Tesis de Licenciatura

MAAL: Logging centralizado para sistemas distribuidos

Alejandro Isacovich
Matías Castellani

Director: Dr. Víctor Adrián Braberman

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

Alejandro Isacovich	LU 681/96	aleisa@dc.uba.ar
Matías Andrés Castellani	LU 142/99	mcastel@dc.uba.ar

Abstract

Los sistemas distribuidos son cada vez más comunes debido al decremento de los costos de hardware, alta velocidad en comunicaciones, etc. En particular, los de tiempo real requieren atención especial debido a que usualmente llevan a cabo tareas críticas que pueden representar pérdidas, ya sea en términos de vidas o grandes sumas de dinero.

Este tipo de sistemas presentan, además, un conjunto de dificultades que los hacen especialmente complicados. Entre estas se encuentran la depuración (*debugging*), el testeo y la verificación. Inclusive, una vez detectado un defecto, es difícil encontrar en cual de los módulos o nodos esta exactamente la falla.

Una de las causas de estas dificultades es el obstáculo de los sistemas distribuidos para obtener una visión global de su funcionamiento. Es decir, si se pudiera obtener una traza global de la ejecución del sistema completo se podría facilitar el testeo de la aplicación.

En esta tesis se presenta una técnica que permite minimizar el impacto y el esfuerzo necesario para poder lograr que aplicaciones distribuidas de tiempo real puedan generar trazas globales de las corridas cuyos eventos respeten el orden causal.

En orden de minimizar el impacto se desarrolló un lenguaje declarativo que permite especificar fuera de la aplicación cuales son los eventos a registrar en las trazas.

Para garantizar el orden causal de los eventos de las trazas generadas se diseño, demostró e implementó un algoritmo que se encarga de mantener ordenados los eventos a medida que estos se suceden.

Con el objetivo de minimizar el esfuerzo para adaptar la aplicación de modo que genere la traza, la técnica incluye un módulo de instrumentación que le incorpora el componente que produce la información para la traza.

Se desarrolló una prueba de concepto de la técnica para aplicaciones implementadas en JAVA y se utilizó en dos casos de estudio. En el primero, se toma como base una simulacion simple del comportamiento de un ferrocarril y los cruces con barera. En el segundo, se utiliza como aplicación distribuida la implementacion de un cliente de BitTorrent llamada Azureus.

Por último, para facilitar su lectura, se diseño e implementó una visualizacion de las trazas en forma gráfica similar a un diagrama de secuencias entre los nodos.

1 Introducción

Los sistemas distribuidos son cada vez más comunes debido al decremento de los costos de hardware, alta velocidad en comunicaciones, etc. En particular, los de tiempo real requieren atención especial debido a que usualmente llevan a cabo tareas críticas que pueden representar pérdidas, ya sea en términos de vidas o grandes sumas de dinero.

Este tipo de sistemas presentan, además, un conjunto de dificultades que los hacen especialmente complicados. Entre estas se encuentran la depuración (*debugging*), el testeo y la verificación. Inclusive, una vez detectado un defecto, es difícil encontrar en cual de los módulos o nodos esta exactamente la falla.

Una de las causas de estas dificultades es el obstáculo de los sistemas distribuidos para obtener una visión global de su funcionamiento. Es decir, si se pudiera obtener la información consolidada y global de la ejecución del sistema completo con características que provean suficiente confiabilidad para poder sacar conclusiones de la ejecución real, se podría facilitar el testeo de la aplicación.

En los sistemas centralizados se puede obtener esta información a través de la generación de una traza o *log* que registre la historia de ejecución de los mismos. Ya que este sistema cuenta con un solo nodo, los eventos registrados en la traza tienen una relación directa con la ejecución y mantienen su orden de ocurrencia, por ende, ningún evento se registra antes que los eventos que lo causan. Esto significa que respetan el orden causal.

En los sistemas distribuidos resulta más complicado generar una traza global porque cada nodo genera su propia traza y consolidarlas de manera que se respete el orden causal no es trivial.

Estas trazas posibilitan realizar diversos análisis de la ejecución. Entre ellos, una técnica interesante para el testeo de aplicaciones consiste en generar la traza y contrastarla con reglas formales de especificación, tal como lo hacen [SBFO02], [JPF88], [PF98] y [MAC99]. Esto puede realizarse tanto *on-line* o una vez finalizada la corrida.

Para que se puedan verificar reglas que involucran a todo el sistema, esta técnica requiere que la traza cumpla, al menos, con las siguientes propiedades:

- Debe estar unificada para todo el sistema. Es decir, debe ser global.
- Debe respetar el orden causal de los eventos que se registraron. Si un evento es causa de otro, la traza debe reflejar esa relación.
- Si se quiere verificar sistemas de tiempo real se debe registrar el momento de ocurrencia (*timestamp*) de los eventos con un error acotado.

Esta tesis tiene como objetivo presentar una técnica para que sistemas distribuidos de tiempo real implementados en Java generen una traza global con las características necesarias para su correcto testeo utilizando los mecanismos de verificación mencionados arriba.

Para esto, se definirá el problema formalmente y se mostrará el estado del arte de la tecnología relacionada. A partir de la definición del problema se presenta una solución con su arquitectura de alto nivel. La solución incluye un algoritmo que resuelve el problema de la unificación de las trazas, un lenguaje de especificación de eventos y una prueba de concepto.

2 Descripción del problema y estado del arte

Hay sistemas distribuidos que carecen de la posibilidad de obtener una traza unificada global en forma directa ya que, en principio, cada nodo produce su propia traza y unirlas todas no es un proceso trivial.
Por otro lado, está el problema de especificar qué información se registrará en la traza. La información completa de la ejecución del sistema es, en general, demasiado abundante y solo una fracción de la misma es realmente útil.
Existen varias herramientas para la generación y unificación de trazas en sistemas distribuidos, no obstante ninguna de estas herramientas cumplen con todos los siguientes requerimientos y características, considerados importantes para que la herramienta sea útil en el campo de la verificación de sistemas con requerimientos de tiempo real.

Requerimientos necesarios	
RPERS:	Abierto para personalizar la traza. Es decir, que permita modificar el contenido de la traza para adaptarse a distintas herramientas.
RCAUS:	Respetar el orden causal de los eventos. [JJJR94]
RTIME:	El error en la diferencia entre los *timestamps* de dos eventos de distintos nodos debe ser acotado en función de los parámetros de la red.
RGLOB:	La traza generada tiene que estar unificada para todo el sistema distribuido.

Características importantes	
REVEN:	Definir un método fácil e intuitivo para la especificación de eventos e información del mismo como por ejemplo, cuando y como se registrarán.
RINV:	Bajo nivel de invasión, tanto en la parte estática (código), como en la parte transmisión de mensajes.
RFLEX:	Flexibilidad para adaptarse a distintos tipos de sistemas de mensajería para sistemas distribuidos (*middleware*) ya que estos son, en general, cerrados y por ende se dificulta su modificación.

Este problema se puede dividir en dos partes. La primera es la de especificación de eventos para la generación de la traza a partir de un sistema centralizado. La segunda es la de generar una traza global, para sistemas distribuidos de tiempo real, a partir de las trazas individuales de cada uno de los nodos. A continuación se detalla la situación actual de cada una de estas.

2.1 Especificación y generación de eventos

El problema de la generación de trazas es ortogonal al dominio de negocio de las aplicaciones. Es por esto que es aconsejable que la información de los eventos que se deben generar y su significado estén separados del resto del sistema.
Existen en la actualidad varios productos que permiten incorporar la generación de eventos a un sistema siguiendo un esquema similar al descrito anteriormente, a saber:

- Definen un lenguaje para especificar eventos de alto nivel por fuera de la aplicación.

- Instrumentan la aplicación de modo que ésta produzca la información correspondiente cuando sucede alguno de los eventos especificados.
- Proveen una herramienta para el procesamiento y monitoreo de los eventos.

Entre estos productos se encuentran *MaC* [MAC99], *Temporal Rover* [ROV85], *Java Pathfinder* [PF98] y *TraceInvader* [CHS97]. Ninguno cumple completamente con los requerimientos REVEN, RINV, RFLEX, RPERS, RGLOB.

Los que se pueden destacar como los más completos son *MaC* y *TraceInvader*. A continuación se describirán sus limitaciones sin tener en cuenta las características favorables de cada una de estas.

MaC provee un ambiente poco amigable para la especificación de eventos. O sea, no cumple el requerimiento REVEN. Utiliza componentes que actualmente están discontinuados, lo cual lo hace poco robusto para ser utilizado en el desarrollo de un nuevo sistema. Es muy poco flexible y no permite personalizar la información de los eventos generados, lo que imposibilita agregarle funcionalidad. Por lo tanto, no satisface los requerimientos RFLEX y RPERS.

Por su parte *TraceInvader* no brinda ningún soporte para especificar eventos de alto nivel. El mecanismo que provee para generar la traza consiste en modificar el sistema de mensajería para que éste registre el envío y recepción de cada mensaje. La única forma en que permite que se registren eventos locales en la traza es modificando el sistema manualmente. Se concluye entonces que esta herramienta no cumple los requerimientos REVEN y RINV (si se desean generar eventos locales).

Tampoco cumple con RFLEX ya que si se desea utilizar un sistema de mensajería distinto al provisto se lo debe modificar a fondo para que envíe la información que requiere esta herramienta.

Al generar la traza modificando las primitivas del sistema de mensajería es muy complicado agregar información extra para poder personalizar la traza (RPERS).

2.2 Generación de la traza global

Existen dos algoritmos conocidos que resuelven la causalidad de los eventos, uno es el introducido por Lamport [LAM78] y el otro por Fidge [FID91]. TraceInvader [CHS97] utiliza el algoritmo de Lamport para la generación de la traza global.

La traza generada usando ambos algoritmos cumple con las condiciones necesarias, pero van un paso más; lo cumplen durante la generación de la misma. Esto lo logran agregando información en cada mensaje entre los nodos de la red distribuida. Por ejemplo [LAM78] envía un número de orden y [FID91] envía un arreglo de números de orden con uno por cada nodo en la red. El problema es que la mayoría de los sistemas de mensajería existentes no contemplan este requerimiento y, por lo tanto, es necesaria su modificación.

A nuestro entender, no existe ningún algoritmo que provea los requisitos planteados anteriormente y no impliquen cambios ni en el subsistema de comunicación de los nodos ni en el sistema distribuido. Tampoco existe un algoritmo que permita mantener la causalidad de los eventos sin enviar información extra en los mensajes.

3 Arquitectura de la solución

En líneas generales, la solución que proponemos se basa en una herramienta que permite generar trazas globales de sistemas que originalmente no tenían esa capacidad. Para esto se atacan dos problemas fundamentales. El de adaptación de la aplicación distribuida a ser testeada, denominada de aquí en adelante DAUT (*Distrubuted Appication Under Test*), para que registre la información de sus eventos y el de mantener la causalidad de los mismos en una traza global.

Esta herramienta esta diseñada para ser utilizada por un "Ingeniero de Test" denomidado de aquí en adelante TE (*Test Engineer*) que tendrá la responsabilidad de implementar la generación de una traza en la DAUT. Para realizar este trabajo necesita tener definidos cuales son los eventos que se requiere que el sistema registre. Por ejemplo, para un sistema de sincronización de semáforos, se necesita registrar un evento para cada semáforo en cada cambio de estado.

MAAL provee un lenguaje declarativo que permite al TE escribir de forma cohesiva toda la información de cómo generar este registro. Es decir, una vez decididos cuales son los eventos del sistema el TE configura en un solo lugar los puntos en los cuales éstos ocurren durante la ejecución. Para el ejemplo del sistema de control de semáforos, simplemente el TE debe detectar el código en el sistema que produce cada cambio de estado.

A esta configuración se la denomina Especificación de Eventos y a partir de ésta comienza el proceso de instrumentación que se definirá en 3.1. En este proceso, en base a la Especificación de Eventos y a la DAUT, se genera un nuevo sistema denominado IDAUT (*Instrumented Distributed Application Under Test*) cuyos nodos informan los eventos que ocurren a un servidor central llamado *TraceServer*. Esta nueva aplicación IDAUT no es más que la DAUT más el modulo de generación de eventos llamado EGM (*Event Generation Module*).

El proceso de instrumentación se basa en el paradigma de Programación Orientada a Aspectos **(AOP)** especializándolo para que a partir de la Especificación de Eventos se generen los aspectos correspondientes que implementan la funcionalidad de registración de eventos.

Como siguiente paso, el TE compila el código generado conjuntamente con el código de la aplicación original, obteniéndose el ejecutable de la aplicación ya instrumentada.

La IDAUT genera pedidos de registro de eventos que son recolectados por el *TraceServer* que tiene como responsabilidad principal mantener la causalidad de los mismos y proveer las herramientas para generar la traza global unificada en el formato que se requiera según el caso. Por ejemplo, generar un archivo de *log* o ser la fuente de eventos para un sistema de tiempo real de verificación de trazas.

Una de las diferencias mas destacadas de esta técnica es el diseño de un nuevo algoritmo que mantiene la causalidad de los eventos sin la necesidad de modificar el *middleware* utilizado por la DAUT.

Además, se modeló la Especificación de Eventos de manera de que sea extensible, flexible y no intrusiva. En la prueba de concepto se utilizó XML como base para el lenguaje de especificación de eventos. Pero se podría generar la Especificación de Eventos si utilizar XML. Por ejemplo con una aplicación visual o de forma programática.

3.1 Especificación, instrumentación y generación de eventos

En esta sección se explicará como MAAL facilita al TE codificar la especificación de eventos por fuera código original de la DAUT y como logra que esta misma genere los eventos a partir de esta especificación.
La DAUT pertenece a un dominio de problema propio que es, en general, distinto al dominio de la generación de eventos, a pesar de que los dos dominios de problemas son autónomos tiene que existir un acoplamiento que se da cuando se pretende que la DAUT genere eventos utilizando esta técnica. El acoplamiento esta dado desde el sistema de generación de eventos hacia el sistema distribuido y no al revés. Es decir, la DAUT no depende del sistema generación de eventos pero si este último depende de la DAUT. Visto de otra forma, tiene sentido la existencia de la DAUT sin la generación de eventos, pero no lo contrario.
La solución que ofrece esta técnica es un proceso sistemático que, en base a una especificación de eventos escrita en el lenguaje MAALEDL, pueda adaptar automáticamente la DAUT agregándole el modulo de generación de eventos (EGM). Al proceso de agregar el EGM a una DAUT se lo denominará **Instrumentación**. En la Fig. 1 se puede visualizar gráficamente este proceso.

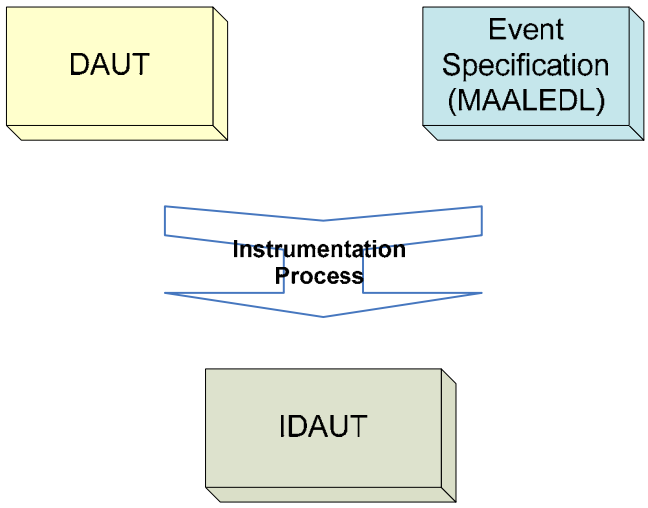

Fig. 1 – Instrumentation

La funcionalidad de la DAUT se mantiene intacta en la IDAUT además. La IDAUT no es más que la DAUT + EGM. Entonces se pueden deducir que la arquitectura original de la DAUT se conserva sin modificaciones en la IDAUT.

3.1.1 Definición e instrumentación de eventos

La definición de un evento se puede ver como la asociación entre la semántica del mismo y el código en donde se produce. Por ejemplo, si que quiere especificar el evento de un semáforo “cambió a amarillo” el TE tiene que especificar en el lenguaje MAALEDL cual es el código que al ejecutarse produce el cambio de luz a amarillo y asociarlo con la semántica “el semáforo x cambió a amarillo”.
El conjunto de todas estas asociaciones forman la especificación de eventos y se escriben en el lenguaje MAALEDL que se detalla mas adelante en la sección 5.
El objetivo es que esta especificación de eventos en conjunto con la DAUT sean la base para que MAAL pueda registrar los eventos en una Traza Global.

Para agregar las funcionalidades a la DAUT de detectar, generar y enviar los pedidos de registro de eventos al *TraceServer* se utiliza la Programación Orientada a Aspectos **(AOP)**. AOP es una manera de extender el paradigma de OOP (Programación orientada a objetos)
Los eventos inicialmente se especifican en el lenguaje MAALEDL y luego, por medio de un proceso que toma como entrada esta especificación se generan los **aspectos** de AOP correspondiente a la generación de eventos para la DAUT. Como último paso de la instrumentación, se incorporan los aspectos generados en la DAUT.
En particular, para la prueba de concepto se utilizó *AspectJ* [ASPECTJ] como lenguaje de programación orientada a aspectos. Esta implementación de AOP esta acoplada con el lenguaje Java. La instrumentación se realiza a través de la compilación de la DAUT con el compilador de *AspectJ*.
El código generado *AspectJ* puede ser modificado, depurado y/ó inspeccionado fácilmente ya que usa la sintaxis de Java. A este se código se lo denominará código auto-generado. En el capitulo 6 **Prueba de Concepto** se puede ver la implementación con más detalle. Este proceso se puede visualizar en la Fig. 2.
El proceso de instrumentación detallado está separado en 2 partes la primera es la generación de código *AspectJ* y la segunda es la compilación del código auto-generado junto con el código de la DAUT para generar la IDAUT. Este mismo proceso se puede unificar para que el código no pueda ser modificado.

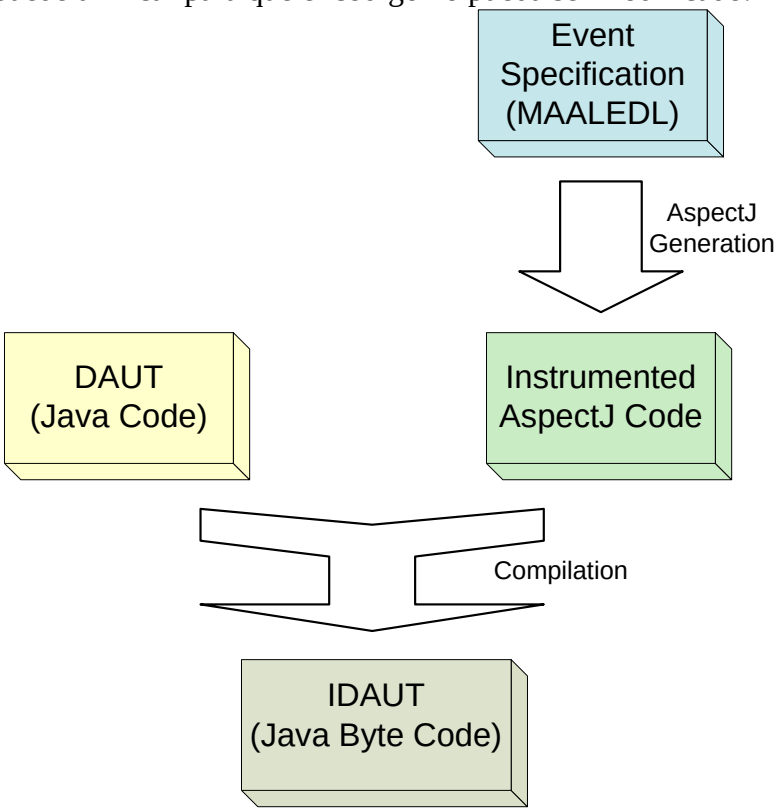

Fig. 2 - Instrumentación de una aplicación en Java

3.1.2 Modulo de generación de Eventos (EGM)

Este módulo es el que se agrega a la DAUT en el proceso de instrumentación. Es el encargado de detectar, generar y enviar los eventos durante la ejecución de la IDAUT. Su funcionamiento lo define la especificación de eventos y parte de su código es generado automáticamente en la instrumentación.
La EGM se conforma con 3 componentes que son detalladas a continuación.

InstrumentedEvents:	Se encarga de detectar en tiempo de ejecución los eventos especificados, recopilar la información particular de cada evento y desencadenar la construcción y el registro del mismo.
EventBuilderHelper:	A pedido de la componente *InstrumentedEvents*, se encarga de crear los objetos que representan a un evento en base a la información particular que éste le suministra. Es decir, su responsabilidad es realizar toda la lógica genérica que requiere construir un evento.
EventRaiser:	Envia los pedidos de registro de eventos al ***TraceServer***.

La componente ***InstrumentedEvents*** se genera dinámicamente a partir de la Especificación de Eventos en el proceso de instrumentación ya que es completamente dependiente de la DAUT y de los eventos que se quiren generar. Por el contrario ***EventRaiser*** y ***EventBuilderHelper*** son componentes estándar que son agregados directamente. Se puede visualizar la arquitectura de la IDAUT en la Fig. 3.

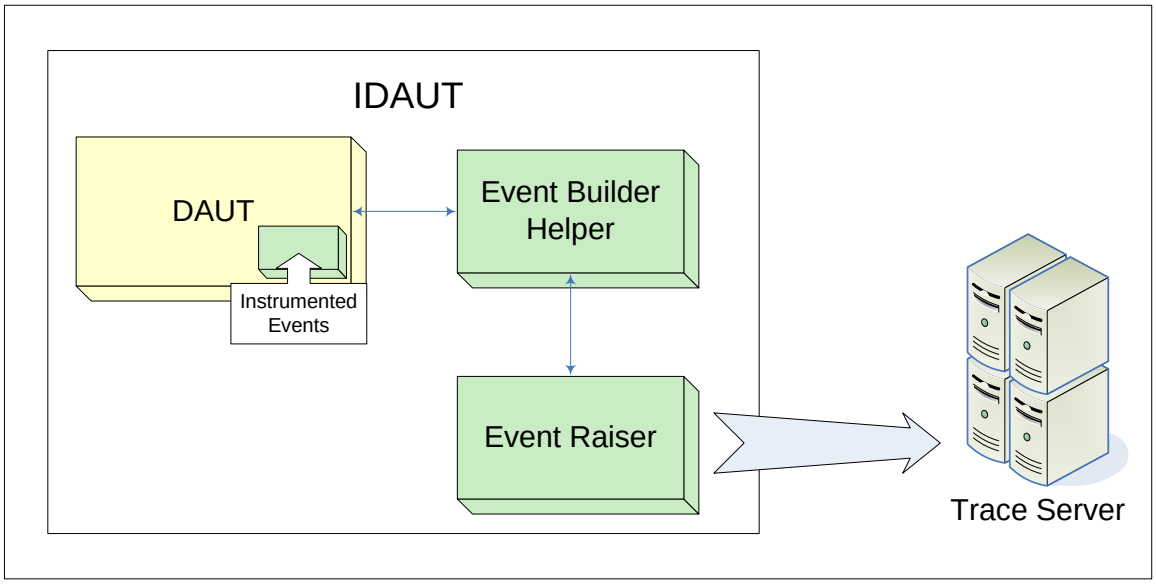

Fig. 3 - Arquitectura de la IDAUT

3.2 Arquitectura de Alto Nivel (en tiempo de ejecución)

En esta sección se explicará como una aplicación que ya fue instrumentada IDAUT registra los pedidos de eventos en el *TraceServer* para luego generar una traza global respetando el orden causal de los eventos.
En los sistemas distribuidos los distintos nodos interactúan por medio de algún *midleware*. La IDAUT no se ve modificada en este aspecto luego de la

instrumentación. La diferencia es que ahora, además, cada nodo genera pedidos de registros de eventos y se los envía al *TraceServer* por medio de un canal confiable y que asegura que se mantiene el orden entre los pedidos de registro. Es decir, si un nodo *A* envía un pedido de registro de un evento e_1 y luego de un evento e_2, el *TraceServer* recibe primero e_1 y luego e_2. El *TraceServer* recibe los pedidos de registro, los procesa y registra en el *System Trace* asegurando por medio del *Causality Algorithm* que los eventos permanezcan ordenados con respecto al orden causal. Este algoritmo se detalla en el capitulo 4.
El *System Trace* es la traza global que cumple con todos los requerimientos propuestos en la descripción del problema.
La Fig. 4 muestra como se relacionan los procesos de la IDAUT con el Trace Server.

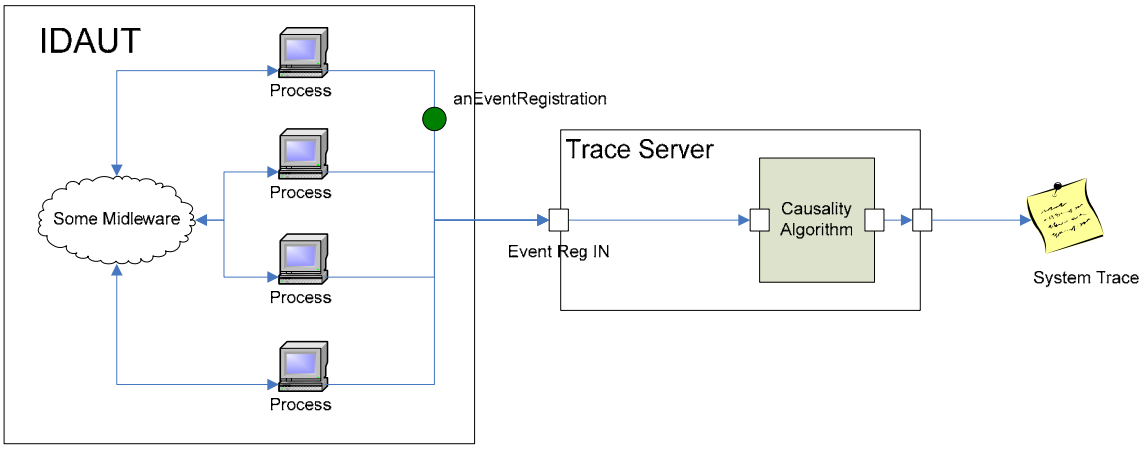

Fig. 4 - Arq. Alto Nivel

3.3 Time stamping y acotamiento de error

El requerimiento RTIME que debe cumplir la traza generada indica que el error en la diferencia entre los *timestamps* de dos eventos de distintos nodos debe ser acotado en función de los parámetros de la red.
Para este propósito se utilizó el protocolo NTP [NTP] que sincroniza los relojes de los nodos.
Éste es un protocolo que funciona sobre redes de datos de latencia variable y puede mantener un error de hasta 10 milisegundos (1/100 seg) sobre Internet y puede alcanzar una precisión de 200 microsegundos (1/5000 seg) o mejor en redes locales bajo condiciones ideales [NTPWP] [MILLS91].
NTP permite mantener un error mínimo acotado en base a las condiciones de los nodos y de la red sobre la que están montados. De esta forma, el *timestamp* de cada evento será asignado localmente por en nodo origen del mismo.
Sin embargo, como se verá en los casos de estudio, el error acotado no es suficiente para garantizar la coherencia de los timestamps de todos los eventos de la traza.

