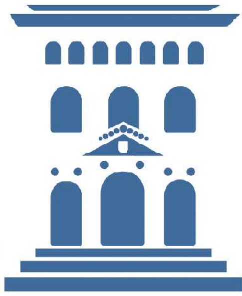


Universidad de Zaragoza
Centro Politécnico Superior

WS-PTRLinda: un sistema de coordinación temporizado y persistente basado en tecnologías de servicios Web

Ingeniería Informática
Departamento de Informática e Ingeniería de Sistemas

Proyecto Fin de Carrera

Autor: Juan Mengual Lombar
Director: Francisco Javier Fabra Caro

Noviembre 2010

Como nunca sabes si habrá más publicaciones, este es un buen momento para una dedicatoria.

A mamá, papá y sis porque no es posible pedir más de ellos.
A Belén y Noe, porque son unas amigas informáticas increíbles.
A mis amigos, que no pueden ser mejores.
A Alberto, que podemos estar meses sin hablar y seguimos como siempre.
A Nacho, porque soportarme durante 25 años no es fácil.
A Santi, Ángel y otros de Boulevard, porque hemos vivido grandes momentos juntos, y los que quedan.
A David, porque pese a ser murciano es uno de mis mejores amigos.
A Alba y Borja, por todas esas horas en Torino.
A mi abuelo, porque estaré satisfecho si consigo llegar a ser la mitad de querido que él.
A Bobes, por todo y porque estoy aquí por culpa de esa Atari.
Y a Pedro, por acompañarme durante todo el camino haciendo funciones de secretario, chófer, confidente y sobretodo amigo. Porque no te vuelvas a marchar a Finlandia y porque al final lo hemos conseguido.

WS-PTRLinda: un sistema de coordinación temporizado y persistente basado en tecnologías de servicios Web

RESUMEN

Hoy en día los procesos Web son actores habituales de internet que siguen aumentando en número y complejidad. La evolución de las comunicaciones en cuanto a rapidez y seguridad ha permitido el nacimiento de una nube donde herramientas, servicios y espacio se ofrecen virtualizados. Los procesos que antes funcionaban localmente ahora se enfrentan a un mundo online donde deben comunicarse y coordinarse entre ellos, ahora son procesos Web.

Los procesos Web involucran múltiples usuarios que resultan complejos de coordinar y comunicar, para ello surgen middlewares de coordinación. Uno de los elementos de este tipo de middlewares es el bróker de mensajes, que debe coordinar a los servicios Web que lo usan para interaccionar. Actúa como un repositorio de mensajes y datos al cual todos los servicios implicados pueden acceder.

En 2006 el Grupo de Integración de Sistemas Distribuidos y Heterogéneos (GIDHE) de la Universidad de Zaragoza hizo una propuesta de broker de mensajes, RLinda, basado en Linda y diseñado e implementado mediante tecnología de Redes de Petri de alto nivel. La plataforma actual tiene ciertas limitaciones que dificultan su integración en entornos donde las tecnologías y estándares de servicios Web son utilizados.

WS-PTRLinda se construye sobre el núcleo de RLinda, implementando las principales funcionalidades con la misma tecnología que el resto del sistema: las redes por Referencia, una subclase de las Object Petri Nets. El nuevo sistema se compone de dos nuevas capas, una de persistencia y otra de temporización.

La capa de persistencia aporta un sistema de almacenamiento persistente de los datos permitiendo al sistema ser aplicado en entornos con características de alta disponibilidad. Esta capa se basa en el empleo de la herramienta Hibernate para comunicase con una base de datos MySQL, ofreciendo dos conceptos de persistencia que proporcionan mayor flexibilidad para enfrentarse a un escenario concreto.

La capa de temporización aplica el concepto el tiempo a tanto a las operaciones como a los datos. Esta capa descansa sobre una serie de modificaciones en la red de Petri que modela el sistema y algunos objetos que funcionan en *background* ocupados de gestionar el tiempo que una operación o dato es válida/o. El concepto de tiempo añade nuevas posibilidades a las anteriores operaciones del sistema y lo habilita para desplegarse en ciertos campos de aplicación.

