición de *data race* ocurre y la línea 23 muestra este mensaje.

Los *eventclass* definidos anteriormente permiten ilustrar cómo por medio de la utilización del lenguaje propuesto se puede modelar, de manera determinista, una máquina de estados que permite detectar y reaccionar a la ocurrencia de las condiciones de *data race* descritas anteriormente.

2. Detección de Deadlock Distribuido

Solución en el lenguaje propuesto para la detección del *deadlock* tratado en secciones anteriores.

```
1. eventclass DeadlockTest{
2.
3.     event prepare();
4.     event commit();
5.
6.     s1: seq(
7.        tPrep:  call(prepare())  &&  host(src)  >  tCommit  ||
          t2ndPrep,
8.        tCommit: call(commit()) && host(targ) > tPrep,
9.        t2ndPrep: call(prepare()) && host(src)
10.    ) && step(s1, t2ndPrep);
11.
12.    public void generateDeadlock(){
13.       String msg = “Deadlock detected!”;
14.
15.       reaction(s1): showMessage(msg){
16.           System.out.println(msg);
17.       }
18.    }
19.  }
```

Primero se declaran dos eventos que son *prepare()* y *commit()*, luego se implementa una secuencia *s1* (maquina de estados finitos) con tres estados y tres

transiciones. El primer estado acepta un *call* al método *prepare*() del *src* (caché fuente). Una vez el método es recibido, la maquina de estados cambia a un estado que acepta transiciones *tCommit* y *t2ndPrep*. Si *targ* (caché destino) cumple con la transición *tCommit*, que es el comportamiento normal, la maquina retorna al primer estado. Finalmente, si la secuencia detecta después del primer *tPrep* un *t2ndPrep* en el cache fuente, la máquina de estados reconoce un estado de *deadlock*. Hay que tener en cuenta que la definición de la secuencia debe ser ordenada causalmente con el fin de asegurar que los eventos serán detectados en el orden correcto en cualquier entorno distribuido.

Conclusiones

En este trabajo de grado, hemos presentado el diseño de un lenguaje de programación orientado a eventos, el cual permite la declaración, ejecución, detección y coordinación de patrones de eventos complejos en sistemas distribuidos; para lo cual se extendió de la librería KETAL. Esta permite el uso de autómatas finitos deterministas con guardas para la captura de eventos complejos, y utiliza además, los relojes vectoriales de Mattern para el ordenamiento causal de estos eventos. Hemos realizado una extensión a librería de KETAL, en la cual se implementó una nueva característica para el llamado síncrono y asíncrono con futuros para el envió de eventos; cabe resaltar que los detectores de eventos compuestos (Composite Event Detection) existentes como EventJava carecen de estos mecanismos de sincronización. Hemos presentado un análisis de varios problemas concurrentes en aplicaciones distribuidas como *liveness* y *data races*, y se mostró cómo el lenguaje planteado puede capturar estos errores mediante el modelamiento de autómatas.

La utilización de autómatas deterministas para modelar este tipo de problemas concurrentes en ambientes distribuidos resulta bastante conveniente porque, además de que permite concebir y crear de manera sencilla la interacción entre eventos, la complejidad del código resultante depende directamente del orden de ocurrencia único de los eventos para que se presente el problema abordado, independientemente de si dichos eventos hacen parte de diferentes procesos concurrentes o no. Sin embargo, el mecanismo de modelado de estos problemas mediante autómatas presenta restricciones, ya que aunque estos permiten abordar y modelar diferentes problemas, limitan el alcance del lenguaje para tratar situaciones con casos particulares; por ejemplo, el caso donde se necesite contemplar los tiempos de ejecución de los eventos, para que el consumo de estos sea exacto y preciso. En este modelo de autómatas no se contemplan este tipo de variables. Como trabajo futuro, se estudiarán la incorporación de otras abstracciones como *push down automatas* y *lógica temporal* para extender la funcionalidad y el alcance de esta herramienta, permitiendo superar las restricciones mencionadas, y además se espera la implementación completa del lenguaje y la utilización de este como base para un depurador distribuido en IDE´s de Java como Eclipse.

Referencias

[1] A. S. Tanenbaum, Sistemas Operativos Distribuidos, Primera ed., Prentice Hall, 1996.