4 Algoritmo para la construcción de trazas globales

En esta sección se presentará un algoritmo que soluciona el problema de orden causal, se detallarán las precondiciones necesarias para su correcto funcionamiento y la demostración de correctitud del mismo.

4.1 Marco Formal

En este trabajo se sigue la notación y formalismo presentado por [Lam78] en cuanto al ordenamiento de eventos en un sistema distribuido. A continuación se hará un breve resumen de elementos básicos que se utilizaran en el resto del informe.

Se define como sistema distribuido a un conjunto de procesos interconectados. Cada proceso consiste de una secuencia de eventos. Se utilizará el término "nodo" o "proceso" indistintamente a lo largo de esta sección. Se asume que los eventos de un proceso forman una secuencia en la cual el evento *a* está antes que el *b* si *a* ocurrió antes que *b*. Es decir, todos los eventos de un proceso forman un orden total a priori. Nótese que, a partir de esta definición, un evento **está antes** que otro en la secuencia de un nodo y se utilizará esta expresión para expresar que **ocurrió** antes.

Se asume también que enviar o recibir un mensaje es un evento en un proceso. A partir de esto se puede definir la relación "ocurrió antes" (notada por "$\rightarrow$") como la relación más pequeña entre eventos de un sistema distribuido que satisface las tres siguientes condiciones:

Si *a* y *b* son eventos del mismo proceso y *a* está antes que *b* en la secuencia entonces $a \rightarrow b$.

Si *s* es el envío de un mensaje desde un proceso y *r* es la recepción del mismo mensaje por otro proceso entonces $s \rightarrow r$.

Si $a \rightarrow b$ y $b \rightarrow c$ entonces $a \rightarrow c$.

Se dice que dos eventos son concurrentes si $a \not\rightarrow b$ y $b \not\rightarrow a$.

Se asume que $a \not\rightarrow a$ para todo evento *a*. Esto implica que $\rightarrow$ es un orden parcial no reflexivo sobre todos los eventos del sistema distribuido.

Cabe aclarar que un nodo puede enviarse un mensaje a si mismo, y si los eventos *s* y *r* representan el envió y la recepción de ese mensaje respectivamente entonces, por 1, $s \rightarrow r$.

Se define traza [JJJR94] como el conjunto de registros de los eventos de un sistema distribuido más un orden parcial entre estos registros. Se utiliza la notación *"R(a)"* para referirse al registro del evento *a* y "$\prec^+$" a la relación de orden entre los registros. Si *R* es el conjunto de registros de eventos, se define $T = (R, \prec^+)$ como la traza *T* compuesta por los registros de eventos *R* y relacionados por $\prec^+$. Se usa el símbolo $\prec^+$ en vez de $\prec$ porque la relación de orden de una traza es naturalmente transitiva y no tiene sentido una relación de orden que no esté clausurada. Además, de esta forma se puede utilizar $\prec$ para definir una relación mas relajada que es útil en muchos casos.

Usando estas definiciones se puede re-escribir el requerimiento de causalidad de manera formal: Se requiere que el sistema de generación de trazas cumpla que si *a* y *b* son dos eventos cualquiera, entonces se cumple que

Si $a \rightarrow b$ entonces $R(a) \prec^+ R(b)$.

4.2 El problema de la causalidad

Analizando el problema desde una perspectiva simplificada se puede comenzar con la obtención de la traza de un sistema centralizado, monoproceso y monotarea.

Si el nodo registra los eventos en el orden que ocurren y tomamos la relación $\prec^+$ como el orden en que se encuentran los registros en la traza, entonces es verdad que si $a \rightarrow b$ entonces $R(a) \prec^+ R(b)$.

Ahora, si se cuenta con un sistema distribuido en donde cada proceso produce su propia traza como en el ejemplo anterior, no se implica que se origine una traza global del sistema completo.

Para mostrar esto se analizará un caso particular. Se tiene un sistema distribuido con dos procesos que generan sus propias trazas de acuerdo al esquema del ejemplo anterior. Es decir, registran los eventos de forma local en el orden que ocurren. Supóngase que durante la ejecución del sistema se producen los eventos de la Fig. 5.

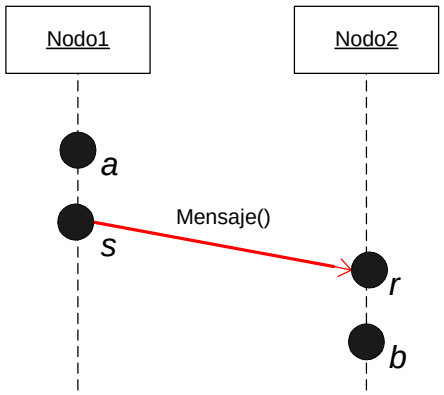

Fig. 5

Como resultado se obtienen dos trazas separadas. Una con los eventos *a* y *s* más la relación $R(a) \prec R(s)$ y otra con los eventos *r* y *b* más la relación $R(r) \prec R(b)$.

Uniendo directamente la información de las trazas generadas por ambos procesos se obtiene una traza $T = (R, \prec^+)$ donde *R* contiene los cuatro eventos *a,s,r* y *b* y $\prec^+$ se conforma por los pares $R(a) \prec^+ R(s)$ y $R(r) \prec^+ R(b)$ ya que esta es la única información que se registró en cuanto a la relación. Sin embargo, por la condición 2) de la definición de la relación $\rightarrow$, si *s* es el envío de un mensaje desde un proceso y *r* es la recepción del mismo mensaje por otro proceso entonces $s \rightarrow r$. Pero $R(s) \not\prec^+ R(r)$ en la traza generada por lo cual existen $a \rightarrow b$ tales que $R(a) \not\prec^+ R(b)$.

Se concluye que esta traza no cumple con el requerimiento de causalidad.

4.3 Aproximación a la solución

En esta sección se construye un algoritmo que resuelve el problema de causalidad con ciertas restricciones. Este algoritmo supone que existen *n* procesos y estos procesos están conectados formando un sistema distribuido.

Además existe un proceso *traceServer* al cual todos los procesos están conectados.

Se define un conjunto de precondiciones mínimas que requiere el algoritmo para su funcionamiento.

Precondición 1 - No se pierden mensajes

La red sobre la cual funcionan estos procesos no pierde mensajes. Es decir, si un mensaje que se supone enviado desde un proceso a otro no llega es porque o bien falló el proceso que envía el mensaje antes de enviarlo o bien falló el receptor antes de recibirlo. ↜

Precondición 2 - No se desordenan los mensajes

Los mensajes llegan en el orden que fueron enviados. Si un proceso envía dos mensajes a otro, el receptor los recibe en el mismo orden que fueron enviados. ↜

Estas condiciones valen para la red entre los procesos del sistema distribuido y para la conexión entre los procesos y el *traceServer*.

El algoritmo se separa en 2 tipos de participantes, los *procesos* del sistema distribuido siendo analizado y el *traceServer*. Los primeros envían información de sus eventos al *traceServer* de forma ordenada. Cada evento es enviado con su correspondiente *timestamp*.

En un primer análisis, la incorporación del *traceServer* puede resolver el problema planteado en el ejemplo anterior asumiendo que éste puede descubrir las relaciones de causalidad entre los eventos de envío y recepción de mensajes. O sea, si se cuenta con una técnica que identifique cuando un pedido de registro de *receive* está relacionado con su anterior pedido de *send* **que ya fue registrado**, se puede actualizar la relación de causalidad para que incluya esta dependencia. Más abajo se detalla como funcionaría esta técnica.

La Fig. 6 muestra la ejecución del mismo ejemplo anterior pero utilizando un nodo que recibe y registra los eventos en la traza.

En este caso, cada nodo no registra su traza, sino que le delega el registro de evento al *traceServer*. Se mantienen las precondiciones de que cada nodo envía el mensaje de registro en el mismo orden que suceden los mensajes y de que la red no desordena estos mensajes.

Asimismo, el proceso *traceServer* se encarga de recibir los pedidos de registro y ejecutarlos en el orden de llegada. Además se ocupa de detectar que los eventos *s* y *r* están relacionados y actualizar la relación $\prec^+$ consecuentemente.

La traza resultante, entonces, estaría definida por la secuencia *{R(a), R(s), R(r), R(b)}* en la cual *R(i)* esta antes que *R(j)* en la secuencia si $R(i) \prec^+ R(j)$.

A modo de ejemplo, se demuestra que para todo *a, b* eventos de la ejecución que muestra la Fig. 6 (específicamente de esa ejecución), si $a \rightarrow b$ entonces $R(a) \prec^+ R(b)$. O sea que para este ejemplo específico se cumple el requerimiento de causalidad.

Demostración del Ejemplo: Por la condición 1) de la definición de la relación $\rightarrow$, $a \rightarrow s$ y $r \rightarrow b$.

Por la condición 2) $s \rightarrow r$. Y por la condición 3, $a \rightarrow r$, $a \rightarrow b$ y $s \rightarrow b$. Entonces, los eventos de la ejecución forman la secuencia *{a, s, r, b}* donde un evento *i* esta antes de un evento *j* si $i \rightarrow j$. Por otro lado, la relación $\prec^+$ forma la secuencia de registros *{R(a), R(s), R(r), R(b)}*, donde para cada evento *i*, *R(i)* esta en la misma ubicación que *i* en la secuencia de eventos. En consecuencia, para cualquier evento *i,j* tal que $i \rightarrow j$ en la ejecución de la Fig. 6, $R(i) \prec^+ R(j)$. ↜

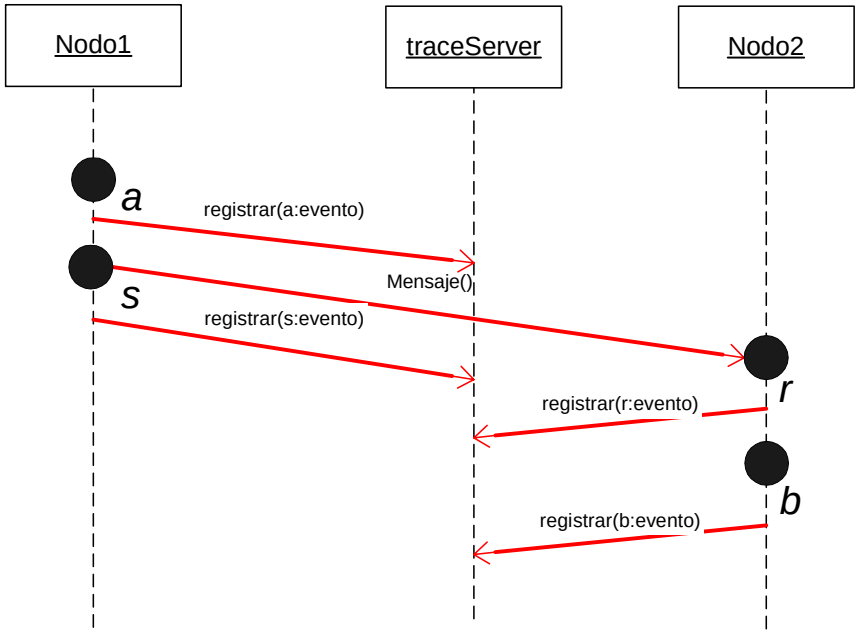

Fig. 6

Se demostró que la traza generada para el último caso cumple con el requerimiento de causalidad. Sin embargo en ese ejemplo se muestra una ejecución en donde la mensajería cumple con una característica no siempre disponible: el mensaje *registrar(s:evento)* llega antes al *traceServer* que el mensaje *registrar(r:evento)*, es decir, llega antes el *send* que el *receive*. Esto no tiene porque siempre ser así: perfectamente puede darse el caso inverso y, en tal, se registraría un evento de recepción de mensaje antes que su respectivo envío, lo cual violaría el requisito de causalidad (si bien $s \rightarrow r$, r estaría registrado en la traza antes que s y por ende $R(s) \not\prec^{+} R(r)$). Es decir, un algoritmo ingenuo como este no soporta que los pedidos de registro de envío y recepción lleguen desordenados.
Ésta restricción es la que diferencia a esta aproximación de la solución final. En el siguiente capitulo se mostrará una técnica para garantizar que los pedidos de registro de envío de mensaje se procesen antes que los de recepción de mensaje.

Otro punto que quedó sin resolver es cómo el nodo *traceServer*, ante la llegada del mensaje *registrar(r:evento)*, identifica que *r* corresponde a la recepción de *s*. Para esto se define el siguiente **modelo de mensajes de eventos** entre el *traceServer* y el resto de los nodos.
Existen tres tipos de eventos: *local*, *send* y *receive*. Un evento *local* representa un evento cualquiera en algún nodo del sistema distribuido, un evento *send* representa el acto de envío de un mensaje desde un nodo a otro y un evento *receive* representa la recepción en un nodo de un mensaje enviado por otro nodo. Se puede ver que los eventos de tipo *send* y *receive* son, además, eventos de tipo *local*.
Cada nodo envía al *traceServer* el pedido de registro de un evento *local* indicando el nodo donde se produjo.

Cada nodo envía al *traceServer* el pedido de registro de un evento *send* indicando los nodos de origen y destino de mensaje enviado.
Cada nodo envía al *traceServer* el pedido de registro de un evento *receive* indicando los nodos de origen y destino del mensaje recibido. Los efectos colaterales de estas condiciones se analizan en **4.5.1 Modelo de mensajes de eventos**.
En base a este modelo se define formalmente en lenguaje de especificación [Object-Z] *LocalEvent*, *SendEvent*, *ReceiveEvent*. Pero antes se da un breve resumen de la notación utilizada para simplificar la lectura.
Las clases se definen a través de un cuadro con el siguiente formato.

NombreDeClase
[ClasesHeredadas] *[DefinicionesLocales]* *[EsquemaDeEstadoIinicial]* *[Operaciones]* *[Invariante]*

Las *DefinicionesLocales* son una lista de variables de forma $|identificador:Tipo$. El *EsquemaDeEstadoIinicial* define el estado de las variables locales al instanciar un objeto de la case. Su sintaxis es

INIT
Predicados

Las operaciones definen el comportamiento de una clase y el acceso a sus variables locales. Su sintaxis es

NombreDeOperacion
Declaraciones
Axiomas

En las declaraciones se definen los parámetros de entrada y de salida de la operación. Los primeros se distinguen con un signo de pregunta al final de su nombre y los últimos con un símbolo de admiración.
El invariante es típicamente un predicado que restringe el comportamiento de los objetos de esa clase.

Ahora sí se puede pasar a la definición de eventos. Un *LocalEvent* es simplemente una dupla que contiene un nombre y un nodo. Formalmente,

LocalEvent
| *owner* : *Node* | *name* : *Name*

Owner
owner! : *Node*
owner! = *owner*

Name
name! : *Name*
name! = *name*

Un *SendEvent* es básicamente un *LocalEvent* más un nodo de destino. El método *match* establece una relación muy simple entre eventos *send* y *receive* que indica que el nodo que envía el mensaje es el mismo para ambos eventos y lo mismo sucede con el nodo que recibe el mensaje. Es decir, este par de eventos **podrían** pertenecer al mismo mensaje.

SendEvent
LocalEvent | *destination* : *Node*

Destination
destination! : *Node*
destination! : *destination*

Match
aReceiveEvent ? : *ReceiveEvent* *match*! : *Bool*
match! = *owner* = *aReceiveEvent.source* ∧ *destination* = *aReceiveEvent.owner*

Un *ReceiveEvent* no es más que un local event con el agregado de un nodo origen del mensaje.

ReceiveEvent
LocalEvent | *source* : *Node*

source
source! : *Node*
source! : *source*

Basándose en este modelo de eventos se puede armar una técnica para que el *traceServer* pueda determinar qué evento *send* corresponde a qué evento *receive*. Para esto se definen las clases *Message* y *SystemTrace*. Un *Message* sólo es una dupla de un *SendEvent* y un *ReceiveEvent*.

Message
$\mid sendEvent : SendEvent$
$\mid receiveEvent : ReceiveEvent$

sendEvent
$sendEvent! : SendEvent$
$sendEvent! = SendEvent$

receiveEvent
$receiveEvent! : ReceiveEvent$
$receiveEvent! = receiveEvent$

$sendEvent.match(receiveEvent)$

Un *SystemTrace* modela en leguaje Z la idea de que una traza es, como se definió al principio, una dupla que contiene un conjunto de eventos y una relación de orden. Una diferencia importante es que esta traza establece una relación de orden $\prec$ que debe ser clausurada para cumplir las propiedades de la relación $\prec^+$ de una traza.
Inicialmente la traza está vacía. El método *addLocal* agrega eventos de tipo *LocalEvent, ReceiveEvent y SendEvent*. Notar que también actualiza la relación $\prec$ para que el evento agregado sea posterior al último evento del mismo nodo.
El método *addMessage* actualiza la relación de orden de forma que dos eventos de envío y recepción ya registrados queden relacionados causalmente como parte de un mismo mensaje entre dos nodos. Los métodos *sendMatching* y *happenedBefore* están explicados en su definición.

SystemTrace
$\mid events : \mathrm{P} LocalEvent$ $\mid \_ \prec \_ : LocalEvent \leftrightarrow LocalEvent$
$(\forall e_1, e_2)(e_1 \prec e_2 \Rightarrow e_1 \in events \wedge e_2 \in events)$

INIT
$events = \{\}; \prec = \{\}$

addLocalEvent
$\triangle(events)$ $\triangle(causality)$ $aLocalEvent? : LocalEvent$
$events' = events \cup \{aLocalEvent?\}$ $\prec' = \prec \cup \{(e, aLocalEvent?) \mid e \in events \wedge$ $e.owner = aLocalEvent?.owner \wedge$ $(\neg \exists z)(e \prec z \wedge e.owner = z.owner)\}$

addMessage

$\triangle(causality)$ $aMessage?:Message$
$\prec' = \prec \cup \{(aMessage.sendEvent, aMessage.receiveEvent)\}$

sendMatching
$aReceiveEvent?:ReceiveEvent$ $aSendEvent!:SendEvent$
/* Retorna el primer evento que *matchea* con el evento de recepción y que todavía no está asociado con ningún evento de recepción */ $aSendEvent! = \min(s \mid s.match(aReceiveEvent) \wedge$ $(\nexists r)(s \prec r \wedge s.owner \neq r.owner), \prec^+)$

happenedBefore
$aEvent1?:LocalEvent$ $aEvent2?:LocalEvent$ $happenedBefore!:Bool$
/* Permite obtener información del orden de los eventos respecto a la relación $\prec$ clausurada */ $happenedBefore! = aEvent1? \prec^+ aEvent2?$

Como se dijo, este *modelo de mensajes de eventos* es el que utiliza el *traceServer* del ejemplo de la Fig. 6 para determinar la relación causal entre eventos de distintos nodos. Su utilización es esencial en el algoritmo que se muestra a contunuacion y que es el que el *traceServer* ejecuta en cada llegada de pedido de registro de evento. Se asume que antes de la llegada del primer evento el *SystemTrace* st está vacío.

Entrada: a:LocalEvent.

1.- st.addLocalEvent(a)
2.- Si a es ReceiveEvent
    st.addMessage(Message[st.sendMatching(a), a])

En el paso 1 este algoritmo actualiza la relación $\prec$ para que refleje que el ultimo evento reportado *a* ocurrió después que el último evento registrado (llámeselo *b*) para el nodo *a.owner*.
En el paso 2, en caso que el evento *a* que se está procesando sea un *ReceiveEvent*, se busca el evento de envío ya registrado que corresponda con *a*. Este evento, como no se desordenan los mensajes, es necesariamente el *SendEvent* más antiguo que no se haya relacionado con ningún otro *ReceiveEvent*.
Así, el traceServer identifica que el evento *r* es consecuencia del evento *s* del ejemplo de la Fig. 6.
Cabe aclarar que no se registra que *a* ocurrió después que todos los mensajes registrados para el mismo nodo. Esta información puede obtenerse clausurando transitivamente $\prec$ puesto que para todo *c* ocurrido en *a.owner* si $R(c) \prec R(b)$ y

$R(b) \prec R(a)$ entonces debería ser válido que $R(c) \prec R(a)$ para que la relación tenga sentido. Por esta razón se define $\prec^+$ como la clausura transitiva de la relación $\prec$.

Retomando el problema de la restricción que tiene esta aproximación, se puede demostrar[1] que el algoritmo de arriba genera correctamente una traza bajo las condiciones pedidas más una agregada:

> Precondición 3 – Los eventos *send* llegan antes que sus *receive*
>
> Para todos los mensajes entre los nodos, el pedido de registro del evento *send* llega antes que el pedido de registro del evento *receive*. ↜

Esta precondición solo se utiliza como paso intermedio. El algoritmo final no la requiere.

4.4 Algoritmo de ordenamiento

Como se vio en la sección anterior los algoritmos presentados no cumplen con el requerimiento de causalidad sin agregar la Precondición 3 – Los eventos *send* llegan antes que sus *receive*. Dicho esto, si se establece un mecanismo por el cual el *traceServer* ordene los mensajes en caso que un pedido de registro de un evento *receive* llegue antes que el de su respectivo *send* se podría construir una traza que cumpla con el requerimiento de causalidad sin la esa restricción. Es decir, eliminar la Precondición 3. Para esto se define el algoritmo - Causality Sort que utilizará el *traceServer* para procesar los pedidos de registro de eventos. Éste reemplaza el utilizado por el *traceServer* para el ejemplo anterior pero utiliza el mismo **modelo de mensajes de eventos**.
El algoritmo tiene como entrada una secuencia "bien formada" de pedidos de registro de eventos. Esta secuencia se puede construir con todos los pedidos de registro que llegan al *traceServer* en el mismo orden en que llegaron. A continuación se definen formalmente las propiedades de una secuencia bien formada.

> **Definición**: Un conjunto *P* de pedidos de registro de eventos es "bien formado" si y solo si para todo pedido de registro de un evento de *send* perteneciente a *P* existe un pedido de registro de evento de *receive* perteneciente a *P* que corresponde al mismo mensaje, y para todo pedido de registro de un evento de *receive* perteneciente a *P* existe un pedido de registro de evento de *send* perteneciente a *P P* que corresponde al mismo mensaje.

En otras palabras un conjunto "bien formado" de pedidos de registro de eventos no es más que un conjunto en el que cada uno de los pedidos de *send* tiene su *receive* correspondiente y viceversa.

> **Definición**: Una secuencia S es "bien formada" si y solo si el conjunto subyacente de la secuencia es bien formado y todos los eventos de un mismo nodo están en la misma secuencia en que ocurrieron originalmente.

Si se construye una secuencia solo y con todos los eventos que llegan a un traceServer durante una ejecución completa de una UAT se obtiene una secuencia "bien formada" ya que las precondiciones de la red necesarias garantizan tanto el orden como la existencia de los eventos. Es decir, si se envían de a uno en orden los pedidos de registro que están en una secuencia "bien formada" al *traceServer* se obtiene el mismo

[1] No se demuestra ya que la idea solo se incluye a modo explicativo. Además, como las precondiciones del algoritmo son muy fuertes no tiene relevancia en este trabajo.

resultado que haciendo la corrida directamente iterando sobre la misma ya que una secuencia bien formada no es más que el orden en el que llegaron los pedidos de registro al *traceServer*.
Es por esto que el Algoritmo 1 - Causality Sort formalmente recibe como entrada una secuencia “bien formada” y registra los eventos en la traza respetando el orden causal.
La idea principal detrás de éste es detectar los pedidos de registro de eventos que llegan desordenados y retenerlos hasta que lleguen todos los pedidos de registro que deberían haber llegado antes para poder registrar todos los pedidos respetando el orden causal.

Algoritmo 1 - Causality Sort

```
     Entrada: Una secuencia S bien formada.
1.   st:SystemTrace
2.   pending:PendingEvents
3.   st.INIT
4.   pending.INIT
5.   while  S Is Not Empty
6.         e ← S.head
7.         o ← e.owner
8.         S ← S.tail
9.         If pending_o Is Not Empty, pending_o.add(e), break;
10.
11.        If e is LocalEvent
12.              st.addLocalEvent(e)
13.
14.        If e is ReceiveEvent
15.              If exists st.sendMatching(e)
16.                    st.addLocalEvent(e)
17.                    st.addMessage(Message[st. sendMatching(a), a])
18.              Else
19.                    pending_o.add(e)
20.
21.        If e is SendEvent
22.              d ← e.destination
23.              st.addLocalEvent(e)
24.              If pending_d Is Not Empty
25.                    If e.match(pending_d.top)
26.                          S ← pending_d + S
27.  end while
28.  return st
```

4.4.1 Demostración

La demostración se va a realizar sobre un conjunto de entrada finito y "bien formado". Si se toma el orden en el que llegan todos los eventos del conjunto al *traceServer* queda definido un orden total que puede ser representado por una secuencia, siendo esta una secuencia "bien formada".

El Algoritmo 1 - Causality Sort se compone únicamente de un ciclo y por ende se utilizara la técnica de la propiedad invariante para demostrarlo. Los estados iniciales y finales del algoritmo se corresponden con los del ciclo.