El proyecto añade además una interfaz accesible mediante estándares de servicios web (SOAP y REST) que conecta directamente con el núcleo de la aplicación, evitando el cuello de botella que la tecnología RMI ocasionaba en el RLinda original y validación de datos en base a esquemas XMLSchema.

Finalmente se desarrolla una aplicación de descarga P2P como caso real de aplicación y se analizan las prestaciones del sistema desarrollado en un clúster, analizando los resultados con respecto a los obtenidos inicialmente para RLinda.

Índice General

Capítulo 1 – Introducción …………………………………………………………………….….. 5

Capítulo 2 – Trabajo Relacionado ………………………………………………………...… 7
2.1 El modelo de coordinación Linda ………………………………………………...…... 7
2.2 RLinda, la implementación de la universidad de Zaragoza …………………………….... 8
2.2.1 Ejemplo de cómo se ejecuta cada operación
2.2.2 La función de búsqueda y asignación o matching
2.3 Distributed RLinda ………………………………………………………………...……..11
2.4 RLinda WS-Security ………………………………..…………………….……….…11
2.5 Otras soluciones basadas en Linda ……………………………………..…………..12
2.5.1 JavaSpaces
2.5.2 GigaSpaces
2.5.3 TSpaces
2.5.4 Objecive Linda
2.5.4 ELLIS
2.5.5 XMLSpaces
2.5.6 WorkSpaces
2.6 Casos de aplicación del modelo de coordinación Linda ……………………….…......14
2.6.1 Espacio de tuplas para redes sociales en teléfono móviles
2.6.2 Coordinación en sistemas multiagente heterogéneos
2.6.3 Cómo construir un ComputerFarm
2.6.4 Plataforma basada en un espacio de tuplas para aplicaciones móviles adaptativas
2.6.5 Computación basada en espacios de tuplas para la Web Semántica

Capítulo 3 – Objetivos ………………………………………………………...……....17

Capítulo 4 – Diseño de WS-PTRLinda…………………………………………....…...…18
4.1 Fase uno: RLinda Modificado (WS-RLinda ………………………..…..…………..18
4.2 Fase dos: Persistent RLinda ………………………………………..…....……........19
4.3 Fase tres: Temporized RLinda…………………………………….………......…....20
4.4 El sistema completo: WS-PTRLinda…………………………….…………...…21

Capítulo 5 – Implementación de WS-PTRLinda ……………………………………...…22
5.1 *Persistent* RLinda ……………………………………………………...………...…22
5.1.1 Funcionalidades añadidas
5.1.2 Tecnologías utilizadas
5.1.2 ¿Qué datos deben ser persistidos?
5.1.2 Organización de Persistent RLinda como servicio Web
5.1.3 Arquitectura del sistema
5.1.4 Tipos de persistencia ofrecidos
5.1.5 Funciones Load/Save
5.2 *Temporized* RLinda ……………………………………………………………….…...30
5.2.1 Funcionalidades añadidas
5.2.2 Organización de TRLinda como servicio Web
5.2.3 Arquitectura del sistema
5.2.4 Aplicando la temporización a: las tuplas
5.2.5 Aplicando temporización a: las operaciones
5.2.6 Traza de interacción SleepWalker – Sleeper

5.3 WS-Persistent Temporized RLinda 36

Capítulo 6 – Simulación, evaluación y resultados..........37

Capítulo 7 – Caso de uso: una aplicación de descarga P2P 40
7.1 un nuevo cliente P2p: FLARLARdownloader 40
7.2 Modelo de coordinación 40
7.3 Descarga de tuplas 41
7.4 GUI (Interfaz gráfica) 42

Capítulo 8 – Conclusiones..........43
8.1 Conclusiones a nivel técnico 43
8.2 Trabajo futuro 44
8.3 Conclusiones personales..........44