[2] A. Tanenbaum y M. V. Steen, Distributed Systems: Principles and Paradigms, 2nd Edition ed., Upper Saddle River, NJ, USA: Prentice Hall, Inc., 2006.

[3] J. G. Sopeña, F. J. Soriano Camino y J. J. Moreno Navarro, «Informe de Vigilancia Tecnológica Madri+d "Tecnologías Software Orientadas a Servicios",» Fundación Madri+d para el Conocimeinto Velázques, 76. E-28001, Madrid.

[4] E. Gamma, R. Helm, R. Johnson y J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley, 1994.

[5] D. Alur, D. Malks y J. Crupi, Core J2EE Patterns: Best Practices and Design Strategies, 2nd Edition ed., Mountain View, CA, USA: Prentice Hall/Sun Microsystems Press.

[6] G. Hohpe y B. Woolf, Enterprise Integration Patterns: Designing, Building and Deploying Messaging Solutions, Boston, MA, USA: Addison-Wesley Longman Publishing Co., 2003.

[7] D. Navarro Benavides, M. Südholt, R. Douence y J. M. Menaud, «Invasive Patterns for Distributed Programs,» *In Proc. of the 9th International Symposium on Distributed Objects and Applications (DOA'07). LNCS, Springer Verlag,* nº ISSN: 0302-9743, Nov. 2007.

[8] K. R. Jayaram y P. Eugster, «EventJava: An Extension of Java for Event Correlation,» *Drossopoulou, S. (ed.) ECOOP 2009. Springer, Heidleberg,* vol. 5653, pp. 570-594.

[9] K. Jayaram y P. Eugster, «Scalable Efficient Composite Event Detection,» *D. Clarke and G. Agha, editors, 12th International Conference on Coordination Models and Languages (COORDINATION 2010), LNCS 6116, Amsterdam, The Netherlands,* pp. 168-182, 2010.

[10] D. L. Benavides Navarro, A. Barrera, K. O. Garces y H. Arboleda, «Detecting and Coordinating Complex Patterns of Distributed Events with KETAL,»

Proceedings of the 2011 Latin American Conference in Informatics (CLEI), vol. 281, pp. 127-141, 2011.

[11] O. a. i. affiliates, «The JavaTM Tutorials,» 1995, 2012. [En línea]. Available: http://docs.oracle.com/javase/tutorial/essential/concurrency/liveness.html.

[12] J. Magee y J. Kramer, «Concurrency State Models & Java Programs,» *Department of Computing, Imperial College London, UK.*

[13] D. L. Navarro Benavides, R. Douence y M. Südholt, «Debugging and Testing Middleware with Aspect-Based Control-Flow and Causal Patterns,» *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware,* pp. 193-202, 2008.

[14] L. Lamport, «Time, Clocks and the Ordering of Evens in a Distributed System,» *Commun, ACM 21,* pp. 558-565, 1978.

[15] F. Mattern, «Virtual Time and Global States of Distributed Systems,» *Proceedings of the International Workshop on Parallel and Distributed Algorithms, Cateau de Bonas. France,* 1998.

[16] K. Serebryany y T. Iskhodzhanov, «Thread Sanitizer - Data Race Detection in Practice».

[17] R. De Castro, Teoría de la computación: lenguajes, autómatas, gramáticas, Bogotá: Universidad Nacional de Colombia, Sede Bogotá, 2004.

[18] A. Galton, «The Stanford Encyclopedia Philosophy (Fall 2008 Edition),» Edward N. Zalta, [En línea]. Available: http://plato.stanford.edu/archives/fall2008/entries/logic-temporal/.

[19] JBoss Cache TreeCache, «A Structured, Replicated, Transactional Cache,» *User Documentaion. Release 1.4.1 "Cayenne".*

[20] A. Møller, «dk.brics.automaton – finite-state automata and regular expressions for Java,» *URL http://www.brics.dk/automaton/,* latest visit on may 2011 (2010).

[21] «JGroups Home Page,» latest visit on June 2012. [En línea]. Available: http://www.jgroups.org/.

[22] Oracle, «Java API,» latest visit on June 2012. [En línea]. Available:

http://docs.oracle.com/javase/1.5.0/docs/api/.