Notar que en esta demostración se hace un abuso de notación en cuanto a los eventos reales y los registros de eventos. Para un evento *a*, a su registro de evento *R(a)* se lo denomina también *a* donde esto no genere confusión.

4.4.1.1 Estado Inicial

El estado inicial y precondición del algoritmo esta definido por el siguiente conjunto.

$\{S_0 = S \wedge wellFormed(S)\}$

Donde *wellFormed(S)* indica que S es una secuencia "bien formada".

4.4.1.2 Guarda del ciclo

La guarda del ciclo indica que la secuencia no sea vacía. Es decir, $\{\#S > 0\}$.

4.4.1.3 Invariante

Para simplificar la lectura se define el conjunto *pendingEvents* como

$$pendingEvents = \bigcup_{n \in Nodes} pending_n.$$

El invariante tiene ocho predicados. El primero (*i.*) dice que si un evento registrado en la traza *ocurrió antes* que otro, la traza refleja esa propiedad y viceversa.
El segundo (*ii.*) indica que todos los eventos que hay son los de S_0 y que ningún evento puede estar en *st.events* o en *pendingEvents* o en *S* al mismo tiempo. Es decir, los conjuntos *st.events*, *pendingEvents* y *S* forman una partición de S_0
El siguiente predicado (*iii.*) afirma que si un evento esta en *pendingEvents* es porque uno de los eventos que *ocurrió antes* está en S. Es decir, todavía no fue procesado.
El cuarto (*iv.*) que en cada conjunto *pending* los eventos están ordenados siguiendo el orden de ocurrencia. Se introduce la relación $<_{pending_n}$ donde $a <_{pending_n} b$ si *a* está antes que *b* en la cola $pending_n$. El quinto (*v.*) dice lo mismo que el anterior pero sobre los eventos que están en *S*. El sexto (*vi.*) dice que para los eventos de un mismo nodo *n*, los que estan en *S* ocurrieron antes que lo que están en $pending_n$.
El séptimo (*vi.*) dice que no puede haber eventos en *st* que son menores causalmente a eventos que están en *S*. Y por último el octavo (*viii.*) dice lo mismo que en el (*vii.*) pero para los eventos que están en *pendingEvents*.

$$\begin{aligned}
&\{i. - (\forall a,b)(a,b \in st.events \wedge a \prec_{st}^{+} b \Leftrightarrow a \rightarrow b) \wedge \\
&ii. - partition(\{st.events, pendingEvents, S\}, S_0) \wedge \\
&iii. - (\forall b)(b \in pendingEvents \Rightarrow (\exists a)(a \rightarrow b \wedge a \in S)) \wedge \\
&iv. - (\forall a,b,n)(a,b \in pending_n \wedge (a <_{pending_n} b \Rightarrow a \rightarrow b)) \wedge \\
&v. - (\forall a,b)(a,b \in S \wedge a.owner = b.owner \wedge a \rightarrow b \Rightarrow a <_S b) \wedge \\
&vi. - (\forall a,b)(a.owner = b.owner \wedge a \in S \wedge b \in pendingEvents \Rightarrow a \rightarrow b) \wedge \\
&vii. - (\forall a,b)(a \in S \wedge b \in st \Rightarrow a \not\rightarrow b) \wedge \\
&viii. - (\forall a,b)(a \in pendingEvents \wedge b \in st \Rightarrow a \not\rightarrow b)\}
\end{aligned}$$

4.4.1.4 Estado Final

La poscondición del algoritmo define formalmente en términos de estados del algoritmo el requerimiento de causalidad que se especifica al principio del capitulo. El primer término requiere que para cualquier par de eventos registrados en la traza, su relación de causalidad refleje su orden de ocurrencia. El segundo, sólo asienta que todos los eventos de la traza sean los que ocurrieron y ninguno más.

$$\begin{aligned}
&\{(\forall a,b)(a,b \in st.events \wedge a \prec_{st}^{+} b \Leftrightarrow a \rightarrow b) \wedge \\
&(\forall e)(e \in st.events \Leftrightarrow e \in S_0)\}
\end{aligned}$$

4.4.1.5 Invariante y negación de la guarda implican estado final

Si se conjuntan el invariante y la guarda del ciclo negada, se obtiene el siguiente conjunto de estados

$$
\begin{aligned}
&\{i.-(\forall a,b)(a,b \in st.events \wedge a \prec_{st}^{+} b \Leftrightarrow a \rightarrow b) \wedge \\
&ii.- partition(\{st.events, pendingEvents, S\}, S_0) \wedge \\
&iii.-(\forall b)(b \in pendingEvents \Rightarrow (\exists a)(a \rightarrow b \wedge a \in S)) \wedge \\
&iv.-(\forall a,b,n)(a,b \in pending_n \wedge (a <_{pending_n} b \Rightarrow a \rightarrow b)) \wedge \\
&v.-(\forall a,b)(a,b \in S \wedge a.owner = b.owner \wedge a \rightarrow b \Rightarrow a <_S b) \wedge \\
&vi.-(\forall a,b)(a.owner = b.owner \wedge a \in S \wedge b \in pendingEvents \Rightarrow a \rightarrow b) \wedge \\
&vii.-(\forall a,b)(a \in S \wedge b \in st \Rightarrow a \not\rightarrow b) \wedge \\
&viii.-(\forall a,b)(a \in pendingEvents \wedge b \in st \Rightarrow a \not\rightarrow b) \wedge \\
&ix.-\neg(\#S > 0)\}
\end{aligned}
$$

Por la condición *iii* y la negación de la guarda se deduce fácilmente que *pendingEvents* es un conjunto vacío. Esto, sumado a que *S* es un conjunto vacío y *ii*, implica $partition(\{st.events, \varnothing, \varnothing\}, S_0)$ que es lo mismo que decir que

$(\forall e)(e \in st.events \Leftrightarrow e \in S_0)$.

Entonces con *i* y el predicado anterior se llega al estado final.

$$
\begin{aligned}
&\{(\forall a,b)(a,b \in st.events \wedge a \prec_{st}^{+} b \Leftrightarrow a \rightarrow b) \wedge \\
&(\forall e)(e \in st.events \Leftrightarrow e \in S_0)\}
\end{aligned}
$$

4.4.1.6 El invariante se cumple

Antes de comenzar el ciclo, el *SystemTrace st* esta vacío, $S = S_0$ y como cada una de la listas de pendientes esta vacía, entonces *pendingEvents* también está vacío. En este estado, es trivial ver que todos los predicados del invariante son ciertos. El invariante vale al inicio del ciclo.

A continuación se demostrará que al final de cada paso del ciclo el invariante se cumple. Para esto se analiza el su validez al terminar una iteración para cada combinación posible de tipo de evento que se procesa y estado de las estructuras. Cada estado corresponde a un caso de flujo del cuerpo del ciclo, definido por las guardas de los condicionales del algoritmo:

If $pending_o$ **Is Not Empty** (Caso 1, línea 9)
If *e* **is** *LocalEvent* (Caso 2, línea 11)
If *e* **is** *ReceiveEvent* y **exists** *st.sendMatching(e)* (Caso 3, subcaso 1, línea 15)
If *e* **is** *ReceiveEvent* y no **exists** *st.sendMatching(e)* (Caso 3, subcaso 2, línea 18)
If *e* **is** *SendEvent* (Caso 4, línea 21)

En cada uno de los casos, se analizan los ocho terminos del invariante.

Caso 1:
$pending_o$ no está vacío.

i.- Como en este caso no se actualiza *st*, sigue valiendo.
ii.- Vale porque se saca un elemento de *S* y se lo pone en *PendingEvents*.
iii.-Sigue valiendo ya que solo se agregó *e* a $pending_o$, existe un evento *a* que traba a $pending_o$ porque $a \rightarrow pendings_o.top$, *a* está en *S* y $pendings_o.top \rightarrow e$ entonces, por transitividad, $a \rightarrow e$.

iv.- Vale porque, por *vi*, *e* es efecto de todos los evento de $pending_o$ y como *e* se agrega al final de $pending_o$, $(\forall b)(b \in pending_o \wedge b <_{pending_o} e)$.
v.- Vale porque sólo se está eliminando un elemento de *S*.
vi.- Por *v*, ningún evento del nodo *o* perteneciente a *S* es causa de *e*. Entonces, si agrego *e* a $pending_o$, sigue valiendo *vi*.
vii.- Vale porque no se modificó *st* y se eliminó un elemento de *S*.
viii.- Vale porque se mueve *e* desde *S* a *PendingEvents* y vale *vii*.

Caso 2:
e es un *LocalEvent* y $pending_o$ está vacío.

Cuando se trata de un evento local se ejecuta el método *addLocalEvent* de *SystemTrace* en el que se agrega el evento al conjunto de eventos de *st* y se actualiza la relación de causalidad solo se si existe un evento del mismo nodo ya registrado.

i.- Por *vii* y por *v* sigue valiendo.
ii.- Vale porque e pasa de *S* a *st*.
iii.- Todos los eventos de pendientes son efecto del tope, que es un evento de recepción, que a su vez es efecto de un evento de envío, como el evento que se acaba de registrar no es un evento de envío sigue valiendo.
iv.- Vale porque no se modificó *pendingEvents*.
v.- Sigue valiendo porque sólo se eliminó un elemento de *S*.
vi.- Como no se modifica *pendingEvents* y de S sólo se saca un elemento vale.
vii.- En el paso anterior *vii* valía y se saca *e* el primer elemento de *S* y como vale *v* y *e* es un evento local ningún elemento de *S* es causa de *e*. Entonces al agregarlo a *st* sigue valiendo *vii*.
viii.-
Si se supone que no se cumple, entonces $(\exists a, b)(a \in pendingEvents \wedge b \in st \wedge a \rightarrow b)$.
Aparte se sabe que en el paso anterior valía *viii*. En este paso sólo se está agregando un evento local a *st*, los únicos eventos que son causa directa de eventos locales son eventos del mismo nodo, lo que es un absurdo porque $pending_o$ esta vacío y para que haya causalidad indirecta tiene que ser sí o sí por medio de un evento de recepción, entonces tiene que existir un evento de recepción *r* tal que *r.owner=e.owner* y un evento *c* perteneciente a *pendingEvents* tal que *r* es causa de *c* con lo que se cae en otro absurdo ya que *r* estaba desde el paso anterior en *st* y *c* estaba en *pendingEvents* por lo tanto no valía *viii* en el paso anterior.

Caso 3:
e es un *ReceiveEvent* y $pending_o$ está vacío.

Para el caso de los eventos de recepción existen 2 subcasos. El primero sucede cuando ya fue registrado el evento de envio correspondiente a este evento de recepción y el segundo cuando todavía no fue registrado este evento. En este último caso es muy simple de ver que el invariante sigue valiendo ya que es similar al Caso 1. Salvo por los pedicados iii, iv que se demuestran a continuación.
iii.- Sigue valiendo ya que solo se agregó *e* a $pending_o$, que estaba vacío, existe un evento *a* y un evento *s* tal que *a* traba $pending_{s.owner}$ porque $a \rightarrow pendings_{s.owner}.top$, *a* está en *S*, $s \in pendings_{s.owner}$ y *s* se corresponde con *e* y por transitividad $s \rightarrow e$. O existe *s* tal que $s \in S$ y *s* se corresponde a *e* lo que implica que $s \rightarrow e$.

iv.- Vale porque $pendings_o$ queda con un solo elemento.

Ahora se demostrará que si el evento de envío correspondiente a este evento de recepción pertenece a *st* el invariante se sigue cumpliendo.
En este caso primero se ejecuta el método *addLocalEvent*, al igual que el caso 2, que actualiza la relación de causalidad con respecto a los eventos sucedidos en el mismo nodo y luego se ejecuta el método *addMessage* el cual se encarga de actualizar la relación de causalidad que existe entre el mensaje de envio y recepción que forman este mensaje. En este caso los predicados *ii*, *iii*, *iv*, *v* y *vi* siguen valiendo porque son iguales al Caso 2. A continuación de demostrarán los predicados *i* y *vii*.

i.- La relación de causalidad para los eventos locales se sigue cumpliendo al igual que en los casos anteriores. Como el evento *s* de envío correspondiente con *e* pertenece a *st*, se está actualizando la relación de causalidad y como *i* valía en el paso anterior $(\forall a)(a \in st.events \wedge a \prec_{st}^{+} s \Leftrightarrow a \rightarrow s)$.

Como se acaba de actualizar la relación de causalidad de modo que $s \prec_{st} e$, se puede ver que se cumple que $(\forall a)(a \in st.events \wedge a \prec_{st}^{+} e \Leftrightarrow a \rightarrow e)$. Por ende se cumple *i*.

vii.- Se cumple porque al igual que en los casos anteriores los eventos del mismo nodo que son efecto de *e* ya se encuentran registrados en *st* por la ejecución de *addLocalEvent*. Y los únicos eventos causas no locales de *e* son el evento de envío *s* que se corresponde con *e* y todas sus causas y como *s* pertenece a *st* y valía *vii* en el paso anterior todas sus causas pertenecen a *st* y ninguna en *S* registradas.
viii.- Con la misma idea que en el Caso 2 para la misma propiedad se puede ver que pasa lo mismo con el evento causa directo y con el indirecto provocado por otro evento de recepción del mismo nodo. Solo que acá existe un caso más que es que los derivados directos de *e*, que es un evento de recepción, sean las causas del evento de envío *s* correspondiente a *e*. Lo cual es un absurdo porque *s* ya estaba registrado de antes.

Caso 4:
e es un *SendEvent*

i.- Como se ejecuta el método addLocalEvent la relación de causalidad se sigue cumpliendo al igual que en el Caso 2.
ii.- Vale porque e pasa de *S* a *st*.
iii,*iv*,*vi*,*viii*.- Si *e* no destraba $pending_d$ este predicado vale porque no se modifica *pendingEvents* caso contrario también se cumplen ya que se vacía completamente $pending_d$.
v.- Si *e* no destraba $pending_d$ este predicado vale ya que solo se eliminó *e* de *S*. Caso contrario por *iv* los evento que se agregan a S están ordenados causalmente y porque *vi* valía en el paso anterior cuando se concatena $pending_d$ con *S* sigue valiendo *v*.
vi.- Como no se modifica *pendingEvents* y de S sólo se saca un elemento vale.
vii.- Si *e* no destraba $pending_d$ este predicado vale porque no se modifica *S*. En caso contrario, por *viii*, los eventos de $pending_d$ no son causa de ningún evento de *st*, por ende, si se concatena $pending_d$ a S, *vii* sigue valiendo.

4.4.1.7 El algoritmo termina

En esta sección se va a demostrar que si se cumplen las precondiciones el algoritmo termina.

Se pueden distinguir 4 casos en el que se hacen tratamientos distintos a los pedidos de registro. Estos casos son los mismos que se trataron en la demostración de la validez del invariante. A continuación se va a demostrar que en cada uno o bien aumenta el tamaño de la traza o aumentará en una cantidad finita y acotada de pasos hasta llegar al tamaño de S_0.

Caso 1:
$pending_o$ no está vacío.

Se elimina el pedido de registro de S para almacenarlo temporalmente en $pending_o$. El pedido de registro va a quedar trabado hasta que llegue el pedido de registro del evento de envío *s* correspondiente al pedido de recepción *r* que está trabando la cola. Como el pedido *r* fue almacenado en el Caso 3, subcaso 2, se puede decir que en una cantidad finita de pasos va a llegar s y se van a destrabar todos los elementos de la cola para ser procesados nuevamente.

Caso 2:
e es un *LocalEvent* y $pending_o$ está vacío.

Se registra este pedido en la traza y se elimina el pedido de *S*. Por lo tanto *st* crece en 1.

Caso 3:
e es un *ReceiveEvent* y $pending_o$ está vacío.

Al igual que para la demostración de la validez del invariante, para este caso existen 2 subcasos. El primero sucede cuando ya fue registrado el evento de envio correspondiente a este evento de recepción y el segundo cuando todavía no fue registrado este evento. En el primero directamente se registra este pedido en la traza y se elimina el pedido de *S*. Por lo tanto *st* crece en 1.
El segundo subcaso es un poco más complejo y se analiza a continuación. Si $pending_o$ está vacío, *e* es un pedido de registro de un evento de recepción y el evento de envío que se corresponde con *e* todavía no fue registrado entonces se elimina el pedido de registro *e* de *S* y se lo almacena temporalmente en $pending_o$ hasta que llegue el pedido de registro del evento de envío correspondiente a *e*.
En este caso, no se registra nada en *st*. Pero, en una cantidad finita de pasos se va a destrabar esta cola de pendientes y se registrará este evento en la traza porque S_0 es una secuencia finita “bien formada”.
Demostración:
Supóngase que no se destraba en ningún momento esta cola de pendientes y ya se consumió toda la secuencia *S*. Esto solo puede ser si el pedido de registro de envío correspondiente a este *e* esta trabado en otra cola. Y a su vez, esta cola esta trabada. Y así sucesivamente con todas las colas restantes. Pero alguna tiene que estar trabada porque tiene su pedido de registro de envío en una de las colas anteriores y entraron en *deadlock*. En la Fig. 7 se puede visualizar gráficamente este problema. El pedido *e* puede ser cualquiera de los pedidos de registro de recepción r_1, r_i, r o r_j, es decir, o pertenece a un ciclo o llega a este ciclo de alguna manera.

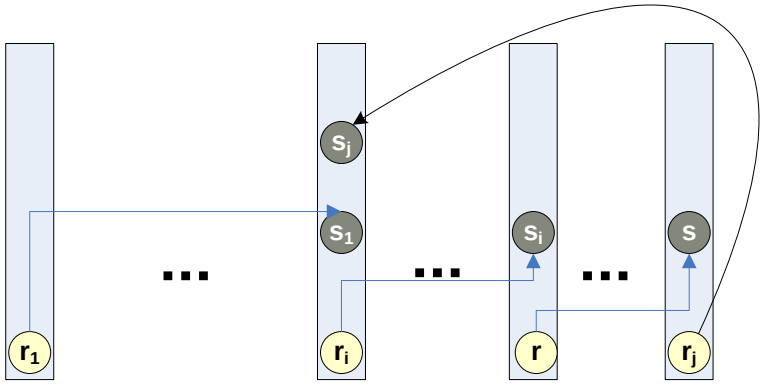

Fig. 7 - DeadLock

Este caso es un absurdo que se puede ver fácilmente en un ciclo de solo 2 colas como el caso de la Fig. 8 y se puede extender a un ciclo de n. Esto significa que r_j esta antes que s_i en *S* y que r_i esta antes que s_j y como son del mismo nodo significa que $r_i \rightarrow s_j$ y que $r_j \rightarrow s_i$ pero a su vez por la definición de $\rightarrow$ también se cumple que $s_i \rightarrow r_i$ y $s_j \rightarrow r_j$ lo cual es un absurdo por la Precondición 2 - No se desordenan los mensajes que hereda S_0 por se una secuencia “bien formada”.

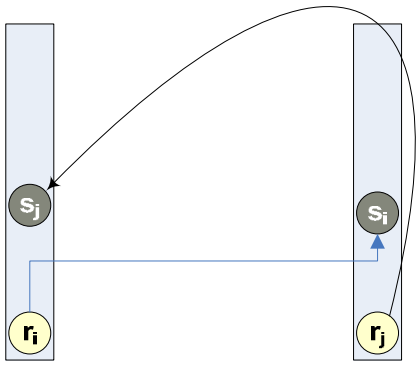

Fig. 8 - Ciclo de 2

Entonces queda demostrado que no se puede trabar una cola de pendientes de forma que se genere un ciclo.
También podría ser que nunca se termine de procesar *S*. Lo cual también es un absurdo porque en todos los pasos o bien se elimina directamente un elemento o se incrementa *S* pero solo con los pedidos de registro que estaban anteriormente en *S* y que hasta el momento no fueron registrados y además en este paso siempre se registran 2 eventos en *st* que no vuelven más a *S* (Caso 4).

Caso 4:
e es un *SendEvent*

Si $pending_o$ está vacío y *e* es un pedido de registro de un evento de envío entonces se registra *e* en la traza y se elimina este pedido de *S*. Por lo tanto crece *st* en 1.
Si $pending_d$ no esta vacío entonces se agregan a *S* todos los elementos de $pending_d$. Es decir, se destraban todos los pedidos de registro que estaban trabados en la cola para que se los vuelva a procesar. En este paso *S* aumenta de tamaño pero en su cabeza queda el evento de recepción que trababa la cola $pending_d$ el cual ya tiene registrado su evento de envío (este evento es el *e* que se acaba de registrar). Así, en el próximo paso de la iteración se está en el caso 3 subcaso 1.

Siendo que la traza aumenta de tamaño en cada paso o se mantiene estable para volver a aumentar de tamaño en una cantidad finita y acotada de pasos y el tamaño de la traza no puede crecer más que el tamaño inicial de *S* queda demostrado que el algoritmo termina.

4.5 Implicaciones

En este capitulo se analizan las consecuencias de las precondiciones que necesita el algoritmo propuesto para asegurar el requerimiento de causalidad en la traza generada.

4.5.1 Modelo de mensajes de eventos

Una de las restricciones más fuertes de este modelo para la generación de trazas en sistemas distribuidos es la que define el **modelo de mensajes de eventos**. Repasando: cada nodo envía al *traceServer* el pedido de registro de un evento *receive* indicando los nodos de origen y destino del mensaje recibido.
Se puede ver que si bien es razonable que un nodo que envía un mensaje sepa cual es su origen (el mismo nodo) y cual es su destino (sino, no podría enviarlo), es común que en muchos sistemas el nodo que recibe el mensaje no sepa, a priori, el origen del mismo.
Un ejemplo de un caso con estas características es el de los sistemas distribuidos a través de Servicios Web. En éstos, los nodos que brindan servicios no saben de donde vienen los pedidos. Sin embargo, la mayoría de las implementaciones permiten obtener esa información de alguna manera. En el los capítulos 7, 8 se muestran técnicas para resolver este problema en dos casos distintos. Uno utiliza RMI como *middelware* y el otro está directamente montado sobre TCP/IP.

4.5.2 Ordenamiento de mensajes y confiabilidad

Según la
Precondición 1 - No se pierden mensajes y la *Precondición 2 - No se desordenan los mensajes*, son requisitos que la red sobre la cual los procesos del sistema intercambian mensajes no pierda paquetes y que los mensajes lleguen en el orden que fueron enviados.
Si bien muchos sistemas distribuidos no necesitan estas condiciones para su correcto funcionamiento, muchas herramientas de *middleware* las cumplen, siempre y cuando se las use adecuadamente.
Muchos sistemas de manejo y administración de colas como IBM MQSeries, MSMQ, ActiveMQ, JBossMQ y JORAM cumplen con estas condiciones y posibilitan que procesos distribuidos utilicen el modelo productor consumidor sobre colas. La garantía fundamental de las estas colas es que los mensajes nunca se pierden y dos

mensajes insertados al final de la cola en cierto orden serán consumidos en el mismo orden si se extraen desde el principio de la misma.
Otro ejemplo que cubre muchos casos es el protocolo TCP. Este garantiza que, en condiciones normales, los paquetes son entregados y que no se desordenan [CNTAN96], [CNPD99]. Si los procesos que escuchan los mensajes a través de TCP envían los pedidos de registro de los eventos *receive* antes de escuchar el siguiente mensaje entonces, para corridas normales, el sistema cumple las precondiciones necesarias.

4.6 Ventajas y desventajas

En esta sección se verán las distintas ventajas y desventajas de este algoritmo comparado con algoritmos similares conocidos.
Leslie Lamport presenta en [Lam78] un algoritmo que permite ordenar los eventos de un sistema distribuido de forma tal que cumplan la "Condición del Reloj". Esto significa que el algoritmo produce una función *C* que asigna un número a cada evento (si E es el conjunto de eventos, se define $C : E \to \mathrm{N}$) y que si $a \to b$ entonces $C(a) < C(b)$.
Se había definido que una traza es un conjunto de registros de eventos más una relación que los ordene. Si se registran todos los eventos de *E* y se llama *R(a)* a todo evento $a \in E$ registrado, entonces se puede definir la relación $\prec$ de la siguiente forma
Para todo evento $a \in E$, $R(a) \prec^{+} R(b)$ si y solo si $C(a) < C(b)$.
A partir de esto se puede construir una traza $T=(R, \prec^{+})$ a partir del algoritmo de Lamport. Se analiza ahora la característica de causalidad de esta traza.
Como para todo par de eventos *a* y *b* de *E*, $a \to b$ implica que $C(a) < C(b)$, y existe *R(a)* y *R(b)* entonces $R(a) \prec^{+} R(b)$. Entonces, se cumple que si $a \to b$ vale $R(a) \prec^{+} R(b)$ y por ende la traza T cumple con el requerimiento de causalidad.

Sin embargo, diseñar una técnica de instrumentación para lograr que algún tipo genérico de sistema distribuido utilice el algoritmo de Lamport para la generación de trazas es un problema más complejo. Ya que en todos los casos hay que modificar cada uno de los *middlewares* que se utilice para que se envíe la información adicional requerida por el algoritmo para poder ordenar causalmente los eventos.
Las desventajas que tiene el algoritmo presentado en este trabajo con respecto al de Leslie Lamport están relacionadas con las precondiciones definidas en las secciones **4 Algoritmo para la construcción de trazas globales** y **4.3 Aproximación a la solución** y. Por la forma en que el algoritmo de Lamport maneja el orden de los eventos **no** requiere ni que la red no pierda mensajes (Precondición 1) ni que los mensajes de un mismo nodo lleguen ordenados (Precondición 2).
Además, como cada mensaje esta identificado unívocamente, no es necesario conocer para cada mensaje origen y destino como lo indican los puntos 2, 3 y 4 del 4.5.1 Modelo de mensajes de eventos.
Otro algoritmo con el que se puede comparar es el presentado por Colin Fidge en [FID91]. Este algoritmo está basado fuertemente en el introducido por Lamport con la particularidad de que garantiza que dados dos eventos *a* y *b*, no solo vale que $a \to b$ implica que $C(a) < C(b)$ sino que agrega la doble implicación. Es decir que $a \to b$ si y solo si $C(a) < C(b)$. Como la técnica es, a grandes rasgos, muy similar a la de Lamport, sus ventajas y desventajas son las mismas.