Capítulo 9 – Referencias 46

ANEXO 1 - Tecnologías Utilizadas 49

ANEXO 2– Diagrama de esfuerzos..........53

ANEXO 3– Manual de usuario de WS-PTRLinda 57

ANEXO 4– Modelo de desarrollo de software 73

ANEXO 5– Limitaciones y problemas encontrados..........76

ANEXO 6– Implementación de WS-PTRLinda extendida 80

ANEXO 7– Glosario de términos 117

Capítulo 1 | **Introducción**

Hoy en día los procesos Web son actores habituales de internet que siguen aumentando en número y complejidad. Las empresas dejan de fabricar sus propias herramientas para confiar en las que se ofrecen de forma on-line y lo mismo sucede con su espacio de almacenamiento local. La evolución de las comunicaciones en cuanto a rapidez y seguridad ha permitido el nacimiento de una nube donde herramientas, servicios y espacio se ofrecen virtualizados. La nube ya no es un concepto solo para las personas relacionadas con el mundo de la informática sino que ha dado el salto de los centros de investigación a la administración pública, el mundo empresarial y poco a poco al usuario base de internet. Los procesos que antes funcionaban localmente ahora se enfrentan a un mundo online donde deben comunicarse y coordinarse entre ellos, ahora son procesos Web.

Los procesos Web involucran múltiples usuarios que resultan complejos de coordinar y comunicar. Esto se debe a que están compuestos por otros procesos más sencillos que pueden estar ejecutándose en máquinas ubicadas en diferentes puntos de la red, y a que estos procesos sencillos deben ejecutarse en un orden y una coordinación concreta, comunicándose en todo momento entre ellos y enviándose tanto mensajes como datos. Normalmente utilizan la tecnología de los servicios Web.

Un proceso Web requiere de una coordinación y definición de las coreografías, relaciones, lógica de negocio propia del proceso Web e interacciones necesarias entre los servicios Web más sencillos. Como todo esto resulta complejo nacen los sistemas *middleware* para procesos Web, que coordinan todas estas tareas y envuelven la lógica de negocio. Estos sistemas se sitúan en el centro de la comunicación, se acceden también como servicio Web y se encargan de comunicar y coordinar a los servicios que acceden a ellos. Estos sistemas suelen estar compuestos por tres módulos:

- Un componente de composición también conocido como motor de *workflows*, en el que se define y ejecuta la lógica de negocio del servicio Web complejo.

- Una componente de conversación que gestiona las conversaciones entre los procesos involucrados en un proceso de negocio que ya ha empezado y realiza las invocaciones pertinentes a los servicios Web del exterior.

- Un *broker* de mensajes para encaminar los mensajes que se intercambian los participantes. Este módulo separa la lógica del intercambio de mensajes de los protocolos de comunicación que se emplean para transmitir y recibir mensajes. Es en este *broker* de mensajes donde se desarrolla este PFC.

El *broker* de mensajes debe coordinar a los servicios Web que lo usan para interaccionar. Actúa como un repositorio de mensajes y datos al cual todos los servicios implicados pueden acceder. Dicho de otra forma, el *broker* de mensajes realiza la coordinación definida y dirigida tanto por el motor de *workflows* como por el componente de conversación. Podemos definir este *broker* como un sistema de coordinación.

En 2006 el Grupo de Integración de Sistemas Distribuidos y Heterogéneos (GIDHE) de la Universidad de Zaragoza hizo una propuesta de *broker* de mensajes para este tipo de middleware basado en el lenguaje de coordinación Linda. Este modelo de coordinación nació hace años en el ámbito de la programación concurrente y emplea la comunicación generativa como una manera de coordinar varios procesos que se ejecutan al mismo tiempo y que deben enviarse mensajes y datos entre sí. Este modelo resulta muy útil hoy en día aplicado a los procesos Web que se ejecutan

simultáneamente en diferentes puntos de la red y que deben coordinarse y comunicarse para alcanzar sus objetivos con éxito.

Este nuevo enfoque se llama RLinda y en su versión original ofrece acceso al mismo a través del protocolo RMI y como servicio Web mediante el protocolo SOAP. Varios proyectos de fin de carrera han extendido la versión inicial, aportándole memoria distribuida (Distributed RLinda) o estándares de seguridad para garantizar las transacciones y el intercambio de mensajes y datos (WS-Security RLinda). Este PFC añade dos nuevas capas al sistema, persistencia y temporización.

Esta primera capa de persistencia se sitúa en un contexto de amplia concurrencia de servicios Web accediendo al sistema para comunicarse y coordinarse. Cada uno es parte de su proceso Web y tiene sus propios objetivos pero se necesitan unos a otros para progresar. Esta capa se apoya en la tecnología de Hibernate que se comunica con una base de datos MySQL para ofrecer dos posibilidades distintas de tolerancia a fallos para que, si algo sucede, el sistema pueda reiniciarse y los procesos puedan continuar desde el punto de progreso en el que estaban.

La capa de temporización por otro lado añade el concepto de tiempo a las operaciones y los datos, permitiendo que solo sean válidos en el sistema durante el tiempo que el proceso Web considere oportuno. Se consigue así un sistema en el que los datos son válidos durante el tiempo que establezca el emisor, que se descartan de manera automática sin necesidad de que deba retirarlos.

La versión original de RLinda utiliza el protocolo SOAP para la comunicación como servicio Web. Este protocolo se está quedando obsoleto, una muestra es que los grandes proveedores de la web 2.0 como Google, Yahoo o Facebook han migrado a la tecnología REST. Por ello en este PFC añade el protocolo REST a las formas de acceder a RLinda.

La nueva versión de RLinda recibe el nombre de WS-PTRLinda (Persistent Temporized RLinda).

Esta memoria del proyecto fin de carrera se organiza del siguiente modo:

- En el capítulo siguiente se da a conocer el modelo de coordinación Linda, la base de todo. Se verá la implementación de la Universidad de Zaragoza, RLinda, otros PFC sobre esta, se nombrarán otras implementaciones existentes en el mercado y algunos ejemplos de uso de sistemas similares en la actualidad.
- En el capítulo tres se enumeran los objetivos a realizar de este PFC.
- En los capítulos cuarto y quinto se presenta el diseño de WS-PTRLinda y su implementación.
- Los capítulos sexto y séptimo presentan la evaluación de los resultados obtenidos en las simulaciones realizadas y un caso práctico de una aplicación P2P que utiliza WS-PTRLinda para sincronizar sus clientes.
- En el último capítulo se presentan las conclusiones finales del proyecto.
- Una serie de anexos se incluyen presentado las tecnologías utilizadas, el manual de usuario, el modelo de desarrollo de software, un diagrama de esfuerzos, la aplicación de prueba en detalle, limitaciones y problemas encontrados, glosario de términos, referencias empleadas y una versión extendida del capítulo de implementación.

Capítulo 2 | **Trabajo Relacionado**

En este capítulo veremos en primer lugar el modelo de coordinación Linda, propuesto en 1992 para coordinar procesos que se ejecutan en paralelo y que da origen a la implementación desarrollada en el grupo GIDHE, RLinda, dentro de la Universidad de Zaragoza. La visión de Linda proporcionará al lector una base de en qué consiste el modelo de coordinación necesaria para un seguimiento adecuado de esta memoria. Por otro lado se verá la implementación de RLinda y su funcionamiento sin profundizar en exceso para que el lector adquiera una idea de cómo funciona este sistema de coordinación y qué sucede desde que el cliente ejecuta un método hasta que obtiene un resultado. Hablaremos también de las mejoras introducidas por otros PFC de esta universidad, de otras alternativas comerciales a nuestra implementación de Linda y finalmente se expondrán casos de uso de espacios de tuplas.