5 Lenguaje de especificación de eventos MAALEDL

El lenguaje de especificación de eventos MAALEDL (MAAL Event Definition Language) es una de las formas que ofrece este trabajo para instrumentar una DAUT implementada en *JAVA*. Es decir, es una manera de definir cómo y que eventos de un sistema conformarán la traza. Para esto permite al TI definir los eventos que la DAUT informará al *TraceServer* para que sean registrados adecuadamente en la traza global. Este lenguaje ofrece las siguientes características:

Permite definir eventos locales y mensajes, de los descritos en el algoritmo, entre nodos por fuera del código de la DAUT permitiendo que no se contamine el código con código no inherente al dominio de la aplicación y disminuyendo notablemente el riesgo de incorporar *bugs* en el proceso de instrumentación.
Permite definir eventos a nivel de métodos de clases o de instancias. Por ejemplo, se puede especificar un evento que informe si un "tren rápido" se está acercando a una barrera y que este se genere solamente si se trata de un tren de este tipo, suponiendo que el tipo de tren es una propiedad de la instancia del mismo.
Permite especificar a los eventos semántica, nombre y paquete con la misma semántica que *JAVA*.
Permite configurar para cada evento el momento de ejecución del evento (*raisePoint*), si este será lanzado en el momento en que se llama al método (*onInvoke*) o cuando termina de ejecutarse (*onReturn*). Es significativo aclarar la importancia poder configurar en momento de ejecución del evento ya que el orden causal de los eventos de la traza pueden verse modificados por el uso del mismo. Supóngase dos eventos *a* y *b*, que ocurren si son invocados los métodos m_a y m_b respectivamente. Supóngase también que el método m_a realiza una llamada al método m_b. Si en ambos eventos se define su *raisePoint* como *onInvoke*, una llamada a m_a registrá en la traza primero al evento *a* y luego al evento *b*. Pero si el *raisePoint* del evento *a* esta definido como *onReturn*, no importa como se defina *b*, una llamada a m_a registrá en la traza los eventos en el orden inverso ya que el método m_a no retornará hasta que no haya terminado m_b.

El proceso para definir los eventos se compone de 2 partes una es la especificación de cada uno de los eventos y mensajes en el lenguaje MAALEDL y la implementación de ciertas interfaces en JAVA que sirven de soporte al EGM y que se enumeran y se detallan conjuntamente con la sintaxis y semántica del lenguaje. A las clases implementadas en la segunda parte serán denominadas "clases soporte de MAALEDL".

Para la implementación del lenguaje se utilizó como soporte XML [W3C] (Extensible Markup Language) ya que es un lenguaje estándar y tiene algunas propiedades, que serán descritas posteriormente, que lo hacen apropiado para la definición de eventos. Una de ellas es la posibilidad de extensión que provee permitiendo la evolución de la definición de eventos. Además, XML es un lenguaje descriptivo preparado para la lectura y escritura tanto por personas como por herramientas automáticas. Por otro lado, está diseñado para representar cualquier tipo de datos jerárquicos, en particular eventos. Al ser estándar, existen numerosas herramientas de edición y chequeos sintácticos. Asimismo, existe diversidad de programas generadores de parsers específicos que facilitan su uso.

A continuación se mostrará una descripción de la semántica de cada uno de los elementos del lenguaje y por último define formalmente mediante su DTD (Documents Type Definition) [W3C] la sintaxis del lenguaje.

5.1 Elementos del lenguaje

Para exponer la sintaxis del lenguaje se utilizará la notación de Definiciones Regulares [ASU90]. Los términos que están en **negrita** son símbolos terminales.
Ya que a través del lenguaje se define la traza del sistema, una definición comienza con la etiqueta <trace>. La estructura general de una especificación tiene la forma

trace ← **<trace>** (localEvent | messageEvent)* **</trace>**

Como ya se vio, los eventos pueden dividirse entre 'locales' y 'mensajes'. Se pueden definir tantos de cada uno como se necesite. Los primeros son eventos que ocurren en un nodo en particular y se definen mediante la estructura

localEvent ← **<localEvent** eventName packageName? **>**
eventBody
</localevent>

eventBody ← raisePoint? ownerRetreiver methodPattern predicate? semantics?

La definición de eventos 'locales' requiere que se le asigne un nombre como clasificador de eventos. Es decir, los eventos generados con el mismo nombre pertenecen a un mismo grupo a nivel semántico. El *packageName* tiene que ver con la generación del código de los aspectos. Indica el nombre del paquete Java donde se creará el aspecto correspondiente a la instrumentación del evento local. La sintaxis de estas expresiones es

eventName ← **eventName="**eventNameString"
packageName ← **packageName="**packageString". Valor por omisión: **ar.uba.dc.maal.event.aspect**

eventNameString ← Nombre del evento. Puede ser cualquier *JavaIdentifier*. Por ejemplo **OpenGate**.
packageNameString ← Nombre un paquete Java válido. Por ejemplo **ar.dc.uba.mypackage**

La etiqueta opcional <*raisePoint*> permite definir el momento de ocurrencia del evento respecto de la ejecución del método que lo dispara, usando *onInvoke* el registro del evento se realiza en el instante previo a la ejecución del método. En cambio *onReturn* (el valor por omisión) indica que el registro del evento se realiza al final del a ejecución del método. La sintaxis de la etiqueta tiene la forma

raisePoint ← **<raisePoint>**(onInvoke | onReturn)**</raisePoint>**

Una definición de evento local tiene también un *ownerRetreiver***,** que es la componente (en caso de JAVA, la clase) responsable de determinar en que nodo se está produciendo el evento. Se define siguiendo la estructura

ownerRetreiver ← **<ownerRetreiver** nodeRetreiver **/>**
nodeRetreiver ← **className="**nodeRetreiverClassName"
nodeRetreiverClassName ← Nombre de una clase que implemente ar.uba.dc.maal.net.NodeRetreiver. Por ejemplo **ar.dc.uba.mypackage.MyNodeRetreiverClass**. Tiene que estar todo su nombre completo incluyendo el paquete.

El *ownerRetreiver* permite al algoritmo que mantiene la causalidad obtener el nodo de procedencia de los pedidos de registro de eventos. Es decir, representa el nodo *owner(e)* definido en el **modelo de mensajes de eventos**.

Para crear un *ownerRetreiver* debe implementarse la interfaz *ar.uba.dc.maal.net.NodeRetreiver* la cual se usa para todos los objetos que deben determinar un nodo. Esta interfaz define solo el método:

```
public Node getNode(JointPoint jointpoint);
```

Este mismo es llamado cada vez que debe construirse un objeto evento. El parámetro *jointpoint* contiene la información que determinó la ocurrencia del evento, a saber, el objeto que recibió el mensaje que disparó el evento, los argumentos que se utilizaron en la ejecución del método y el nombre del mismo entre otros. Esta información es en algunos casos fundamental para determinar el nodo donde ocurrió el evento.
Volviendo al lenguaje de especificación, lo que sigue son las estructuras que permiten especificar el punto en la DAUT en el cual ocurre el evento. Esto se hace a través de dos elementos, el primero es el *AspectJMethodPattern*, que es una descripción sintáctica de un conjunto de métodos. El segundo es el *predicate* que permite especificar un predicado para cada definición de evento que decidirá, en caso de la llamada a algún método que cumpla con el *AspectJMethodPattern*, si el evento se produce o no.

methodPattern ← <method pattern="AspectJMethodPattern" />
aspectJMethodPattern = call y execute (ver method pattern para call y execute en [ASPECTJ])
predicate ← <predicate className="predicateClassName" /> Valor por omisión: Un predicado que da siempre true.
predicateClassName ← Nombre de una clase que implemente ar.uba.dc.maal.event.Predicate. Por ejemplo ar.dc.uba.mypackage.MyPredicateClass. Tiene que estar el nombre completo de la clase incluyendo el nombre de paquete.

El único método que define la interfaz *ar.uba.dc.maal.event.Predicate* también recibe como parámetro un *JointPoint* para valuar el predicado determinando la ocurrencia o no del evento.
Por ultimo, es posible asignar a cada evento un objeto que represente el significado semántico del mismo. Como esto depende del estado de la IDAUT al momento de ocurrencia, se define la interfaz *ar.uba.dc.maal.event.semantics.SemanticsRetreiver* cuya implementación debe devolver el objeto que representa la semántica del evento a partir del su *JointPoint*. La sintaxis en el lenguaje MAALEDL es

semantics ← <semanticsRetreiver className="semanticsRetreiverClassName" />
semanticsRetreiverClassName ← Nombre de una clase que implemente ar.uba.dc.maal.event.semantics.SemanticsRetreiver. Por ejemplo ar.dc.uba.mypackage.MySemanticsRetreiverClass.

Y así finalizan los elementos que componen la definición de un evento local. Por otro lado, los eventos 'mensaje' indican la ocurrencia de un mensaje entre dos nodos. Están compuestos por un evento de recepción y un evento de envío, que a su vez también son eventos locales.[2] Para especificar la generación de un evento de este tipo se utiliza la etiqueta <message></message> y su sintaxis es la siguiente.

messageEvent ← <message>
sendEvent
receiveEvent
</message>

sendEvent ← <sendEvent eventName packageName>

[2] Es muy importante que la implementación del lenguaje permita validar que los mensajes estén bien definidos, es decir, que para cada mensaje exista un y solo un Evento de Recepción y un y solo un Evento de Envío. Ya que el algoritmo se basa fuertemente en que la red no pierde mensajes, pero si se define, por ejemplo, solo el evento de recepción de un mensaje el que está perdiendo mensajes es el lenguaje mismo.

```
                    eventBody
                    <destinationRetreiver nodeRetreiver />
                </sendEvent>

receiveEvent ← <receiveEvent eventName packageName >
                    eventBody
                    <sourceRetreiver nodeRetreiver />
                </receiveEvent>
```

Los eventos de recepción y los de envío tienen datos particulares pero como también son eventos locales tienen toda la información mencionada arriba.
El atributo particular de la definición de los eventos de recepción es el *sourceRetreiver***,** que es la componente responsable de determinar cual es el nodo que envió el mensaje en tiempo de ejecución (*source(e)*). Así mismo, la definición de los eventos de envío contiene un *destinationRetreiver*, que es la componente responsable de identificar el nodo al cual se enviará el mensaje (*destination(e)*). Tanto el *sourceRetreiver* como el *destinationRetreiver* son clases que implementan *ar.uba.dc.maal.net.NodeRetreiver*. Los nodos que retornan estos objetos son los que utiliza el algoritmo de causalidad para determinar la correspondencia entre los eventos de envío y de recepción y poder construir un evento 'mensaje'.

En la Tabla 1 se resume el significado de cada uno de los elementos que se utilizan en el lenguaje.

Trace	Es el elemento principal. Engloba a todos los eventos y mensajes que se definen y solo tiene que haber uno solo de estos.
localEvent	Declara un evento local en los nodos del sistema, está compuesto de un *methodPattern*, un *ownerRetreiver* y opcionalmente de un *semanticsRetreiver*, un *predicate* y/o un *raisePoint*.
message	Declara el mensaje entre 2 nodos independientes del sistema. Los mensajes están compuestos de un evento de *send* y otro de *receive*. La definición del mensaje asegura que exista uno y solo un Evento de Envío y uno y solo un Evento de Recepción para cada mensaje.
raisePoint	Configura el momento de ocurrencia del evento respecto de la ejecución del método que lo dispara, los valores posibles son 'onInvoke' y 'onReturn'. onInvoke: El registro del evento se realiza en el instante previo a la ejecución del método. onReturn: El registro del evento se realiza al final del a ejecución del método.
methodPattern	Especifica el método que va a generar el evento. Para marcar el método hay que especificar un *pattern* ***call*** o ***execute*** expresado con sintaxis [AspectJ].
predicate	Especifica el predicado que decide en tiempo de corrida si el evento se va a generar o no en base al contexto de ejecución. En *className* se debe especificar el nombre de la clase que implementa la Interfaz *Predicate*. Es opcional y por defecto el evento siempre se genera.
ownerRetreiver	Especifica la clase que tiene la responsabilidad de retornar el nodo en el cual ocurre el evento. Debe implementar la interfaz *NodeRetreiver*
semanticsRetreiver	Especifica la clase que tiene la responsabilidad de define la

	semántica del evento. La semántica es la que le da significado conceptual al evento, por ejemplo, un evento puede ser “se movió el tren” y la semántica diría se movió el tren *t* desde la posición *a* hasta la posición *b* en sentido *Sur* y esto es lo que lo diferencia con otro evento “se movió el tren”. Para definir el *semanticsRetreiver* se debe implementar la interfaz *SemanticRetreiver*.
sendEvent	Define un evento de *send*, este evento se ejecuta como parte de un mensaje entre dos nodos independientes e indica el momento en el que se comenzó con el envío del mensaje. Está compuesto de un *methodPattern*, un *ownerRetreiver*, un *destinationRetreiver* y opcionalmente de un *semanticsRetreiver*, un *predicate* y/o un *raisePoint*.
destinationRetreiver	Especifica la clase que tiene la responsabilidad de decir a donde se dirige un evento de *send*. Debe implementar la interfaz *NodeRetreiver*.
receiveEvent	Define un evento de *receive*, se ejecuta como parte de un mensaje entre dos nodos independientes indicando el momento en el que llegó un mensaje de otro nodo del sistema. Un Evento Receive esta compuesto de un *methodPattern*, un *ownerRetreiver*, un *sourceRetreiver* y opcionalmente de un *semanticsRetreiver*, un *predicate* y/o un *raisePoint*.
sourceRetreiver	Especifica la clase que tiene la responsabilidad de decir de donde viene un evento de *receive*. Debe implementar la interfaz *NodeRetreiver*.
package	Indica donde se va a generar el código para un cierto evento.

Tabla 1 – Semántica del lenguaje MAALEDL

A continuación a modo de ejemplo se muestra la especificación para un evento con el lenguaje MAALEDL.

```
<localEvent name="OpenGate" package="ar.uba.dc.maal.event">
  <ownerRetreiver className="ar.uba.dc.maal.train.node.CrossRailNodeRetreiver"/>
  <semanticsRetreiver
className="ar.uba.dc.maal.train.semantics.OpenGateSemanticsRetreiver"/>
  <method pattern="call(public void ar.uba.dc.maal.train.CrossRail.openGate())"/>
  <predicate className="ar.uba.dc.maal.train.predicate.MyPredicate" />
</localEvent>
```

Código 1 - Ejemplo de *OpenGate*

En la siguiente tabla se explica cada parte de esta definición:

Nombre	**Ejemplo**	**Explicación**
Name	OpenGate	Nombre que identifica al evento en el sistema.
Package	ar.uba.dc.maal.event	Campo opcional que indica el paquete en donde se quiere poner el código auto-generado para este evento en particular.
ownerRetreiver→className	ar.uba.dc.maal.train.node.CrossRailNodeRetreiver	Clase que retorna el nodo en el cual se está generando el evento. Es una de las clases que se tiene que implementar en la aplicación como apoyo al framework.
semanticsRetreiver→className	ar.uba.dc.maal.train.semantics.OpenGateSemanticsRetreiver	Clase que define la semántica del evento. Al igual que la anterior tiene que estar implementada en la aplicación.
method→pattern	call(public void ar.uba.dc.maal.train.CrossRail.openGate())	Define con la sintaxis de AspectJ los puntos de corte (***pointcut***) en donde se va a generar el evento. En este caso dice que el evento de va a producir cuando algún objeto haga una llamada al método *openGate* de la clase *CrossRail*
Predicate→className	ar.uba.dc.maal.train.predicate.MyPredicate	Campo opcional que se utiliza para definir si se genera un evento o no. También se puede implementar un predicado en el que se cuenta con todos los parámetros de la llamada.

5.2 Sintaxis XML

En el Código 2 se presenta la sintaxis del lenguaje mediante su DTD [W3C] (*Document Type Definition*).

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Raise Point -->
<!ELEMENT raisePoint (#PCDATA)>
<!--Detination retreiver-->
<!ELEMENT destinationRetreiver EMPTY>
<!ATTLIST destinationRetreiver className NMTOKEN #REQUIRED>
<!--Event Local-->
<!ELEMENT localEvent (ownerRetreiver,poincut,semanticsRetreiver?,predicate?,raisePoint?)>
<!ATTLIST localEvent
        name NMTOKEN #REQUIRED
        package NMTOKEN #IMPLIED
>
<!--Message-->
<!ELEMENT message (sender,receiver)>
<!--pattern-->
<!ELEMENT poincut EMPTY>
<!ATTLIST poincut pattern CDATA #REQUIRED>
<!--ownerRetreiver-->
<!ELEMENT ownerRetreiver EMPTY>
<!ATTLIST ownerRetreiver className NMTOKEN #REQUIRED>
<!--Predicate-->
<!ELEMENT predicate EMPTY>
<!ATTLIST predicate className NMTOKEN #REQUIRED>
<!--Receiver-->
<!ELEMENT receiver
(ownerRetreiver,poincut,sourceRetreiver,semanticsRetreiver?,predicate?,raisePoint?)>
<!ATTLIST receiver
        name NMTOKEN #REQUIRED
        package NMTOKEN #IMPLIED
>
<!--SemanticsRetreiver-->
<!ELEMENT semanticsRetreiver EMPTY>
<!ATTLIST semanticsRetreiver className NMTOKEN #REQUIRED>
<!--Sender-->
<!ELEMENT sender
(ownerRetreiver,poincut,destinationRetreiver,semanticsRetreiver?,predicate?,raisePoint?)>
<!ATTLIST sender
        name NMTOKEN #REQUIRED
        package NMTOKEN #IMPLIED
>
<!--SourceRetreiver-->
<!ELEMENT sourceRetreiver EMPTY>
<!ATTLIST sourceRetreiver className NMTOKEN #REQUIRED>
<!--Trace-->
<!ELEMENT trace (localEvent|message)*>
```

Código 2 - DTD Lenguaje

6 Prueba de concepto

En esta sección se explica como se resolvió el problema planteado en este trabajo, para esto se desarrolló un sistema a modo de prueba de concepto de la técnica.
Primero se explicará la arquitectura utilizada y como es la interacción entre los componentes más importantes luego se presentará una visualización gráfica de la traza global y por último se detallan las metodologías utilizadas y el manual de usuario.
Para poder entender mejor la arquitectura se utilizará la misma estructura con la que se presentó el problema, primero la especificación y generación de eventos y luego como se genera la traza global y la implementación del algoritmo propuesto en el trabajo.

6.1 Especificación y generación de Eventos

Si bien la técnica propuesta en la sección 3.1 de la arquitectura de la solución es válida para cualquier sistema, para implementar un mecanismo de instrumentación debe conocerse de antemano el lenguaje de programación en el que está escrita la DAUT. En principio porque instrumentar implica modificar el sistema de forma automática y esto depende de si el lenguaje utiliza un compilador, es interpretado, usa una máquina virtual o utiliza alguna variación de estos.
Para el lenguaje de programación Java, que utiliza un compilador para generar un *byte code* que luego necesita de una máquina virtual, existe una herramienta que permite trabajar con aspectos ortogonales de un sistema de forma completamente separada para luego "tejerlos" en el proceso de compilación (*weave*) en un solo producto. Esta herramienta se llama AspectJ [ASPECTJ].
Tomando la **generación de eventos** como uno de los **aspectos** de un sistema, la implementación se valió de esta herramienta para desarrollar el instrumentador que combina la definición de eventos escrita con el lenguaje MAALEDL con sus clases soporte y la DAUT. Esta decisión trajo aparejada la restricción de solo poder instrumentar sistemas desarrollados en Java. Para poder instrumentar DAUT escritas en otro lenguaje hay que modificar levemente el lenguaje y encontrar una herramienta que funcione de manera similar a AspectJ.
La Fig. 9 muestra la arquitectura utilizada para el sistema que construye al Modulo de Generación de Eventos (EGM) para una DAUT. El proceso consiste en interpretar la especificación de eventos en lenguaje MAALEDL y, a través del *Parser*, generar un conjunto de objetos que representan la Definición de la Traza. Es importante destacar que el proceso descrito anteriormente es sólo una manera de hacerlo ya que en la implementación se encuentra reificada la Definición de la Traza, esto se explica en detalle en la sección 6.1.1.
Luego la componente *Code Generator* combina un conjunto de plantillas de código AspectJ con los objetos de la definición de eventos para generar código de AspectJ del modulo EGM. Este último módulo se compone del código AspectJ generado y de las clases soporte implementadas por el TI.

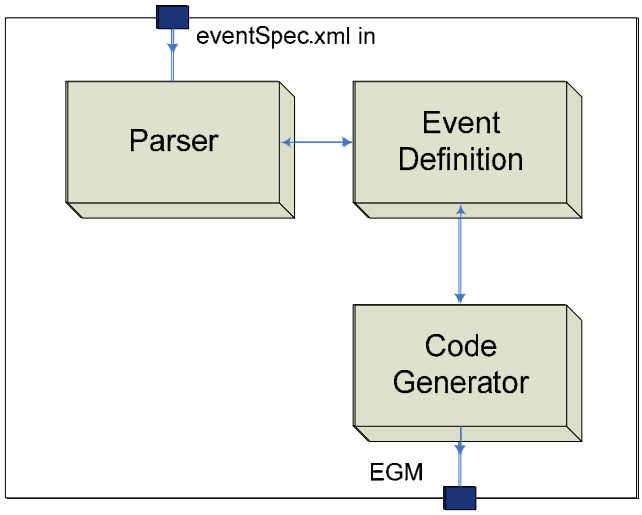

Fig. 9 - Arquitectura generación de código AspectJ

Parser:	Es la componente que tiene la responsabilidad de parsear la especificación de los eventos escritos en la versión en xml del lenguaje MAALEDL. Para implementar el Parser se utilizó Digester (http://jakarta.apache.org/commons/digester/).
Event Definition:	Esta componente representa la especificación de cada uno de los eventos que fueron programados en el lenguaje MAALEDL. También se puede ver como la estructura de datos que se obtiene de la lectura del archivo de especificación de eventos.
Code Generator:	Componente que se encarga de generar código AspectJ. Utiliza la información contenida en Event Definition para producir el código de los aspectos.

Por ultimo, solo resta compilar la DAUT y el EGM con el compilador de AspectJ para obtener la IDAUT, cuya estructura se explicó en Arquitectura de la solución
A continuación se explicarán más en detalle las componentes *EventDefinition*, *CodeGenerator* y el Modelo de Evento que están más orientados a usuarios que pretenden entender como está implementada la solución al detalle o están interesados en extender el modelo y por último se mostrará como se termina generando un evento y se lo envía al *TraceServer*.

6.1.1 Componente EventDefinition

En la Fig. 10 se puede ver el diagrama de clases, sin el protocolo, utilizado para modelar la definición de eventos y más abajo se explican las clases más importantes. Esta componente tiene como clase principal a *TraceDefinition* que es un conjunto de todos los *EventDefinition* y los *MessageDefinition* que luego se transformarán en eventos que aparecerán en la traza. Esta clase no fue puesta en el gráfico para poder ver con claridad la definición de los eventos.
Se puede ver la relación directa y estrecha que hay entre este modelo de objetos y el lenguaje de especificación MAALEDL. Una vez *parseada* la definición de eventos escrita en MAALEDL se obtiene un conjunto de instancias de *EventDefinition* y *MessageDefinition*. Estos objetos son utilizados por el *CodeGenerator* para construir el **aspecto** de generación de eventos correspondiente a la definición.

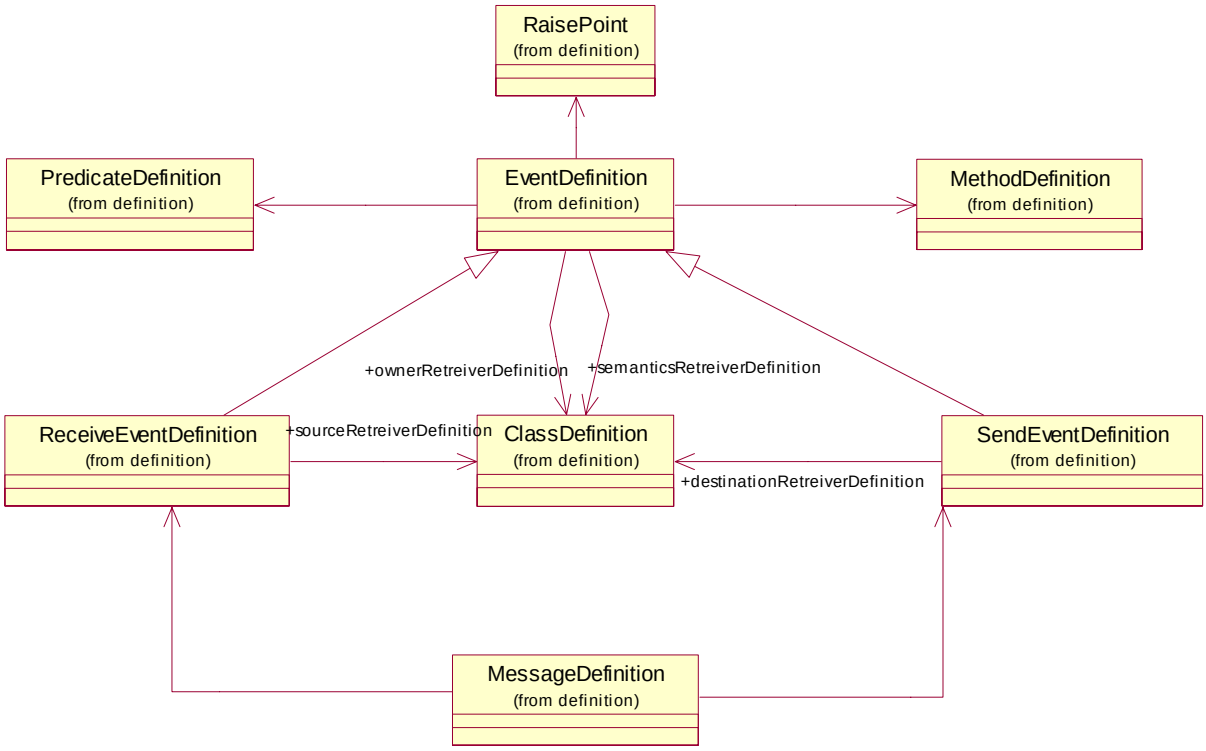

Fig. 10 - Definición de los eventos

EventDefinition:	Representa la definición de los eventos locales *LocalEvent*, estos contienen un nombre, la definición un método (*MethodDefinition*), un momento de ocurrencia, un predicado, la definición de un proveedor para el nodo en el que se ejecuta el evento (*NodeRetreiver*) y la definición de un proveedor para la semántica.
MessageDefinition:	Representa a la definición de los eventos denominados mensajes. La definición de un mensaje esta compuesta de la definición de un evento send (*SendEventDefinition*) y de un evento receive (*ReceiveEventDefinition*).
SendEventDefinition:	Representa la definición de los eventos de envío, que son

	aquellos que son disparados cuando se están enviando un mensaje a otro nodo o proceso. Además de tener la misma información que un *EventDefinition* tiene un proveedor del nodo destino.
ReceiveEventDefinition:	Tiene la responsabilidad de definir los eventos de recepción, que son lo eventos que se disparan cuando llega un mensaje desde otro proceso o nodo. Además de tener la misma información que un *EventDefinition* tiene un proveedor del nodo de donde vino el mensaje.
PredicateDefinition:	Tiene de responsabilidad de definir predicados para definir cuando un evento es generado y cuando no en función del contexto de ejecución.
MethodDefinition:	Se responsabiliza de definir sintácticamente en que método es donde se va a disparar el evento.
RaisePoint:	Define el momento de ejecución de un evento, puede ser antes de la ejecución de un método o después.

6.1.2 Componente CodeGenerator

Esta componente se encarga de generar código AspectJ y utiliza la información de las instancias de *EventDefinition*, contenidas en la instancia de *TraceDefinicion* obtenida luego del parséo de la especificación, para generar el código de los aspectos que se encargarán de generar y enviar los eventos.

Esta componente es un generador de código de *AspectJ* que a partir de la información que se obtiene del procesamiento de la especificación de eventos completa unas plantillas ya predefinidas, que serán explicadas más adelante, y produce el EGM específico para la DAUT. Por cada evento definido en MAALEDL se genera un Aspecto *Autogenerado* copiando una plantilla definiendo el nombre del evento, el método que al ejecutarse lo produce, el nombre de la clase que identifica el nodo en donde se está produciendo el evento, etc.

AspectJ permite organizar los aspectos en una clasificación con jerarquías tal como se puede hacer con las clases. También permite definir aspectos abstractos con métodos, *pointcuts* y *joinpoint* también abstractos [ASPECTJ].

Para este trabajo se diseño un modelo de aspectos, que se puede visualizar en las Fig. 11 - Modelos de Aspectos Abstractos, en donde se encuentra toda la lógica de "cómo" crear, configurar y enviar un evento al *TraceServer*. Esto es así porque no depende de la especificación de eventos y se puede separar la información de "cuando" generar un evento de la de "como" generarlo.

La información de "cuando" si depende directamente de la especificación de eventos y para ello se especializo cada uno de los aspectos abstractos definidos en la jerarquía mencionada anteriormente. Como resultado quedaron 3 "aspectos concretos" uno para cada tipo de evento el primero para los eventos locales Código 3 - Pantilla eventos locales y los otros 2 para los eventos que componen el mensaje, los eventos de envío y los eventos de recepción. Estas especializaciones se utilizan como plantillas a la hora de generar el código de aspecto correspondiente a cada definición de eventos. Cada una de las plantilla contienen *TAGS*[3] que se reemplazan en el proceso de

[3] El *tag* es meta-información que se agrega en la plantilla para que pueda ser reemplazada en tiempo de generación de código. Se definen con un @ delante y otro detrás. Por ejemplo, el *tag* que define el nombre del evento es **@NAME@**.

generación de código por la información específica de la definición del evento como nombre, punto de ejecución, etc. Más adelante se puede ver un ejemplo de todo el proceso completo para un evento.

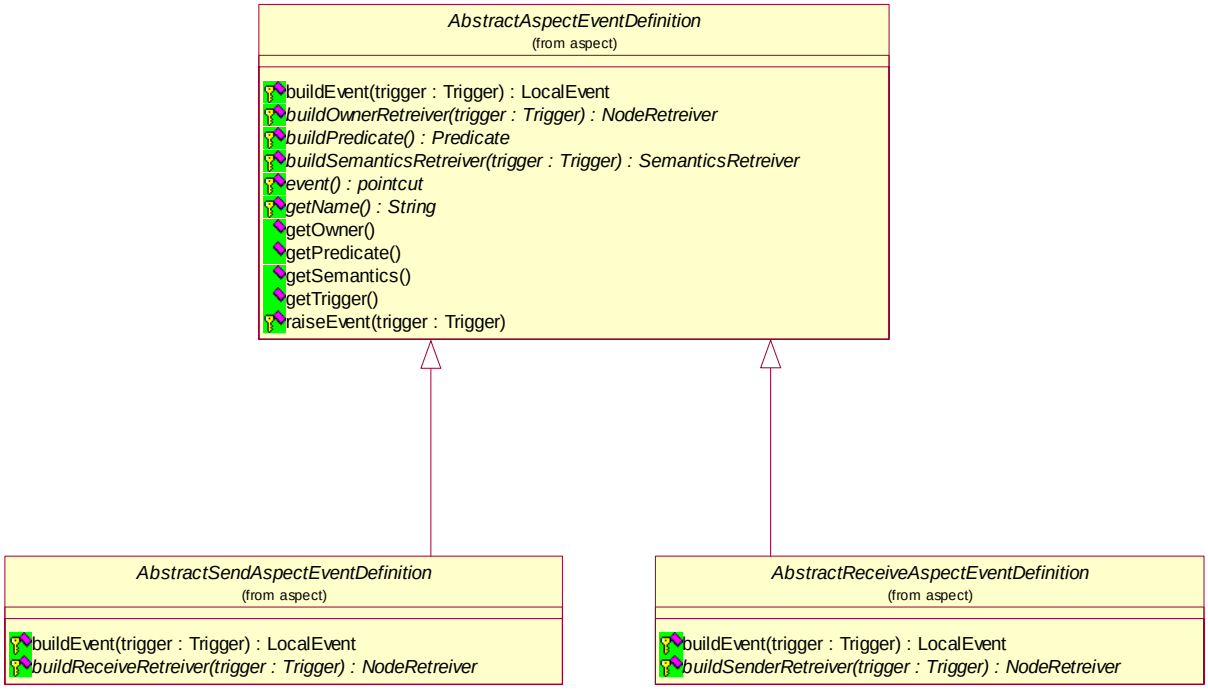

Fig. 11 - Modelos de Aspectos Abstractos

AbstractAspectEventDefinition:	Aspecto Abstracto que tiene la responsabilidad de generar los eventos locales. Define un ***pointcut*** y una serie de métodos abstractos que todos los aspectos concretos que lo hereden tienen que implementar. El ***pointcut*** abstracto se llama *event* y los métodos abstractos *buildOwnerRetreiver*, *buildPredicate*, *buildSemanticsRetreiver* y *getName*.
AbstractSendAspectEventDefinition:	Aspecto Abstracto que se encarga de generar los eventos de envío. Este aspecto tiene como método abstracto *buildReceiveRetreiver*.
AbstractReceiveAspectEventDefinition:	Aspecto Abstracto con la responsabilidad de generar los eventos de recepción. Este aspecto tiene como método abstracto *buildSenderRetreiver*.

A modo de ejemplo, se muestra a continuación el proceso de instrumentación completo para el evento definido en el Código 1 - Ejemplo de *OpenGate* usando el lenguaje MAALEDL.
Este código se generará en la carpeta que corresponde al paquete indicado en el atributo *package* de la definición y con el nombre de archivo *"OpenGateEventDefinition.java"*.

Como se puede ver en el Código 4 - Aspecto autogenerado de OpenGate el *OpenGateEventDefinition* extiende *AbstractAspectEventDefinition* que es uno de los aspectos abstractos de la Fig. 11 - Modelos de Aspectos Abstractos.
El código auto-generado de *OpenGateEventDefinition* utiliza para generarse la plantilla definida en el Código 3 - Pantilla eventos locales.
Una vez generado el código aspecto para cada uno de los eventos sólo resta compilar la DAUT con los aspectos generados para generar las IDAUT ya instrumentada.