2.1 El modelo de coordinación Linda

El modelo de coordinación Linda fue propuesto originalmente por Carriero y Gelernter [1,2]. Linda es un lenguaje de coordinación basado en la comunicación generativa sobre un espacio compartido. Esto quiere decir que la comunicación se realiza mediante el consumo de estructuras de datos pasivas (una vez en el espacio compartido permanecen invariables), estas estructuras se representan como tuplas y por ello el espacio compartido recibe el nombre de espacio de tuplas (*tupleSpace*). Podemos definir el espacio de tuplas como un repositorio de tuplas, que pueden ser de longitud variable, al que se accede mediante el uso de tres operaciones o directivas de comunicación que son atómicas:

- Una operación *out* que escribe tuplas dentro del espacio.
- Una operación *in* que realiza la lectura destructiva de tuplas del espacio compartido.
- Una operación *rd* que realiza una operación de lectura no destructiva.

Las operaciones de lectura pueden recibir como parámetro un patrón (*pattern*) o plantilla de manera que la tupla retirada debe concordar con él, quedando bloqueado el proceso que ejecutó la operación hasta que una tupla es devuelta. Si ninguna tupla se ajusta al patrón dado, permanecerá bloqueado.
Una tupla es similar a una lista en lenguaje Lisp, sus elementos pueden ser de dos tipos, átomos o tuplas (se asume que la tupla vacía no existe). Un patrón no es más que una tupla extendida. Esta extensión consiste en que un átomo puede ser un elemento comodín '*' o *wildcard* que equivale a un elemento cualquiera a la hora de ver si una tupla se ajusta a un patrón dado.

Un conjunto de procesos emplean el espacio de tuplas para comunicarse e interactuar unos con otros mediante la inserción y la extracción de tuplas. Un proceso que desee insertar una tupla deberá ejecutar una operación *out(t)*, que escribe *t* en el espacio de tuplas y no es bloqueante. Un proceso que desee retirar una tupla del espacio compartido debe ejecutar una operación *in(pattern)*, que devuelve una tupla que ajuste al patrón *pattern* o bloquea al proceso si no encuentra ninguna. La forma que Linda tiene de comprobar si una tupla se ajusta a un patrón recibe el nombre de *función de matching* o de búsqueda y asignación.
Existen cuatro funciones diferentes de búsqueda y asignación: búsqueda y asignación fuerte, búsqueda y asignación débil, búsqueda y asignación de atributos y búsqueda y asignación general de atributos. Las tres primeras son implementadas en este PFC y explicadas en el punto 2.2.2.

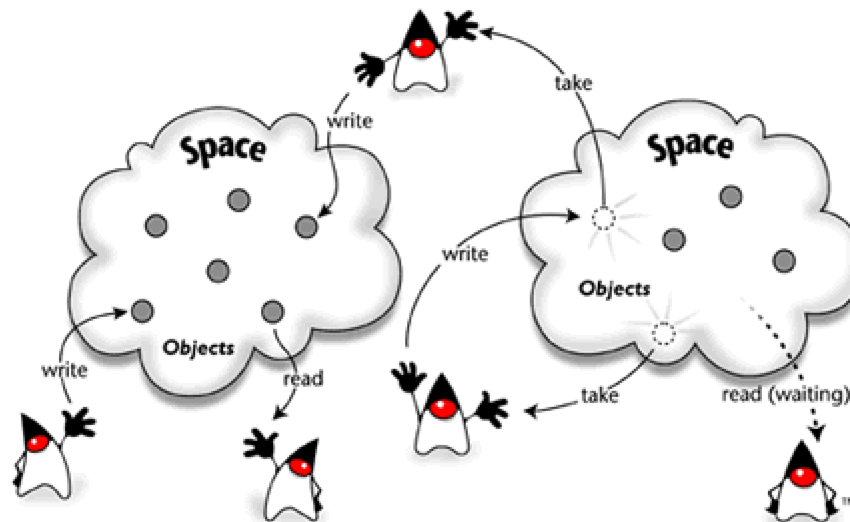


Fig. 1 Interacción con un espacio de tuplas

2.2 RLinda, la implementación de la universidad de Zaragoza

Las redes de Petri pueden ayudar proporcionando una forma sencilla de modelar sistemas distribuidos y concurrentes así como la coordinación y la comunicación entre procesos. El Grupo de Integración de Sistemas Distribuidos y Heterogéneos (GIDHE) de la Universidad de Zaragoza, creó en 2006 una primera implementación del modelo de coordinación Linda basada en las Reference Nets [4]. Posteriormente, en 2007, se evolucionó la implementación centralizada de RLinda[3] hacia su correspondiente implementación distribuida, llamada DRLinda [16].
La elección de Renew para la implementación se debe a, por un lado, que Renew implementa de forma muy eficiente y flexible aspectos de concurrencia (importante para implementar un espacio de coordinación Linda), y a que usa un concepto de tupla que permite implementar de forma fácil y eficiente el concepto de tupla de Linda. Además el lenguaje de inscripción de Renew incluye las tuplas como tipos de datos.

Volviendo con RLinda, las tuplas se codifican en lenguaje XML para la comunicación con el exterior ya que en los últimos años este lenguaje ha destacado como forma de codificación para el intercambio de datos en internet. Un cliente que desee usar esta implementación de Linda puede acceder a ella mediante la invocación de una operación *out*, *in* o *rd* a través de RMI o SOAP, pasando una tupla descrita en formato XML como parámetro. Por ejemplo, una operación de lectura *in(xml_pattern)* devolverá una tupla expresada en formato XML.

La figura 2 muestra la red de Petri (coordinador de RLinda) que implementa el servidor Linda y la interfaz externa. El lugar donde se almacenan las tuplas y de donde se extraen, es decir el repositorio de tuplas, es *tupleSpace*. Veamos cómo funciona esta red de Petri.
Desde el punto de vista del cliente, todas las operaciones son invocadas como operaciones únicas. Las operaciones de lectura *rd* e *in* son funciones con un patrón como parámetro de entrada que devuelven un tupla resultado. La operación de escritura *out* en cambio es un procedimiento con una tupla de parámetro de entrada que es escrita en *tupleSpace*.