```
package @PACKAGE@;
import @PARENT_PACKAGE@.AbstractAspectEventDefinition;
import ar.uba.dc.maal.net.*;
import ar.uba.dc.maal.event.*;
import ar.uba.dc.maal.event.semantics.*;
import org.aspectj.lang.*;
public aspect @NAME@EventDefinition extends AbstractAspectEventDefinition{
	protected pointcut event(): @PATTERN@;
	protected String getName(){
		return "@NAME@";
	}
	protected Predicate buildPredicate(){
		try{
			Predicate predicate = @PREDICATE@;
			return predicate;
		} catch(Exception ex){
			log.error("Error creando el predicado del evento " + getName() , ex);
			return new FalsePredicate();
		}
	}
	protected SemanticsRetreiver buildSemanticsRetreiver(JoinPoint joinpoint){
		return createSemanticsRetreiver("@SEMANTICS@", joinpoint);
	}
	protected NodeRetreiver buildOwnerRetreiver(JoinPoint joinpoint){
		return createNodeRetreiver("@OWNER@", joinpoint);
	}
	@RAISEPOINT@() : event() {
		if (getPredicate().evaluate(thisJoinPoint)){
			raiseEvent(thisJoinPoint);
		}
	}
}
```

Código 3 - Pantilla eventos locales