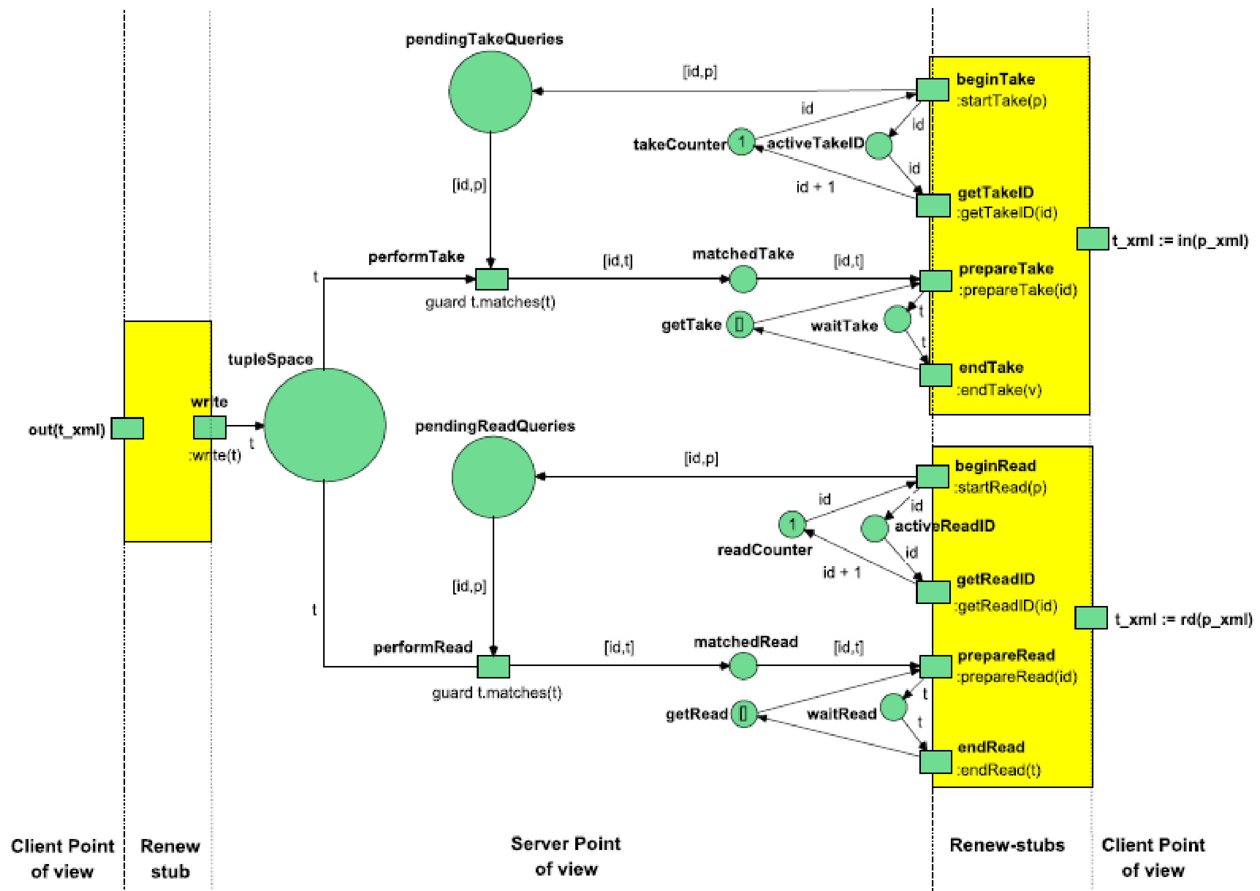


Fig. 2 Coordinador de RLinda

Podemos observar que las operaciones de lectura se componen de varias transiciones en la red. Renew emplea *Renew-stubs* para esconder todas estas transiciones bajo una única operación que el servidor publica. Estos *stubs* son interfaces de lenguaje descriptivo de alto nivel que permite encapsular una secuencia de disparos de transiciones. Mediante estos *stubs* se genera código Java que puede ser referenciado como cualquier clase normal. Renew permite el paso de parámetros entre este código generado mediante el *stub* y la red de Petri.
Ejemplo de código de un *stub*:

```
int startTake(Tuple p) {this:startTake(p); this:getTakeID(return); }
Tuple endTake(int id) { this:prepareTake(id); this:endTake(return); }
```

2.2.1 Ejemplo de cómo se ejecuta una operación

Veamos cómo se comporta la red a través de la traza de una operación de lectura *in(p_xml)*. Desde un punto de vista conceptual la operación se divide en dos partes, una primera en que el patrón es insertado en el sistema y una segunda en que una tupla es devuelta como resultado. Cada una de estas partes se divide a su vez dos pasos que se relacionan mediante un identificador de operación *id*.

El patrón se inserta en el sistema (*this:startTake(p)*), devolviendo un identificador (*this:getTakeID(return)*) asociado al patrón.
Este identificador es pasado (*this:prepareTake(id)*) para obtener la tupla correspondiente a ese *id* que estará esperando en el lugar *matchedTake*.

Dentro de la red las tuplas se implementan con el tipo *Tuple*, pero el cliente escribe y lee tuplas en formato XML. La operación de transformación de un tipo a otro se hace en el *stub* de Renew. El código final ejecutado es el siguiente:

```
XML_Tuple in(Tuple p_xml) { return
parser.Tuple2XML(endTake(startTake(parser.XML2Tuple(p_xml)))); }
```