```
package ar.uba.dc.maal.event;
import ar.uba.dc.maal.event.aspect.AbstractAspectEventDefinition;
import ar.uba.dc.maal.net.*;
import ar.uba.dc.maal.event.semantics.*;
public aspect OpenGateEventDefinition extends AbstractAspectEventDefinition{
	protected pointcut event(): call(public void ar.uba.dc.maal.train.CrossRail.openGate());
	protected String getName(){
		return "OpenGate";
	}
	protected Predicate buildPredicate(){
		try{
			Predicate predicate = (Predicate) Class.forName("ar.uba.dc.maal.train.predicate.MyPredicate").newInstance();
			return predicate;
		} catch(Exception ex){
			log.error("Error creando el predicado del evento " + getName() , ex);
			return new FalsePredicate();
		}
	}
	protected SemanticsRetreiver buildSemanticsRetreiver(JoinPoint joinpoint){
		return createSemanticsRetreiver("ar.uba.dc.maal.train.semantics.OpenGateSemanticsRetreiver", joinpoint);
	}
	protected NodeRetreiver buildOwnerRetreiver(JoinPoint joinpoint){
		return createNodeRetreiver("ar.uba.dc.maal.train.node.CrossRailNodeRetreiver", joinpoint);
	}
	after() : event() {
		if (getPredicate().evaluate(joinpoint)){
			raiseEvent(joinpoint);
		}
	}
}
```

Código 4 - Aspecto autogenerado de OpenGate

6.1.3 Diseño de los eventos

Aquí se detalla el modelo de clases que representan a los eventos que ocurren durante la ejecución de la IDAUT y que son enviados al *TraceServer* para generar la Traza global.

La Fig. 12 muestra el diagrama general de las clases que componen este modelo. Para tener una idea mas clara de la diferencia entre estos eventos y los objetos de la definición de eventos (Fig. 10), se puede ver como ejemplo que un objeto de clase *EventDefinition* conoce un *ownerRetrieverDefinition* de clase *ClassDefinition* mientras que un *LocalEvent* conoce un *owner* de clase *Node*. La relación que hay entre estos objetos es la siguiente. A partir de un *EventDefinition* el sistema sabe, a través de la instrumentación, que durante la ejecución de la IDAUT se tienen que generar ciertos eventos. Estos eventos serán de clase *LocalEvent*. La clase que indica el *ownerRetrieverDefinition* es la que hay que instanciar para poder preguntarle cual es el objeto de clase *Node* que será el *owner* de cada uno de estos eventos.

Las instancias de la clase *Event* y sus subclases son efectivamente parte de las trazas de los sistemas y son los que interesan a la hora de la verificación, testeo y/o control.

Así, ambos modelos están relacionados, pero participan en momentos distintos del ciclo de vida del sistema.

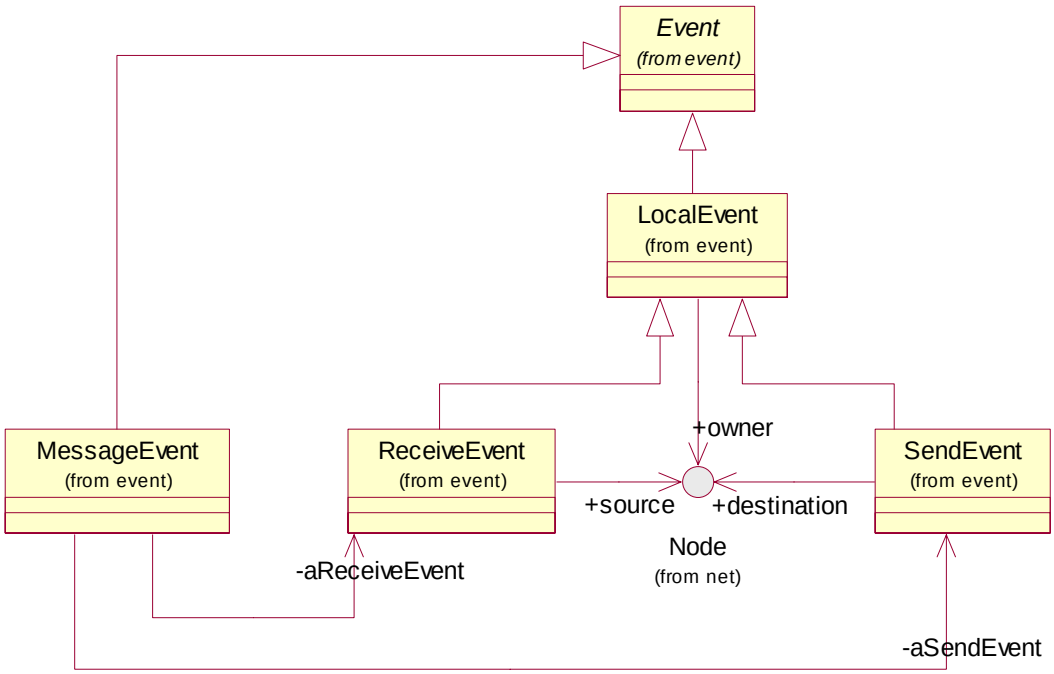

Fig. 12 - Diagrama de clase de los Eventos

Node:	Representa al nodo o proceso independiente del sistema distrinuido. Sus implementaciones tienen que implementar el método ***equals*** para que se pueda identificar de qué nodo vienen los eventos.
Event:	Clase abstracta que representa a un evento. Tiene una semántica y un nombre.
LocalEvent:	Evento local de un nodo. Contiene el nodo en el cual se generó el evento, y un *TimeStamp*.

ReceiveEvent:	Evento local de un nodo que indica que se recibió un mensaje de otro nodo. El protocolo incluye un método para obtener el nodo que generó el mensaje.
SendEvent:	Evento local que indica que se va a generar un mensaje. Y se le puede preguntar por el nodo destino del mensaje.
MessageEvent:	Evento que se genera en el *SystemTrace* cuando se recibieron los eventos locales *SendEvent* y *ReceiveEvent* correspondientes al mismo mensaje.

6.1.4 Creación y envío de un evento al *TraceServer*

Para terminar de entender el proceso completo se va a hacer un diagrama de secuencia explicando la secuencia de pasos y las colaboraciones entre los distintos objetos en la generación de una evento que simboliza la apertura de la barrera (*openGate*) de un supuesto paso a nivel de la ruta 2.
En la Fig. 13 se muestra mediante un diagrama de secuencias como se genera el evento ***openGate*** en el modulo EGM para el nodo ***Ruta2CrossRail***. El diagrama abarca desde que se genera el evento hasta que se envía el mismo a la capa de comunicación para ser enviado al Server. La generación de los otros tipos de eventos locales y los distintos nodos en similar a la generación de este evento, salvando que para generar un ***SendEvent*** es necesario también conocer el nodo destino del mensaje y en el ***ReceiveEvent*** hay que conocer el nodo fuente del mensaje.

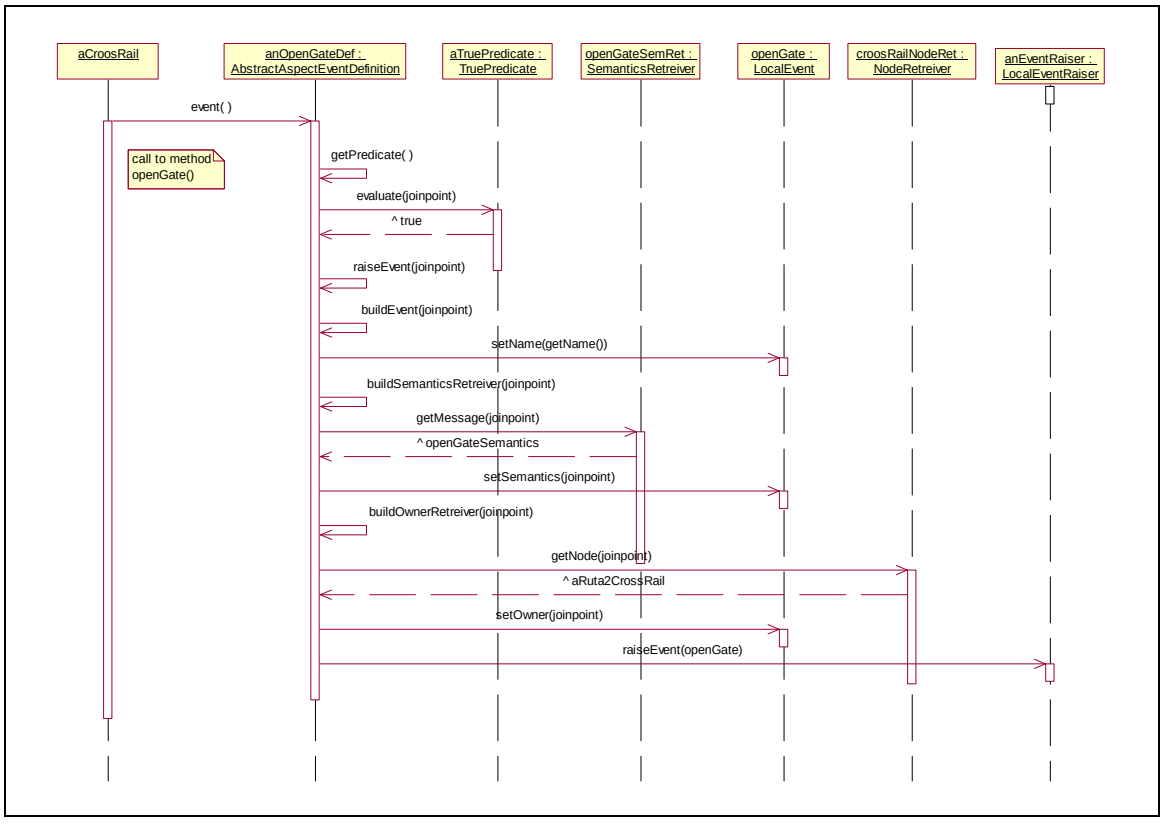

Fig. 13 - Raise openGateEvent en Ruta2CroosRail

Todo comienza cuando AspectJ detecta que se está ejecutando el método "public void ar.uba.dc.maal.train.CrossRail.openGate()". Entonces comienza la ejecución del aspecto *OpenGateEventDefinition* que luego termina enviando el evento *openGate* al *TraceServer*. Lo primero que hace este aspecto es construir el predicado para valuar si

se va a generar el evento o no, como en la especificación de este evento no se indicó ningún predicado se toma el predicado por omisión *TruePredicate*, que siempre da verdadero. Luego comienza el proceso de construcción y envío del mensaje al *TraceServer*. Se construye una instancia de un *LocalEvent* y se le asigna el nombre, la semántica y el nodo *owner* para luego ser enviado al *TraceServer* por medio de *anEventRaiser*.

6.2 Generación de la traza global

Para generar la traza global tiene que existir un proceso que escuche todos los eventos que generan las distintas instancias de IDAUT y las procese por el algoritmo de *Causality Sort* que va ordenando los eventos que llegan y una vez que se encuentran ordenados los va enviando de uno en uno a la salida representada por un *EventRaiser* que es una *interface* que se puede extender para hacer cualquier tipo de salida. Por ejemplo, la visualización es una implementación de *EventRaiser*. Los EventRaiser que ya vienen en la implementación son FileEventRaiser, SystemOutEventRaiser y el nombrado recién GraphicalEventRaiser.
A continuación se detallará la implementación del algoritmo *Causality Sort*.

6.2.1 Implementación del algoritmo Causality Sort

En la implementación, cada ciclo del algoritmo comienza con la llegada de un evento al *traceServer*. Todos los pedidos de registro de eventos son recibidos por un objeto de clase *DistributedEventsMerger* a través del método *addEvent()*. Este objeto es único para la traza e interactúa con otros que implementan las distintas partes del algoritmo.
En el algoritmo la entrada es una secuencia bien formada y acá recibe de a un evento por vez. La secuencia bien formada fue construida especialmente para que represente las pre-condiciones necesarias para que funcione el algoritmo. Estas precondiciones son que no se pierdan mensajes y que para cada evento de envío que llega al *TraceServer* llegue su correspondiente evento de recepción.
En la Fig. 14 se puede ver el modelo de clases utilizado para modelar todo el proceso, y a continuación del diagrama se explica conceptualmente cada una de las clases.
Luego se presentan una serie de diagramas de secuencia para mostrar como interactúan estos objetos en cada caso.

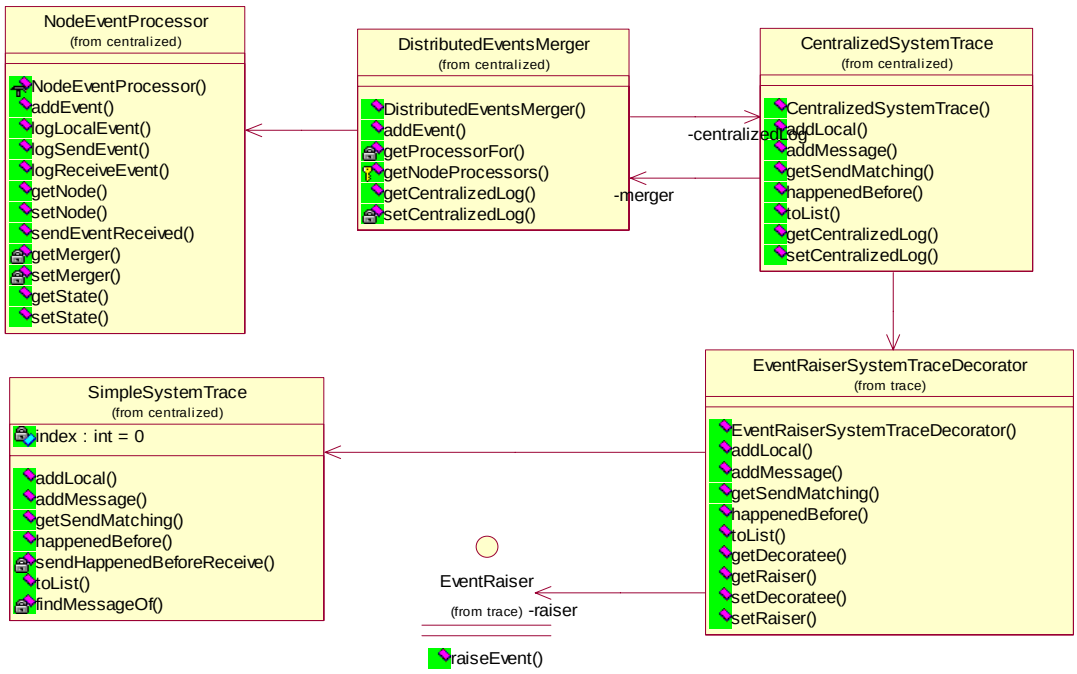

Fig. 14 - Clases Implementación del Algoritmo

NodeEventProcessor:	Existe una instancia para cada uno de los nodos que participan en el sistema. Se encarga de modificar y mantener el estado a medida que llegan los eventos para un nodo en particular. Tiene dos estados posibles: *EmptyState*: Todos los eventos llegaron respetando el orden causal. *NotEmptyState*: Es que el *NodeEventProcessor* recibió un *Receive Event* r antes de que llegue la notificación del *Send Event* s correspondiente a r, en otras palabras, el *NodeEventProcessor* va a estar trabado hasta que llegue s. También tiene la responsabilidad de reenviar los eventos al *CentralizedSystemTrace*.
DistributedEventMerger:	Recibe eventos locales (posiblemente desordenados) y se los pasa al NodeEventProsesor que corresponda.
CentralizedSystemTrace:	Se responsabiliza de avisar a todos los nodeprocessor cuando llega un evento de *Send*, para que en el caso de que alguno de ellos se encuentre trabado esperando (*NotEmptyState*) se pueda destrabar.
EventRaiserSystemTraceDecorator:	Los eventos llegan a este SystemTrace respetando el orden causal. Y tiene como responsabilidad funcionar como un *TeePipe*, enviando los eventos al *EventRaiser* configurado en el sistema y al SimpleSystemTrace.
SimpleSystemTrace:	Tiene la responsabilidad de colectar todos los eventos. Es un SystemTrace que respeta el orden causal.
EventRaiser:	Es una interfaz que se puede extender, para poder generar el System Trace de diferentes maneras y formatos, el que viene por default es un EventRaiser que loguea todo por Standard Output. A este nivel se puede garantizar que cada uno de los eventos que llegan a esta interfaz ya fueron reordenados para que respeten el orden causal de los mismo.

Lo que sigue es un conjunto de escenarios que muestran como estos objetos resuelven la llegada de distintos tipos de eventos para los casos particulares más importantes de estados del *traceServer*. Cada uno empieza cuando llega el evento al *merger* y terminan cuando el algoritmo esta listo para recibir el próximo evento.

6.2.1.1 Escenario 1

Se recibe un evento local (*alocalEvent*) de un nodo que no tiene ningún evento *receive* esperando por su *send* para destrabar al procesador (*EmptyState*). Éste es el caso más simple, el evento se registra directamente en la traza.

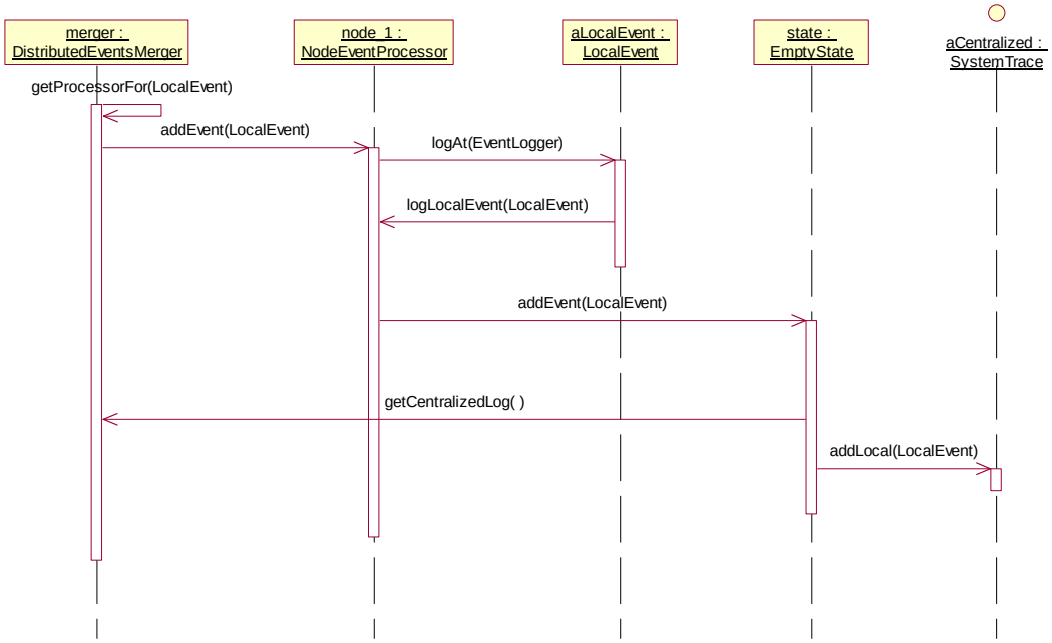

Fig. 15 – Escenario 1

6.2.1.2 Escenario 2

Llega al *traceServer* un evento de recepción (*receiveEvent*). El *NodeEventProcessor* de su nodo *owner* está en estado *EmptyState* y previamente había llegado el evento de envío correspondiente. También es un caso simple porque el *sendEvent* del evento está registrado. El *receiveEvent* se registra en la traza.

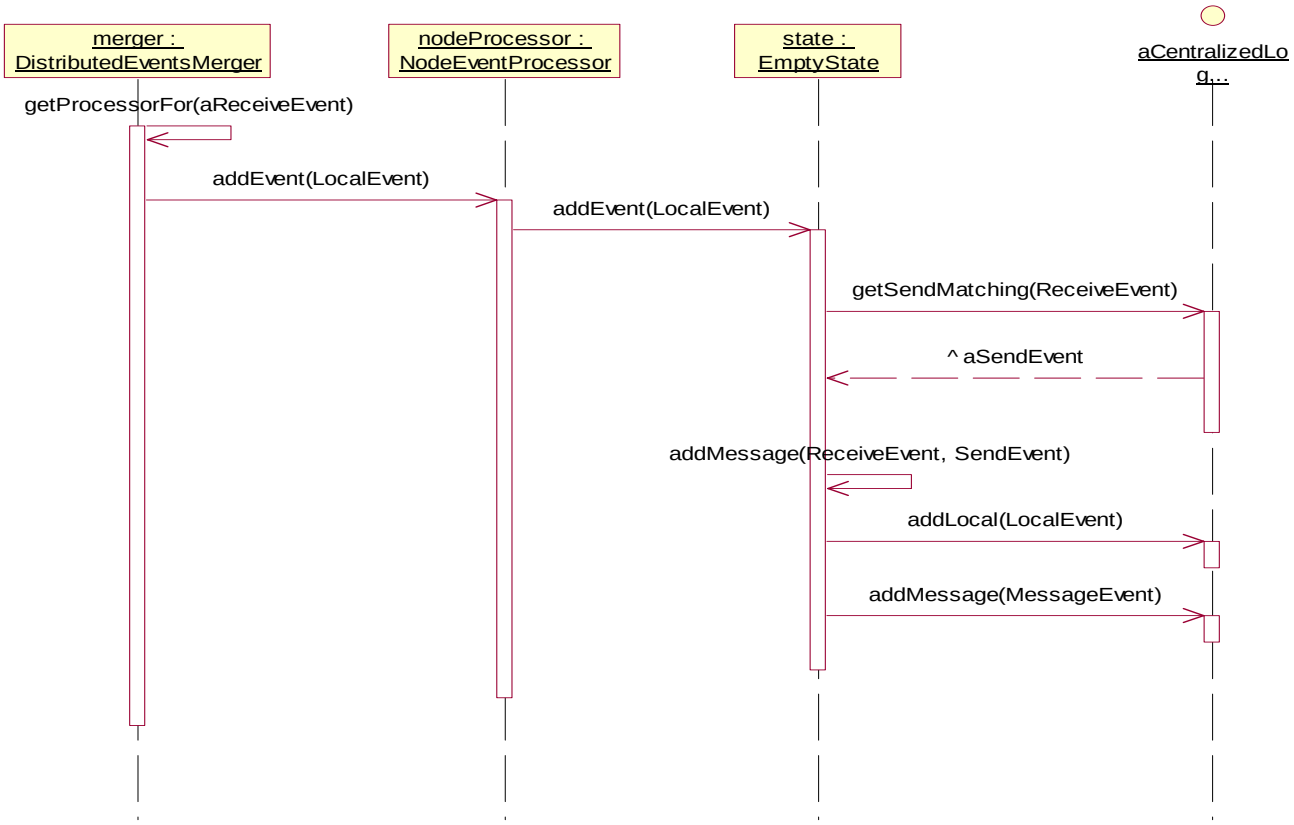

Fig. 16 – Escenario 2

6.2.1.3 Escenario 3

Llega un *receiveEvent* con el *NodeEventProcessor* de su nodo *owner* en *EmptyState* sin que previamente haya llegado el *SendEvent* correspondiente. Consecuentemente el procesador cambia al estado *NotEmptyState*.

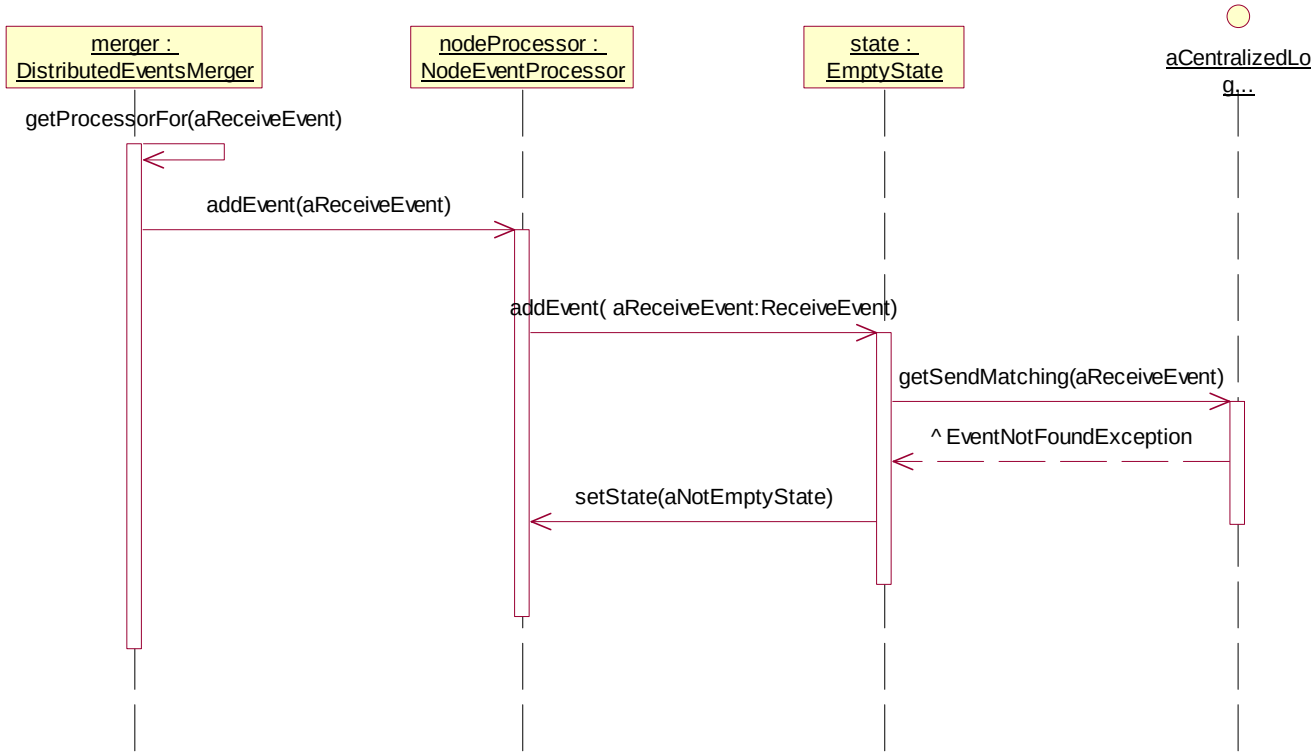

Fig. 17 – Escenario 3

6.2.1.4 Escenario 4

Se recibe un *sendEvent* para destrabar un *NodeEventProcessor* que está en *NotEmptyState*. Todos los eventos encolados en el *processor* del nodo *owner* del *receiveEvent* correspondiente al nuevo evento son reenviados al *DistributedEventsMerger*, acción equivalente a ser reinsertados en la secuencia de entrada *S* del algoritmo C*ausality Sort*

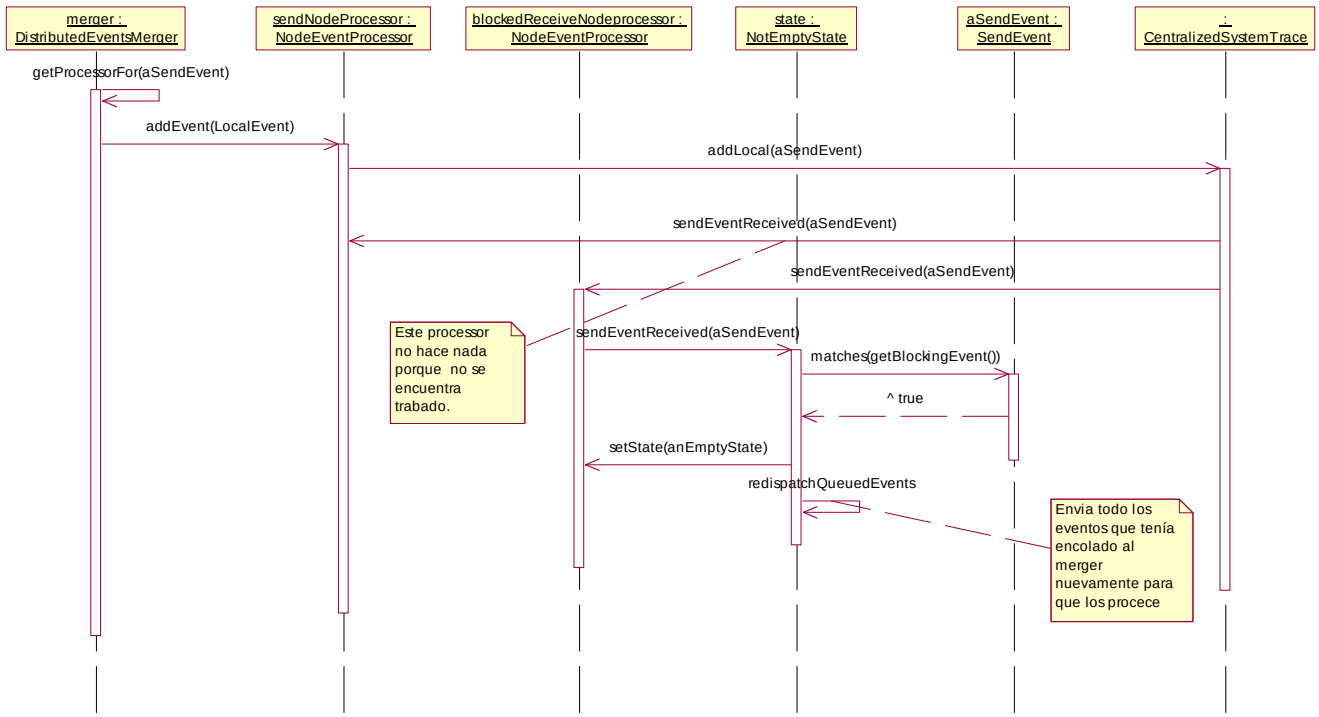

Fig. 18 - Escenario 4

6.3 Visualización

Para poder seguir de manera más simple la corrida de una IDAUT en el ***TraceServer*** se implementó una visualización del ***System Trace*** con un componente gráfico muy similar a un diagrama de secuencia que se va actualizando a medida que se suceden los eventos. Esta visualización es dinámica con respecto a eventos y a nodos, es decir, a medida que se van agregando nodos al sistema también se van agregando a la visualización. Se puede observar un ejemplo de la visualización en la Fig. 19.

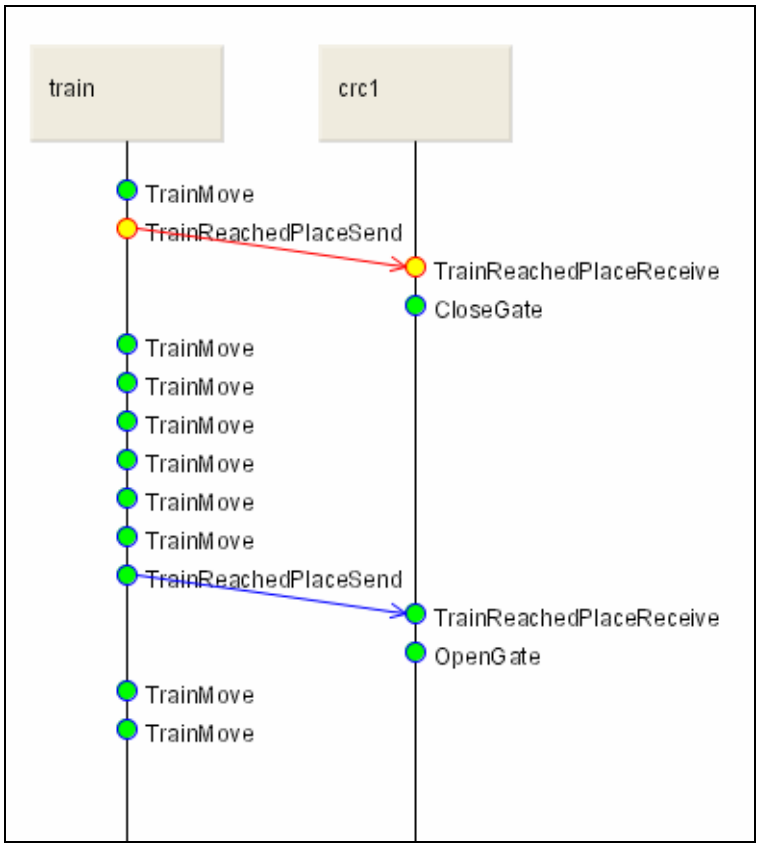

Fig. 19 - Visualización de eventos

La visualización tiene las siguientes propiedades: cada nodo está representado por una caja con el nombre del nodo y con una línea de tiempo, tiene configurado el *tooltip text* sobre la caja con una descripción más completa del nodo tomada del método *toString()* del nodo y sobre la línea de tiempo tiene configurado el *tooltip text* con el nombre del nodo que es el mismo que está en la caja. Sobre la línea de tiempo se van depositando los distintos eventos locales que sucedieron en este nodo, los eventos están representados por círculos y junto a estos sobre la derecha se puede ver el nombre del evento, sobre el circulo está configurado un *tooltip text* con la semántica de evento en formato texto. Cuando hay un mensaje de un nodo a otro se puede ver el evento de envío registrado en la línea de tiempo del nodo que envió el mensaje y el evento de recepción en la línea de tiempo del nodo que recibió el mensaje unidos por una flecha que indica la orientación del mensaje.
Se puede ir moviendo la visualización con el *Mouse* para ver los eventos y los nodos registrados en la visualización.

6.4 Metodología de desarrollo

Para el desarrollo de este trabajo se han considerado diferentes aspectos del proceso de desarrollo de Software, que son considerados importantes para lograr un resultado exitoso en la construcción de una herramienta que resuelva el problema planteado.

6.4.1 Diseño e implementación

Luego de resolver formalmente el problema, se comenzó a elaborar la herramienta MAAL.

Para la implementación se utilizó el lenguaje *Java (*www.java.sun.com), utilizando el IDE *Eclipse* (www.eclipse.org). Esta herramienta provee numerosas funcionalidades en lo que respecta a manejo de código fuente, referencias a clases, diagramas UML, refactoring, integración con otras herramientas, etc.

Para la instrumentación de los eventos se utilizó AspectJ (www.aspectj.org) con el *plugin* de *Eclipse AJDT* (http://www.eclipse.org/ajdt) que permite la edición, el chequeo de sintaxis, debugging, etc. del código aspecto.

Para generar los parsers que implementan las sintaxis concretas de *MAALEDL*, se utilizó *Digester* (http://jakarta.apache.org/commons/digester/), herramienta que facilita el diseño y la implementación de lenguajes escritos en XML. Para *logging* y manejo de colecciones se utilizó las librerías de *Jakarta commons* (http://jakarta.apache.org/commons/logging/)
(http://jakarta.apache.org/commons/collections/)

En ciertas partes de la implementación, y cuando se consideró que la complejidad del código era importante, se utilizó la técnica de Pair Programming (www.extremeprogramming.org/rules/pair.html) para asegurar mejor calidad y menor cantidad de defectos sobre el mismo. En estas partes es donde se le prestó especial atención a los *Test de Unidad.*

Una vez terminado un primer Release estable y con todas las funcionalidades requeridas, se procedió a realizar Reverse Engineering, utilizando *Rational Rose* para generar diagramas de clases que representen las distintas relaciones existentes entre las mismas.

Para la visualización se utilizó *Piccolo* (http://www.cs.umd.edu/hcil/piccolo/) es una herramienta que facilita la construcción de visualizaciones ampliamente utilizada en visualización de la información tanto en el campo académico como en el área comercial.

6.4.2 Configuration Management y administración de Artifacts

Para poder trabajar concurrentemente sobre el proyecto, se ha establecido un repositorio centralizado de Artifacts (código fuente, documentos, archivos de configuración, imágenes, etc.), utilizando como herramienta *CVS* (www.cvs.org). La interfaz de uso desde Windows para *CVS* utilizada fue *Eclipse y Tortoise* (www.wincvs.org).

Para lograr acceso remoto, se utilizó comunicación encriptada sobre Internet, utilizando el producto *Putty* (www.chiark.greenend.org.uk/~sgtatham/putty/), que permitió a través de SSH, acceder al *CVS*.
Así, se logró trabajar en una forma geográficamente distribuida, utilizando la técnica de Updates y Commits.

Una vez terminadas las primeras versiones de implementación, se comenzó a realizar el deployment de la aplicación. Para esto, se utilizó la herramienta *Ant* (www.jakarta.apache.org/ant), siendo ésta un Make multiplataforma, que nos permitió generar los entregables.

6.4.3 Testing

Durante todo el proceso de implementación en Java, se ha utilizado la herramienta *JUnit* (www.junit.org), que ha permitido realizar tests automatizados de unidad, tests de regresión y tests funcionales. Esta herramienta posee una integración con *Eclipse*, permitiendo que desde el ambiente de desarrollo se lance la ejecución de los tests.

Cada vez que se agrega nueva funcionalidad, o se modifica alguna parte del código, se ejecutan nuevamente los tests de unidad, permitiendo tener una especie de oráculo sobre lo que se espera obtener en cada caso de test y ver si efectivamente ha ocurrido lo esperado.

6.4.4 Javadoc

El código fuente fue cuidadosamente documentado, especialmente lo que respecta a métodos públicos, clases y variables de instancia. Para el formato de la documentación se utilizó el Javadoc, siendo el estándar en documentación de código Java.

6.5 Manual de Usuario

Este manual no es exhaustivo y está orientado a personas con conocimiento de JAVA. Su contenido abarca:

- Un tutorial para facilitar la especificación de eventos en el lenguaje MAALEDL.
- Una explicación de cómo compilar una aplicación instrumentada con MAAL.
- Una guía para sobre el uso del TraceServer
- Cómo iniciar una IDAUT para que envíe los eventos a un determinado TraceServer.

6.5.1 Tutorial para especificar eventos en MAALEDL

En esta sección se explicará uno de los procedimientos para poder instrumentar con MAAL una aplicación (DAUT) desarrollada en JAVA. A este procedimiento se le dio un orden, pero este orden es solamente organizativo y no necesariamente se tiene que cumplir completamente.

Para comenzar hay que implementar la clase que representa un nodo en el sistema, esta clase solamente tiene que implementar la interfaz *Nodo*.

A continuación se debe escribir una especificación con los eventos que se quieren agregar a la DAUT haciendo distinción entre los eventos que son locales y los mensajes que van de un nodo a otro. Luego para cada uno de los eventos locales especificados hay que seguir los siguientes pasos:

1.- Analizar el código de la DAUT buscando en que punto de la ejecución y, opcionalmente, en que estado *E* tiene que estar la misma para que dicho evento ocurra. Si el punto de ejecución encontrado no es exactamente antes o después de la ejecución de un método *M* hay que buscar la manera de alterar el código original de la DAUT para que así sea.

2.- Definir *M* por medio de un *patrón de llamada*. Por ejemplo, el patrón para un evento que sucede después de la ejecución de método *public void open()* que está implementado en la clase *Barrera* es "`call(public void Barrera.open())`". Para mayor detalle de la sintaxis para definir la expresión regular de los métodos ver *Pattern Definition* para *call* y *execute* en [ASPECTJ].

3.- Si es necesario, definir el predicado que determina si la DAUT está en el estado *E* encontrado en el paso 1. Si la ocurrencia del evento se da con la llamada al método *M* independientemente del estado de la aplicación no es necesario definir ningún predicado.

Para definir el predicado hay que implementar una clase que implementa la interfaz ***Predicate***

4.- Implementar una clase ***NodeRetreiver*** con un constructor sin parámetros y que, recibiendo el contexto de ejecución del método *M*, retorne una instancia del nodo que representa al *owner* del evento.

5.- Implementar una clase ***SemanticsRetreiver*** con un constructor sin parámetros y que, recibiendo el contexto de ejecución, de *M* retorne una instancia de la semántica del evento. Si no se quiere usar la semántica del evento no es necesario implementar esta clase. Esta semántica se puede utilizar para diferenciar eventos del mismo nombre.

6.- Con toda la información reunida en los pasos anteriores ya se puede escribir la especificación de este evento local en el lenguaje **MAALEDL**.

Los pasos 1, 2 y 6 se deben repetir para cada uno de los eventos locales especificados. En cambio, para los pasos 3, 4 y 5 a medida que se van especificando los distintos eventos se puede encontrar que se pueden reutilizar las clases ya implementadas.

En el caso de los eventos mensaje, como están compuestos de dos eventos locales (uno de envío y otro de recepción), los pasos a seguir para cada uno de estos son muy parecidos a los pasos de los eventos locales definidos anteriormente. La diferencia radica principalmente en que el evento de envío tiene que saber a que nodo está dirigido y el evento de recepción tiene que saber de que nodo viene. Para resolver este problema se agrega un paso más para cada uno y se elimina el paso 6. A continuación se detalla este paso extra para cada uno de los eventos del mensaje:

Evento de Recepción:

6.- Implementar una clase ***NodeRetreiver*** con un constructor sin parámetros que, recibiendo el contexto de ejecución del método *M*, retorne una instancia del nodo de donde vino el mensaje.

Evento de Envío:

6.- Implementar una clase ***NodeRetreiver*** con un constructor sin parámetros que, recibiendo el contexto de ejecución del método *M*, retorne una instancia del nodo a donde se dirige el mensaje.

Una vez completado los pasos para el evento de recepción y para el evento de envío ya se puede escribir la especificación del mensaje en **MAALEDL**.

6.5.2 Compilar la DAUT con eventos en MAALEDL

Al usar AspectJ como herramienta en la instrumentación de eventos especificados en MAALEDL primero se tienen que generar código de AspectJ para luego entretejerlo durante el proceso de compilación y generar la IDAUT con los eventos ya instrumentados. Este proceso se puede unificar en un solo paso o se puede separar para poder utilizar el código auto-generado para realizar seguimientos en tiempo de ejecución (*debug*).
MAAL provee dos formas para generar código AspectJ con los eventos especificados, una es por medio de llamadas a la API de Maal directamente o la otra con ant [ANT] que es una herramienta para *build* estilo *Make* para java.
En resumen las alternativas son generar todo en un solo paso o generar el código intermedio y luego entretejer el código aspecto generado. A continuación se explicará como generar el código por medio de la API y con ant. Y luego como entretejer el código AspectJ con la DAUT para generar la IDAUT. No se van a dar detalles de cómo generar todo junto ya que solo implica realizar una llamada detrás de la otra.

6.5.2.1 Generar código aspecto usando MAAL

La generación de código por medio de MAAL no es más que la ejecución estándar de un programa JAVA que se encuentra en el ***maal.jar***.
Esta clase *main* es "*ar.uba.dc.maal.event.CodeGenerator*" y puede ejecutarse a través del intérprete de comandos. Los parámetros que recibe desde la línea de comando son los que se encuentran en el siguiente cuadro.

Nombre	Nombre Corto	Descripción
file	f	Directorio donde va a ser generado el código AspectJ con los eventos de control. Se asume que el directorio es asignado de manera exclusive para los eventos de control, antes de generar el código se borrará todo el contenido del mismo.
dir	d	Archivo que contiene los eventos especificados en el Lenguaje MAALEDL.

A continuación se muestran unos ejemplos que toman la especificación de un archivo denominado “trace.xml” que se encuentra en el directorio “c:\maal” y deja el código aspecto generado en el directorio “c:\maal\auto-events\”. El primero utiliza los nombres largos y el segundo los nombres cortos para los parámetros.

```
java -cp %MAAL_HOME%\lib\maal.jar ar.uba.dc.maal.event.CodeGenerator --
file=c:/maal/trace.xml --dir=c:/maal/auto-events/
```

Ejemplo 1 - Nombres largos

```
java -cp %MAAL_HOME%\lib\maal.jar ar.uba.dc.maal.event.CodeGenerator -
fc:/maal/trace.xml -dc:/maal/ auto-events/
```

Ejemplo 2 - Nombres cortos

6.5.2.2 Generar código aspecto usando ANT

Para generar el código aspecto hay que configurar la tarea de ANT (*ant task*) **“maal”** dentro del *target* que se quiere ejecutar, los parámetros que acepta la tarea no casualmente están muy relacionados con los parámetros que se mencionan en la sección anterior, estos se detallan en el cuadro que está a continuación:

Attribute	Description	Required
destDir	Directorio donde va a ser generado el código AspectJ con los eventos de control. Se asume que el directorio es asignado de manera exclusive para los eventos de control, antes de generar el código se borrará todo el contenido del mismo.	Yes
eventDefinitionFile	Archivo que contiene los eventos especificados en el Lenguaje MAALEDL.	Yes

Para que la tarea funcione tiene que estar dentro del CLASSPATH de ANT el *jar* de *maal*. A continuación se puede un ejemplo:
Este ejemplo genera el código de los eventos que están especificados en el archivo “trace.xml” que se encuentra en el directorio “c:\maal”. Supone que todas las librerías necesarias inclusive “maal.jar” se encuentran en el directorio “c:\maal\lib”. A continuación se encuentra la configuración que tiene que estar en el build de ANT.

```
    <target name="init.maal">
        <taskdef resource="ar/uba/dc/maal/ant/taskdef/maalTaskDef.properties" >
            <classpath>
                <fileset dir="c:/maal/lib/" includes="**/*.jar"/>
            </classpath>
        </taskdef>
    </target>

    <target name="generateCode" depends=”init.maal”>
        <echo message="Ejecutando maal" />
        <maal   destDir="c:/maal/exanples/"   eventDefinitionFile="c:/trace.xml"/>
    </target>
```

Suponiendo que el archivo de configuración de ANT se llame “*build.xml*” y que este se encuentre en el HOME del proyecto a instrumentar. Para que se genere el código hay que desde el HOME del proyecto ejecutar `ant generateCode` y listo.

6.5.3 Compilar código aspecto con la DAUT

Para compilar y entretejer el código aspecto en la DAUT solamente hay que usar el compilador de [ASPECTJ] en lugar de usar el compilador de JAVA. Los parámetros que recibe el compilador de AspectJ son similares a los que recibe JAVA. De la misma manera que existe una tarea de ANT para compilar con el compilador estándar de JAVA AspectJ provee una tarea para compilar con el compilador de AspectJ.
A continuación se muestra un ejemplo de cómo configurar un *build* de [ANT] para que compile usando el compilador de AspectJ.

```
<!-- Directorio donde se encuentran las librerias necesarias -->
<property name="lib.path"                    value="lib"/>
<!-- Directorio donde se encuentra el código de la aplicación -->
<property name="src.path"                    value="src"/>
<!-- Directorio donde se encuentra el código extra necesario para que funcione maal-->
<property name="src.maal.path"                       value="src"/>
<!-- Directorio donde se encuentra el código autogenerado -->
<property name="auto.event.path"             value="auto-event-src"/>
<!-- Durectorio donde se van a generar los .class -->
<property name="build.path"                  value="classes"/>

<path id="lib.pathId">
       <fileset dir="${lib.path}" includes="**/*.jar"/>
</path>

<!-- Inicializa las tareas de aspectJ -->
<target name="init.taskdefs" >
       <!-- sets name of new task to iajc, old task to ajc -->
       <taskdef resource="org/aspectj/tools/ant/taskdefs/aspectjTaskdefs.properties">
              <classpath>
                     <fileset dir="${lib.path}" includes="aspectj*.jar"/>
              </classpath>
       </taskdef>
</target>

<target name="compile-aj" depends="init.taskdefs" description="Compila usando el
compilador de aspect">
       <echo message="compilando con el compilador de aspectJ" />
       <!-- can use ajc or iajc here -->
       <iajc destdir="${build.path}">
              <src path="${src.path}"/>
              <src path="${src.maal.path}"/>
              <src path="${auto.event.path}"/>
              <classpath>
                     <path refid="lib.pathId"/>
              </classpath>
       </iajc>
</target>
```

El *target* que hay que llamar para compilar es *compile-aj* que toma como código el que se encuentra en donde indican los valores de las propiedades *src.path* (supone ser

el código original de la aplicación), *src.maal.path* (supone ser el código adicional agregado para que funcione maal, por ejemplo ahí se encuentran los distintos retreivers) y *auto.event.path* (sería el código aspecto con los eventos generado con los procedimientos descritos en alguna de las 2 secciones anteriores) y como CLASSPATH el que se encuentra definido en *lib.path.id*. Este target depende del target *init.taskdefs* que es el que se encarga de inicializar el compilador de AscpecJ.

6.5.4 Cómo ejecutar el TraceServer

Para ejecutar el TraceServer hay que tener previamente corriendo un servidor de nombre de RMI con el CLASSPATH correctamente setado, esto quiere decir que tiene que tener las versiones correctas de MAAL y sus dependencias. Versión correcta significa que tiene que ser la misma versión la que corre el Server, el cliente y la Registry. Una vez que está el RMI Server levantado ya se puede levantar el TraceServer. En orden de correr el Trace Server es necesario setearle, al igual que al RMI Server, correctamente el CLASSPATH con las versiones correctas de Maal y sus dependencias y las versiones correctas de la IDAUT con sus dependencias.
A continuación se muestra una impresión de la utilización del Trace Server tal como la muestra el programa cuando no tiene algún parámetro bien seteado.

```
usage: maalServer

 -e,--eventraiser <className>   Nombre de la implementacion de EventRaiser
                                que va a recibir el trace.

 -f,--file <file>               Archivo donde se quiere registrar el log.

 -s,--url <url>                 Url del server donde esta escuchando la
                                RMIRegistry, sino se setea toma localhost

 -p,--port <port>               Puerto donde está escuchando la
                                RMIRegistry, sino se setae toma el default.

 -s,--std                       Imprime el system trace por standar output

 -v,--viz                       Muestra gráficamente el log
```

Es necesario que se utilice al menos un EventRaiser de los que permite configurar, si no se configura al menos uno va a ser imposible de ver la traza global. Los que trae son:

- Viz: Un visualizador de la traza global gráfico.
- File: Escribe el system trace en un archivo.
- Std: Escribe el system trace en la salida estandar.
- EventRaiser: El la manera en que se puede pluguear un event raiser customizado. Tiene que ser el nombre de una clase que implementa EventRaiser

6.5.5 Cómo ejecutar la IDAUT

Para correr la IDAUT hay que ejecutarla de la misma manera que antes ya que el código generado con el compilador de AspectJ es compatible con el generado con JAVA sólo necesita que se agregue en el CLASSPATH el run-time de AspectJ (*aspectjrt.jar*). Para que genere los eventos es necesario especificarle donde se encuentra el ***TraceServer***, esto se hace agregándole la *System Property traceserver.url* con el nombre o la IP de donde se encuentra el servidor de nombres de RMI, este servidor de nombres tiene que ser el mismo que se le seteó al ***TraceServer***. También

es necesario agregar en el CLASSPATH de la IDAUT las librerías necesarias para que **Maal** funcione que son: common-beanutils, commons-collection, common-logging y log4j. Antes de levantar la aplicación hay que asegurarse que esté levantado correctamente del ***TraceServer***.

7 Caso de estudio: Ferrocarril

Como primer caso de estudio y como parte del arnés de test para el desarrollo de la prueba de concepto se desarrolló una simulación de un ferrocarril con cruces de barrera.
En este modelo existen dos componentes principales: trenes y barreras. La simulación abarca el proceso en que cada barrera se cierra al pasar un tren y se abre luego que ya pasó. Para esto, cada barrera esta equipada con un sensor de acercamiento y un sensor de alejamiento.
A nivel conceptual, el caso se diseñó como un sistema distribuido. Los nodos del sistema son los trenes (*Train*) y los cruces (*CrossRail*). En cada *CrossRail* están los dos sensores.

La Fig. 20 muestra un ejemplo de cómo se comunican los nodos para el caso más simple donde un solo tren atraviesa un cruce con barrera. Los sensores y el cruce están en el mismo nodo. Los únicos mensajes que se envían entre nodos distintos son *trainReachedPlace()*.

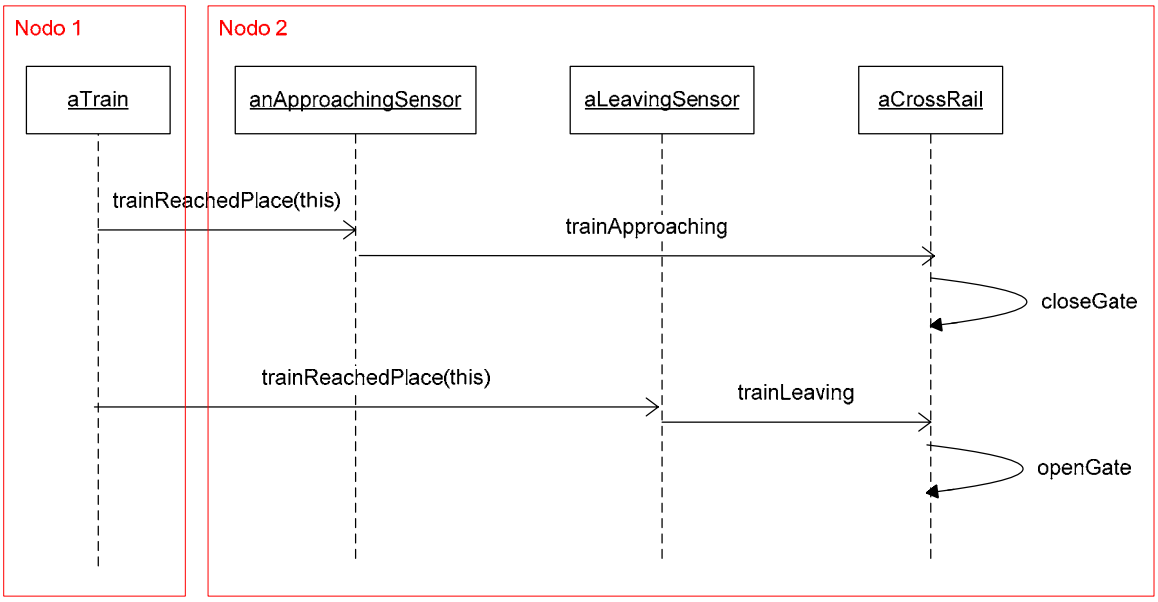

Fig. 20

La complejidad del sistema aumenta a medida que aumentan los trenes y los cruces, teniendo en cuenta, por ejemplo, que una barrera no debe abrirse al alejarse un tren si hay otro aun atravesándola. La Fig. 21 muestra éste caso.

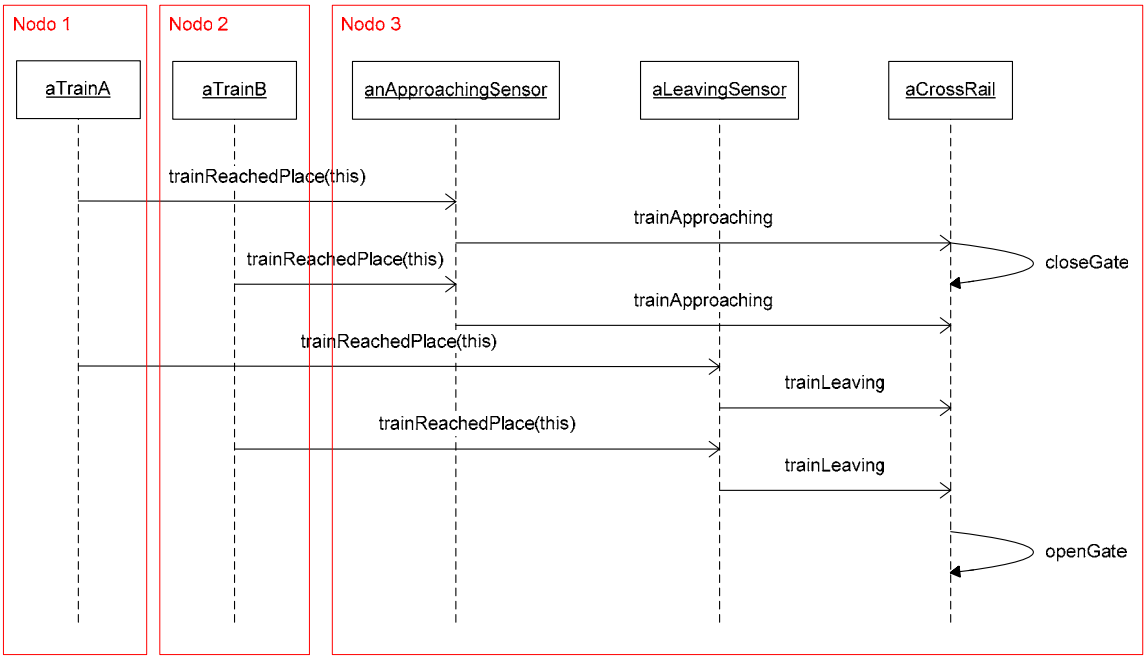

Fig. 21

Este modelo también permite definir restricciones temporales para su funcionamiento. Por ejemplo, podría requerirse que la barrera no se cierre en menos de 1 segundo después que el tren llegó al sensor de acercamiento.
A nivel implementativo, la simulación funciona de la siguiente manera. La vía del ferrocarril se divide en *n* posiciones, cada elemento de la simulación se puede encontrar en una posición a la vez. Si un cruce se encuentra en la posición *m*, sus sensores de acercamiento se encuentran a 5 posiciones de distancia, y sus sensores de alejamiento a 1 posición. Cuando un tren llega a la posición donde se encuentra un sensor, le envía un mensaje. No esta considerado en la simulación que un tren puede ser mas largo que una posición.
Cada cruce debe ser iniciado antes de que los trenes se pongan en movimiento. Una vez que los cruces están funcionando y esperando la llegada de los trenes, estos últimos pueden ponerse en marcha. Los trenes avanzan una posición a la vez pero, como corren en nodos separados, no necesariamente tengan la misma velocidad ni estén sincronizados.
El sistema permite ser configurado para simular la cantidad de trenes y de cruces que se deseen. Además, existe una implementación de cruce fallada, que no tiene en cuenta que más de un tren puede estar atravesando el cruce simultáneamente.

7.1 Instrumentación

En este caso de estudio se utilizó protocolo Java RMI para la comunicación entre los nodos. Este protocolo permite invocar métodos de objetos java instanciados en distintas maquinas virtuales.
Se eligió TCP/IP como medio de transporte de los mensajes RMI, aunque éste soporta otros protocolos como HTTP o SSL. RMI sobre TCP/IP cumple con los requisitos para utilizar MAAL para generar la traza global ya que los paquetes enviados a través de TCP/IP no se pierden ni se desordenan.
Los eventos que se configuraron para ser incluidos en la traza figuran en la Tabla 2.

Nombre	Tipo	Semántica
OpenGate	LocalEvent	La barrera se abre
CloseGate	LocalEvent	La barrera se cierra
TrainMove	LocalEvent	El tren llega a la cierta posición
TrainReachedPlaceSend	SendEvent	El tren envía a cierto sensor que llegó a su posición.
TrainReachedPlaceReceive	ReceiveEvent	El sensor recibe que cierto tren llegó a su posición.

Tabla 2

El resultado de esta configuración de la traza es que cada tren informa paso a paso su posición y su llegada a un sensor, cada sensor que detecta la llegada de un tren y cada barrera si se abre o se cierra. De esta forma se utilizan eventos locales y eventos de mensajes lo que permite testear en gran medida la prueba de concepto.
En la Fig. 23 se presenta la configuración de eventos en lenguaje MAALEDL que se utilizó para instrumentar el caso de estudio del ferrocarril.

7.2 Resultados obtenidos

La simulación se utilizó para numerosos tests que se realizaron durante el desarrollo de la prueba de concepto y demostró ser lo suficientemente versátil y a su vez simple para cumplir satisfactoriamente su tarea. A modo de ejemplo, la Fig. 22 muestra una visualización de una traza generada para una corrida con un tren y una barrera.
Para que la instrumentación no influya en el desarrollo de la simulación, ésta se construyó completamente en forma separada. Una vez finalizada, se procesó con MAAL.
Como se puede ver en la Fig. 23, la especificación de la traza es simple y concisa. El sistema no sufrió modificación alguna más allá del agregado de algunas clases de soporte para identificar los nodos y el uso de RMI como medio de comunicación entre los mismos no trajo ningún problema para MAAL.
Es decir que la implementación de la prueba de concepto sobre este caso de estudio cumplió los requerimientos en cuanto a su relación con el sistema a instrumentar.
Sin embargo, es importante aclarar que dada su simpleza, se considera que la simulación no fue lo suficientemente realista para representar un sistema distribuido actual. A este respecto de realizó un nuevo caso de estudio basado en una sistema real que se analiza en la próxima sección.

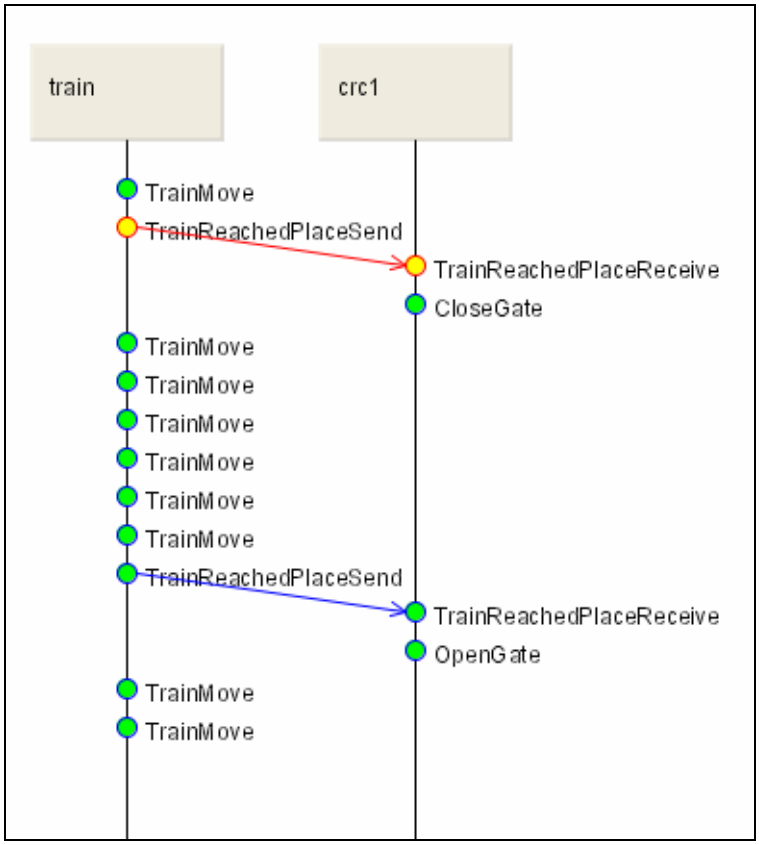

Fig. 22 - Visualización de un tren y una barrera

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<trace>
        <localEvent name="OpenGate" package="ar.uba.dc.maal.event">
                <ownerRetreiver
        className="ar.uba.dc.maal.train.node.CrossRailNodeRetreiver"/>
                <semanticsRetreiver
        className="ar.uba.dc.maal.train.semantics.OpenGateSemanticsRetreiver"/>
                <pointcut pattern="call(public void
        ar.uba.dc.maal.train.CrossRail.openGate())"/>
                <predicate className="ar.uba.dc.maal.event.TruePredicate" />
        </localEvent>
        <localEvent name="CloseGate" package="ar.uba.dc.maal.event">
                <ownerRetreiver
        className="ar.uba.dc.maal.train.node.CrossRailNodeRetreiver"/>
                <semanticsRetreiver
        className="ar.uba.dc.maal.train.semantics.CloseGateSemanticsRetreiver"/>
                <pointcut pattern="call(public void
        ar.uba.dc.maal.train.CrossRail.closeGate())" />
        </localEvent>
        <localEvent name="TrainMove" package="ar.uba.dc.maal.event">
                <ownerRetreiver
        className="ar.uba.dc.maal.train.node.TrainNodeRetreiver"/>
                <semanticsRetreiver
        className="ar.uba.dc.maal.train.semantics.TrainMoveSemanticsRetreiver"/>
                <pointcut pattern="call(public void
        ar.uba.dc.maal.train.Train.move())"/>
        </localEvent>
        <message>
                <sendEvent name="TrainReachedPlaceSend" package="ar.uba.dc.maal.event">
                        <pointcut pattern="execution(public void
        ar.uba.dc.maal.train.rmi.TrainSensorRmiProxy.trainReachedPlace(ar.uba.dc.maal.t
        rain.Train))"/>
                        <raisePoint>onInvoke</raisePoint>
                        <semanticsRetreiver
        className="ar.uba.dc.maal.train.semantics.TrainReachedPlaceSendSemanticsRetreiv
        er"/>
                        <ownerRetreiver
        className="ar.uba.dc.maal.train.rmi.TrainSensorRmiProxyOwnerNodeRetreiver"/>
                        <destinationRetreiver
        className="ar.uba.dc.maal.train.rmi.TrainSensorRmiProxyDestNodeRetreiver"/>
                </sendEvent>
                <receiveEvent name="TrainReachedPlaceReceive"
        package="ar.uba.dc.maal.event">
                        <pointcut pattern="execution(public void
        ar.uba.dc.maal.train.rmi.RmiTrainSensorImpl.trainReachedPlace(ar.uba.dc.maal.tr
        ain.Train) throws java.rmi.RemoteException)" />
                        <semanticsRetreiver
        className="ar.uba.dc.maal.train.semantics.TrainReachedPlaceReceiveSemanticsRetr
        eiver"/>
                        <ownerRetreiver
        className="ar.uba.dc.maal.train.rmi.TrainSensorImplOwnerNodeRetreiver"/>
                        <sourceRetreiver
        className="ar.uba.dc.maal.train.rmi.TrainSensorImplSourceNodeRetreiver"/>
                </receiveEvent>
        </message>
</trace>
```

Fig. 23

8 Caso de estudio - Azureus (Java BitTorrent)

Como segundo caso de estudio se utilizó el sistema ***Azureus***, una implementación del protocolo de distribución de archivos BitTorrent. ***Azureus*** es un sistema distribuido donde cada proceso o instancia participa en el proceso de distribuir la información entre todos maximizando el uso del ancho de banda global de la red. Esta es una aplicación popular dentro de las implementaciones del protocolo y la implementación cuenta con aproximadamente 1920 clases y solo el código ocupa 12.8 MB.
Para aprovecharlo como caso de estudio se tomó cada instancia de ***Azureus*** corriendo en una PC como un Nodo de **Maal** y se generó una traza global de todos los mensajes del protocolo que se intercambian entre nodo y nodo. Por último se realizó un análisis del proceso de aplicación de **Maal** respecto a los requerimientos del mismo.
En orden de entender el caso de estudio se hará un resumen introductorio del protocolo BitTorrent versión 1.0 [BTP1.0] y del funcionamiento del mismo.

8.1 Descripción del protocolo

BitTorrent es un protocolo *Peer2Peer* diseñado para la distribución de archivos a través de Internet. La ventaja que tiene con respecto a FTP u otros medios de transferencia es que a medida que cada uno de los *peers* va completando una parte del archivo que se está bajando automáticamente la comparte con los otros *peers*. Así, la carga de enviar los datos es repartida entre todos los nodos que participan de la distribución. Los participantes en el protocolo son:

Peer:	Es una instancia de un cliente de BitTorrent que está participando en la distribución de un archivo. Usualmente un *peer* no tiene el archivo completo, solo algunas partes.
Seed:	Es un *peer* que ya tiene todo el archivo completo y que sólo participa de manera colaborativa compartiendo el archivo que tiene completo.
Original Seed:	Es el *seed* que comenzó a compartir el archivo
Tracker:	Es el servicio de mediador que conecta a los *peers* entre si. El *tracker* no está directamente involucrado en la transferencia de datos y no tiene el una copia del archivo.
Torrent file:	Es un archivo que contiene meta-información sobre el archivo, este archivo es mucho más pequeño que el archivo que se desea compartir, por ejemplo, con un *torrent* de 80KBytes hay información para compartir un archivo de 800MBytes. Entre otras cosas el *torrent* contiene la lista de *trackers*.

Ya conocidos los participantes se verá como colaboran entre ellos para compartir un archivo. Para distribuir un archivo que se quiere compartir primero se debe crear un archivo *torrent* que sigue un formato especificado en el protocolo. Para esto, como mínimo, es necesario disponer de al menos un *tracker* y del archivo completo. (En general todas las implementaciones de *BitTorrent* proveen herramientas para crear este archivo). Para empezar a distribuir el archivo debe ejecutarse una instancia de un cliente *BitTorrent* indicándole que debe ser *seed* del *torrent* en cuestión. A su vez, éste le informa al *tracker* que está disponible. Ahora sólo resta hacer público el *torrent file* para que las personas interesadas en el archivo puedan comenzar a bajarlo.

Con ese fin existen centenares de páginas WEB que publican los *torrents y* facilitan las búsquedas. Lo importante es que para que un *peer* pueda bajar el archivo necesita si o si un *torrent*, que el *tracker* este *on-line* y que exista algún *peer* conectado en ese momento que tenga el archivo o parte del archivo.
No es necesario que esté conectado el *seed* original ni algún otro *seed* para que se complete la descarga de un archivo ya que a medida que los *peers* tienen partes del archivo ya lo pueden ir compartiendo. Pero si la unión de todas las partes que tiene cada uno de los *peers* que están conectados no forma el archivo completo ninguno va a poder terminar de descargarlo, van a llegar a lo sumo a tener todos las mismas partes.
Ahora solo resta ver como un *peer* comienza con la descarga de un archivo. Cómo se mencionó anteriormente, tiene que disponer de un *torrent*. Como en el *torrent* está la información del *tracker* lo primero que hace es establecer una conexión con el éste informándole que está interesado en este archivo. El *tracker,* en base a información enviada por el candidato a *peer*, valida la autenticidad del pedido y en caso de ser auténtica lo convierte en *peer* y automáticamente le envía una lista con *peers* y/o *seeds* para que este nuevo *peer* se pueda comunicar directamente con ellos para comenzar a descargar el archivo.
En resumen, hasta aquí se describieron los pasos a seguir para compartir y distribuir un archivo por medio de BitTorrent. El protocolo [BTP1.0] especifica en detalle los mensajes existentes entre un *peer* y un *tracker* y los mensajes entre los *peers*.
Cómo caso de estudio para esta tesis se tomó una implementación en JAVA de un cliente de *BitTorrent* llamada ***Azureus***. Como el objetivo es generar una traza de la interacción entre los *peers*, la instrumentación genera eventos ante la ocurrencia de cualquier mensaje entre éstos. El protocolo define los siguientes tipos de mensajes que pueden enviarse entre *peers*:

BTHandshake	Es el primer mensaje que debe enviarse entre un peers.
BTBitfield	Solo puede enviarse inmediatamente luego de un *BTHandshake* e informa cuales son las partes del un archivo que un *peer* tiene.
BTChoke	Un *peer* envía *BTChoke* a otro indicándole que no le envíe ningún *BTRequest* por el momento porque no le va a responder.
BTUnchoke	Indica que el *peer* puede empezar a enviar *BTRequests* para pedir bloques del archivo.
BTInterested	Indica que en cuanto el peer que envía este mensaje reciba *BTUnchoke* empezará a enviar *BTRequests*.
BTUninterested	Indica que el *peer* no está interesado en ninguna de las partes que tiene el otro *peer*.
BTRequest	Pedido de transferencia de cierto bloque específico del archivo.
BTPiece	En respuesta a un *BTRequest*, este mensaje contiene un bloque de archivo a transferir.
BTCancel	Se utiliza para cancelar pedidos de *BTRequest*.

Estos mensajes son todos los necesarios que propone el protocolo para el intercambio de archivos entre los *peers*.

8.2 Instrumentación con Maal del Azureus [AZU]

Para la instrumentación se hizo un análisis en profundidad código de **Azureus** versión 2.3.0.2. La licencia de ***Azureus*** es Open Source y esto es una ventaja ya que es

necesario tener conocimiento y disponibilidad del código para poder instrumentar una aplicación con Maal. Se eligió esta versión porque era la última disponible en el momento en el que se hizo el análisis.
El análisis consistió por un lado en detectar dentro del código cuales son los puntos en los que se hacen los envíos y las recepciones de los mensajes del protocolo hacia y desde los otros *Peers* y por otro lado examinar de que manera se puede obtener la información necesaria para crear la instancia representativa del Maal del nodo que está enviando o recibiendo el mensaje en ese punto (*owner*) y la instancia representativa del nodo que envió o va a recibir el mensaje (*source* o *destination* respectivamente). Esto es para cumplir con los requerimientos del Modelo de mensajes de eventos.
Una vez detectados los puntos de envío y recepción de mensajes entre *Peer* y obtenida la información de cómo instanciar los nodos involucrados en el mensaje ya es hora de instrumentar la aplicación para que genere la traza global (RGLOB). Se tomó el IP que tiene asignada la PC que está corriendo la instancia del **Azureus** como identificador del nodo representativo para *Maal*, esto limita a una sola instancia de **Azureus** corriendo por PC.
En el proceso de instrumentar los mensajes entrantes se crearon 3 clases:
IncomingBTMessageOwnerRetriever: Es la clase responsable de obtener el IP local e instanciar el nodo *owner* del evento de recepción del mensaje.
IncomingBTMessageSourceRetriever: Es responsable de obtener el IP del nodo que envió el mensaje e instanciar el nodo representativo del *Peer* que envió el mensaje.
IncomingBTMessageSemanticsRetreiver: Tiene la responsabilidad de darle semántica al evento de recepción del mensaje. En este caso, indica el tipo de mensaje dentro del protocolo BitTorrent.
Para poder implementar estas clases fue necesario modificar una clase en el código del **Azureus** cambiando un método que era privado a público para poder accederlos desde estas clases. Esto fue lo **único** que se modificó del código de **Azureus** en toda la instrumentación cumpliendo muy bien con el requerimiento (RINV). En la Fig. 24 se muestra un ejemplo de la configuración de **Maal** para generar el evento de recepción del mensaje **BTHandshake** del protocolo eliminando los nombres de los paquetes en los nombres de las clases para que sea más claro a la lectura.

```
<receiveEvent name="ReceivedBTHandshake" package="org.gudy.azureus2.core3.peer.maal.event">
    <sourceRetreiver className="IncomingBTMessageSourceRetriever"/>
    <ownerRetreiver className=" IncomingBTMessageOwnerRetriever"/>
    <semanticsRetreiver className=" IncomingBTMessageSemanticsRetreiver"/>
    <pointcut pattern="call(private void
PEPeerTransportProtocol.decodeBTHandshake(BTHandshake))"/>
</receiveEvent>
```

Fig. 24 – Evento de Recepción de mensaje BTHandShake

Como se puede ver en la Fig. 24 en el *tag pointcut* se pone una referencia al momento en el código del *Azureus* en el que se recibe un mensaje de *BTHandShake*.
Para el caso de los mensajes salientes de igual manera que para los de recepción se implementaron 3 clases que sirven de soporte a **Maal** que son:
OutogoingBTMessageOwnerRetreiver: Es la clase responsable de instanciar el nodo *owner* del evento de envío de mensaje.
OutogoingBTMessageDestinationRetreiver: Es la responsable de instanciar el nodo representativo del *Peer* que va a recibir el mensaje.
OutgoingBTMessageSemanticsRetreiver: Tiene la responsabilidad de darle semántica al evento de envío de mensaje.

En la Fig. 25 se puede observar el extracto de la configuración de **Maal** para el mensaje de envío de un BTHandShake, en este caso no alcanza con especificar el método en el que se envía el mensaje ya que todos mensajes salen por el mismo método. Entonces fue necesario implementar un predicado llamado *SendBTHandshakePredicate* que solo es verdadero si el mensaje que se está enviando es un *BTHandshake* y falso en caso contrario.
En resumen, lo que dice este fragmento es que existe un evento llamado `SentBTHandshake`, que sucede cuando el método `public void OutgoingMessageQueue.addMessage(Message, boolean)` es invocado y cuando el predicado de la clase `SendBTHandshakePredicate` es verdadero.

```
<sendEvent name="SentBTHandshake" package="org.gudy.azureus2.core3.peer.maal.event">
    <destinationRetreiver className="OutogoingBTMessageDestinationRetreiver"/>
    <ownerRetreiver className="OutogoingBTMessageOwnerRetreiver"/>
    <semanticsRetreiver className="OutgoingBTMessageSemanticsRetreiver"/>
    <predicate className="SendBTHandshakePredicate"/>
    <pointcut pattern="call(public void OutgoingMessageQueue.addMessage(Message, boolean ))"/>
</sendEvent>
```

Fig. 25 - Evento de envío de mensaje BTHandShake

Análogamente al caso anterior se tuvo que implementar un predicado para cada evento de envío de mensaje que se instrumento.
Ya configurados cada uno de los eventos de envío de recepción de cada uno de los mensajes de protocolo hay que ensamblar cada envío con su correspondiente recepción para armar cada uno de los mensajes. En la Fig. 26 se muestra como queda ensamblado el mensaje de *BTHandShake* en el archivo de configuración de **Maal**.

```
<message>
    <sendEvent name="SentBTHandshake" package="org.gudy.azureus2.core3.peer.maal.event">
       <destinationRetreiver className="OutogoingBTMessageDestinationRetreiver"/>
       <ownerRetreiver className="OutogoingBTMessageOwnerRetreiver"/>
       <semanticsRetreiver className="OutgoingBTMessageSemanticsRetreiver"/>
       <predicate className="SendBTHandshakePredicate"/>
       <pointcut pattern="call(public void OutgoingMessageQueue.addMessage(Message, boolean ))"/>
    </sendEvent>
    <receiveEvent name="ReceivedBTHandshake" package="org.gudy.azureus2.core3.peer.maal.event">
       <sourceRetreiver className="IncomingBTMessageSourceRetriever"/>
       <ownerRetreiver className=" IncomingBTMessageOwnerRetriever"/>
       <semanticsRetreiver className=" IncomingBTMessageSemanticsRetreiver"/>
       <pointcut pattern="call(private void
PEPeerTransportProtocol.decodeBTHandshake(BTHandshake))"/>
    </receiveEvent>
</message>
```

Fig. 26 - Mensaje BTHandShake

Una vez terminado el archivo de configuración de **Maal** definiendo todos los eventos del protocolo e implementadas las clases de soporte, se instrumentó el código de ***Azureus*** con una tarea de *Ant* que provee la prueba de concepto de **Maal** y luego se compiló todo junto utilizando el compilador de *AspectJ.*
Con todos estos pasos se logró una nueva versión del *Azureus* que tiene la misma funcionalidad que antes con el agregado de que genera los eventos de envío y recepción de los mensajes del protocolo.
Para hacer las pruebas se generaron varios *Torrents*, se instaló el Azureus instrumentado en varias máquinas, se dispuso una PC cómo *traceServer* y otra PC como *Tracker* y se comenzó a compartir archivos entre las distintas instancias de **Azureus** colectando con éxito en el *TraceServer* una traza global respetando el orden causal con todos los mensajes enviados entre los clientes.

8.3 Resultados

Se obtuvieron varios resultados muy satisfactorios con este caso de estudio. Se logró que a partir de una aplicación distribuida real que no colecta eventos en una traza global, lo haga instrumentando la misma con **Maal**. Además sólo hizo falta una mínima modificación en el código original (se hizo público un método que era privado).

La traza cumple con casi todos los requerimientos planteados en el enunciado del problema de la tesis. Sólo se pudo observar, analizando la traza generada (Fig. 27), que el requerimiento de *timestamping* no se cumple en su totalidad. A partir del error de sincronía de los nodos por NTP se puede ver que existen eventos de recepción de mensajes cuyo *timestamp* es anterior que el del evento de envío correspondiente. Estos pares de eventos están resaltados en la traza.

Sin embargo, como conclusión general, se puso a prueba la implementación de la técnica en un caso no ficticio y con un protocolo propietario. El resultado positivo es una prueba de la robustez tanto de la idea como de la prueba de concepto.

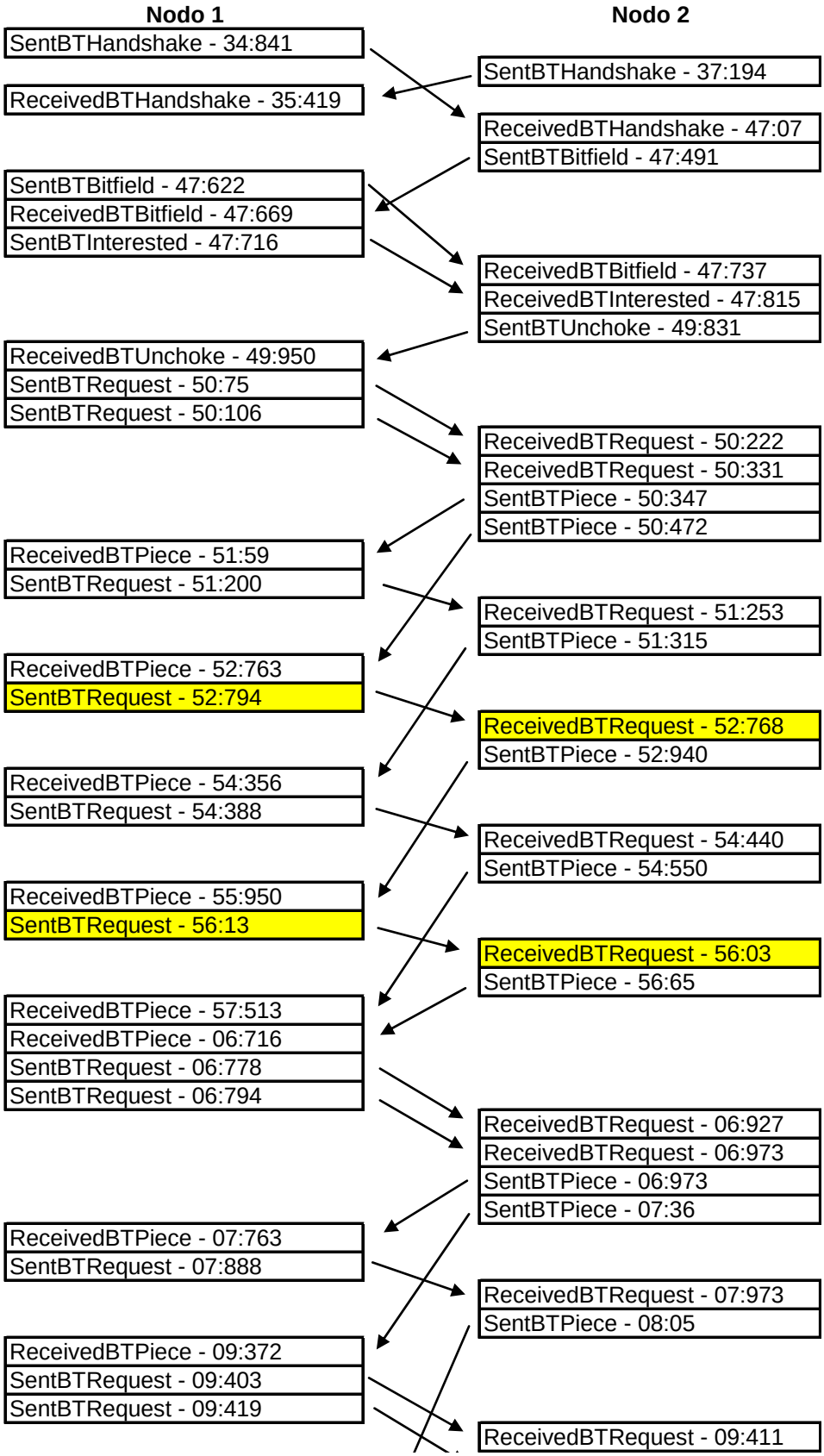

Fig. 27 - Parte de traza global Azureus

9 Conclusiones y aportes

A lo largo de esta tesis se definieron un conjunto de objetivos que apuntan a la creación de una técnica para agregar a sistemas distribuidos de tiempo real la capacidad de generar una traza causalmente consistente con un impacto mínimo. Con este fin, se diseñó un lenguaje declarativo para especificar cuales son los eventos que deben registrarse, un mecanismo para entretejer el sistema de generación de eventos basado en esa especificación dentro de la aplicación, un sistema de envío de información de eventos para cada nodo, un algoritmo de ordenamiento que garantiza la consistencia causal de la traza y un *framework* para desarrollar diferentes componentes para procesar la traza. También se desarrollaron procesadores de este tipo para visualizar la traza en forma gráfica o registrarla en distintos formatos.
El algoritmo propuesto se demostró rigurosamente. Además, se construyó una prueba de concepto de la técnica y se utilizó en dos casos de estudio.
El primero un caso de estudio de laboratorio, el clásico problema de los trenes y las barreras, en el que se logró la instrumentación de los eventos con un mínimo esfuerzo. Cómo quedó la duda si realmente este sistema anterior fue instrumentado con facilidad porque fue construido especialmente para este fin o si realmente este trabajo facilitaba la generación de trazas globales respetando el orden causal, se decidió seleccionar un programa real ya existente. El seleccionado fue la implementación más importante y más popular en Java del protocolo *BitTorrent*, ésta aplicación se llama *Azureus* tiene 1920 clases y solo el código ocupa 12.8 MB. La instrumentación de esta aplicación tuvo 3 desafíos importantes el primero entender el protocolo *BitTorrent*, el segundo entender la implementación y por último, y el más importante, instrumentarlo de ser posible para que genere eventos y los inserte en el *TraceServer*. La conclusión fue que si se pudo instrumentar sin problemas y solamente fueron necesarias crear 17 nuevas clases de soporte.
A continuación se analizan distintas cuestiones que surgen del resultado general del trabajo.
El hecho de contar con un lenguaje declarativo y basado en XML provee un mecanismo fácil e intuitivo de leer y escribir para especificar la forma de generar eventos de una aplicación.
Además, MAAL permite adaptarse a cualquier sistema de *middleware* de forma transparente siempre que éste cumpla con las precondiciones del algoritmo de ordenamiento de eventos. Esto se debe a que no requiere el envío de información adicional insertada en los mensajes propios de la DAUT y por ende no requiere modificaciones como lo hacen otros sistemas similares, por ejemplo los basados en los algoritmos de Fidge y Lamport. Otra consecuencia de esto es que es posible separar las redes de envío de mensajes de la DAUT y la de manejo de información de eventos y de esta forma no afectar en absoluto el trafico de mensajes original.
Dos ideas muy poderosas dentro de la ingeniería de software son: prescindencia (*obliviousness*) y cuantificación (*quantification*). La prescindencia esta relacionada con que el equipo de desarrollo que esta encargado de implementar la funcionalidad principal no debería tener la necesidad de saber de otras dimensiones que pueden afectar su programa. La cuantificación permite definir en que puntos de la aplicación se incorporaran los distintos aspectos sin tener la necesidad de numerarlos uno por uno.
Se buscó que la instrumentación de MAAL apunte estos objetivos en cuanto a la generación de trazas y se buscaron varias alternativas para cumplirlos. En principio se

intentó que la declaración del modelo de eventos no fuera dependiente del lenguaje de programación. Sin embargo, en función de conseguir el concepto de cuantificación, es necesario poder hablar de puntos de corte dentro del programa. Para lograr esto sin importar la sintaxis propia del lenguaje, debería desarrollarse un metalenguaje capaz de predicar sobre estos puntos de forma genérica [SETPOINT]. Como éste no era el objetivo de la tesis, fue necesario acoplarse a un lenguaje en particular y declarar los puntos de corte en base a su sintaxis.

Como AOP permite lograr la prescindencia y la cuantificación en cualquiera de las dimensiones del software, se decidió utilizar éste mecanismo como soporte de la implementación.

Por otro lado, el utilizar AOP facilitó el entretejido del código que genera los eventos en la DAUT.

Este trabajo permite al programador de la DAUT no tener que pensar en el aspecto de la generación de trazas. Sin embargo, sería bueno que el equipo de desarrollo esté enterado de que se está utilizando MAAL para este aspecto y no se desarrolle por equivocación o desconocimiento una solución ad-hoc en caso de que se necesite una traza con alguna de las características que provee esta técnica. Es decir, si bien es importante la idea de prescindencia, no debe ser llevada al extremo.

Se diseño el *framework* para que sea poco intrusivo. Es decir, que sea prácticamente innecesario modificar el código de la DAUT para obtener la generación de trazas. Por ejemplo, en el caso de estudio de Azureus, solo se tuvo que cambiar la visibilidad de un método privado a público en el código ya existente.

Uno de los aportes principales de esta tesis es que reduce el costo de implementación de una traza global que respete el orden causal ya que solo es necesario definir en que punto de la DAUT se generarán los eventos que formaran parte de la misma. MAAL entrega los eventos de la traza de tal forma que permite el uso de los mismos de la forma que el usuario quiera. Por ejemplo, se pueden visualizar gráficamente o registrarse en cualquier formato requerido.

Un punto importante para resaltar es el problema de la sincronización de los relojes por medio de NTP. Existía el riesgo de que el uso de este protocolo pudiera traer inconsistencias en la traza. Como se pudo ver en el caso de estudio de Azureus, este riesgo se manifestó. No se encontró una solución simple, hay una amplia gama de estrategias que se pueden investigar para atacar este problema y se detallan en la sección de Trabajos Futuros.

10 Trabajos futuros

Puede ser muy útil e interesante extender la funcionalidad de la herramienta para poder depurar una aplicación distribuida desde un monitor central. Con capacidad para introducir *breakpoints* en los nodos y ejecutar el sistema paso a paso.
Otra alternativa es implementar la idea de esta tesis en otros lenguajes o generar un metalenguaje que sirva para todos los lenguajes.
Con el tema se *timestimping* y acotamiento del error se pueden analizar varias alternativas. Puede ser interesante un *approach* del manejo de la inconsistencia. Por ejemplo, trabajar con trazas con *timestamp* simbólicos. Esto es timestamp que indica rango de distancias temporales entre eventos. Si se toma una traza y se generan toda la información de distancias temporales. Como la traza está generada con timestamp generado por NTP pueden existir inconsistencias, como se puedo ver en las conclusiones. La idea es relajar la traza simbólica o eliminando restricciones o ampliando rangos. Dada una inconsistencia existen varias soluciones posibles y la elección de la misma es un problema abierto muy interesante para estudiar en el futuro. Se puede estudiar la relación que existe entre este problema con el Cambio de Creencias.

11 Agredecimientos

Agradecemos etérnamente a Marti y Tani que nos acompañaron a lo largo de toda la tesis.
También queremos agradecer a cada una de las personas que nos sirvieron de pilares para poder cumplir con este desafío que nos planteamos.
A nuestros familiares y amigos que abandonamos por tanto tiempo.
Por parte de Matías; Sonia (Mamá), Maria (Abuela), Leni, Anyi, Martín, Oscar, Sasha, Misha, Xenia, Dyna.
Por parte de Alejandro; Daniel y Cristina (papás), Adri (Hermano), Fabi, Mir, Eze, Dany, Javite y Esteban (Amigos).
Por parte de los dos; Victor, Bauna (cvs admin), Sebastián, Data Transfer por brindarnos los recursos, Ine y Ariel Futo.

12 Referencias

[LAM78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[FID91] Colin Fidge. Logical time in distributed computing systems. *Computer*, 24(8):29–33, August 1991.

[CHS97] Mikkel Christiansen Jesper Langfeldt Hagen Kristian Qvistgaard Skov. TraceInvader—Atool for debugging distributed applications. Master's Thesis, Aalborg University Departament of Computer Science, Fredrik Bajers Vej 7E, DK-9220 Aalborg Ost, Denmark, January 1997

[ROV85] D. Drusinsky. The Temporal Rover and the ATG Rover. Springer-Verlag Lecture Notes in Computer Science, 1985, p. 323-329.

[MAC99] I. Lee y , S. Kannan, M. Kim, O. Sokolsky, and M. Viswanathan. Runtime Assurance Based On Formal Specifications. Department of Computer and Information Science, University of Pennsylvania Philadelphia, PA 19104 March 23, 1999
http://www.cis.upenn.edu/~rtg/mac/

[MSS97] M. Mansouri-Samani and M. Sloman. GEM: A generalized event monitoring language for distributed systems. IEE/IOP/BCS Distributed Systems Engineering Journal, 4(2):96-108, June 1997.

[SBFO02] Fernando OLIVETO, Sebastián BLAUSTEIN. Monitoreo de Sistemas Temporizados especificados con Grafos de Diagramas de Secuencia (MSC-Graphs). Facultad de Ciencias Exactas y Naturales Universidad de Buenos Aires, Diciembre 2002

[JJJR94] C. Jard, T. Jéron, G. Jourdan, J. Rampon. A General Approach to Trace-Checking in Distributed Computing System. Institut National de recherché en informatique et automatique, ISSN-0249-6399, Mars 1994

[MILLS91] David L. Mills. Internet Time Synchronization: the Network Time Protocol. Electrical Engineering Department University of Delaware. 1991

[PF98] *Model Checking Java Programs Using Java PathFinder (K. Havelund, T. Pressburger)* (International Journal on Software Tools for Technology Transfer, STTT, 2(4) April 2000. Special issue containing selected submissions for the 4'th SPIN workshop, Paris, November, 1998

[TTAU] Telelogic Tau. http://www.telelogic.com/products/tau/index.cfm

[W3C] The World Wide Web Consortium - http://www.w3c.org

[CNPD99] Computer Networks: A Systems Approach, Second Edition, Larry L. Peterson and Bruce S. Davie.The Morgan Kaufmann Series in Networking, David Clark, Series Editor, October 1999

[CNTAN96] Computer Networks, Andrew S. Tanenbaum. Tercera edición, Prentice Hall, 1996.

[ASPECTJ] AspectJ Project - http://eclipse.org/aspectj

[Object-Z] The Object-Z Specification Language: Version 1, Roger Duke and Paul King and Gordon Rose and Graeme Smith. University of Queensland, AUSTRALIA. April 1991

[BTP1.0] BitTorrent Specification: http://wiki.theory.org/BitTorrentSpecification
BitTorrent Documentation: http://www.bittorrent.com/protocol.html

[AZU] Página principal: http://azureus.sourceforge.net/
Documentación: http://azureus.aelitis.com/wiki/index.php/Main_Page

[ANT] Página principal: http://ant.apache.org/
Documentación: http://ant.apache.org/manual/index.html

[NTP] The NTP Public Services Project - http://ntp.isc.org

[NTPWP] Wikipedia - Network Time Protocol –
http://en.wikipedia.org/wiki/Network_Time_Protocol

[SETPOINT] Rubén Altman - Alan Cyment. SetPoint: Un enfoque semántico para la resolución de pointcuts en AOP. Facultad de Ciencias Exactas y Naturales Universidad de Buenos Aires, Noviembre 2004