Queda por comentar la parte correspondiente a la transición *performTake* de la figura, responsable de aplicar la *función de matching* y escribir la tupla que coincida con el patrón en el lugar *matchedTake*. Esta transición posee dos arcos de entrada: el arco procedente del espacio de tuplas que provee una tupla *t* y el arco procedente de *pendingTakeQueries* que provee un par *id*/patrón. Para que esta transición sea disparada debe de cumplirse la condición de guarda *guard p.matching(t)*. La función de matching devuelve verdadero si la tupla *t* se ajusta al patrón *p*, falso en sentido contrario.
Renew suministra tuplas *t* procedentes del espacio de tuplas hasta que una cumple la condición de guarda. Si ninguna cumple esa condición, la transición no es tomada. Si se cumple, la tupla *t* se deposita en el lugar *matchedTake* junto con el identificador *id* del patrón. En este lugar puede coincidir con otras tuplas procedentes de otras operaciones *in* de lectura, mediante el identificador *id* asociado cada transición *prepareTake* sabrá cuál es su resultado.

La operación rd es similar. La operación *out* se modela con el canal de entrada *:write(t)* que escribe *t* en el espacio de tuplas. De esta manera la operación *out* es una operación no bloqueante mientras que *rd* e *in* sí que se lo son.

2.2.2 La función de búsqueda y asignación o *matching*

Esta función implementa varios criterios a la hora de decidir si una tupla se ajusta a un patrón determinado. Estos difieren en la equivalencia del comodín o wildcard, excepto la búsqueda y asignación de atributos. La versión inicial de RLinda implementa solo la forma débil, pero este PFC añade las otras dos, ofreciendo finalmente las siguiente:

- **Búsqueda y asignación fuerte (*Strong matching*) -** El comodín (*wildcard*) equivale a un elemento atómico, es decir, no equivale a una tupla.
- **Búsqueda y asignación débil (*weak- matching*) -** El comodín (*wildcard*) equivale a cualquier elemento de una tupla, es decir, puede ser un átomo o una tupla.
- **Búsqueda y asignación de atributos (*attribute matching*) -** El patrón debe de ser una tupla de dos elementos. El comodín (*wildcard*) representa tanto átomos como tuplas.

Supongamos la siguiente tupla formada por tres elementos, uno de ellos otra tupla:
[“hola”,[“atributo”,5],666] . Las funciones de búsqueda y asignación se comportan de la siguiente manera:

- **fuerte:** patrón = [“hola”, *,*]. El comodín solo representa elementos atómicos, y el segundo elemento es una tupla luego la función devuelve falso.
- **débil:** patrón = [“hola”,*,*]. El comodín puede ser cualquier elemento incluido una tupla, luego devuelve verdadero.
- **de atributos:** patrón = [“atributo”,*]. La función busca si en la tupla existe una tupla que se ajuste a ese patrón, es decir, si algún elemento de una tupla es una tupla binaria cuyo primero elemento es la cadena *“atributo”* y segundo es cualquier cosa.

2.3 Distributed RLinda

En RLinda [3] el espacio de tuplas está centralizado lo que deriva en problemas típicos de los sistemas centralizados como no ser tolerante a fallos, hacer un mal balance de carga o trato difícil con ciertos aspectos de seguridad. Una implementación distribuida, DRLinda, es el siguiente paso en la evolución de RLinda.

Una implementación descentralizada puede ayudar a superar los problemas anteriormente comentados pero requiere tener consideraciones adicionales en lo que respecta a la definición de las tuplas y sus operaciones. Por un lado la semántica de Linda debe conservarse mientras que por otro lado las tuplas poseen persistencia ilimitada en el otro modelo, característica difícil de conseguir en modelos distribuidos.

En DRLinda el coordinador actúa como un punto central que distribuye datos a otros nodos de RLinda. Estos nodos se pueden añadir, quitar o configurar según el escenario en el que nos encontramos. Externamente la interfaz del coordinador se publica. Internamente este coordinador juega un rol de cliente con el resto de nodos de RLinda a los que distribuye las tuplas recibidas de acuerdo a un algoritmo de balanceo de carga. Una interfaz de configuración dinámica es también publicada permitiendo configurar en tiempo real algunos parámetros (número de nodos, localización, restricciones, características, etc) haciendo el sistema capaz de amoldarse a las necesidades en un momento particular.

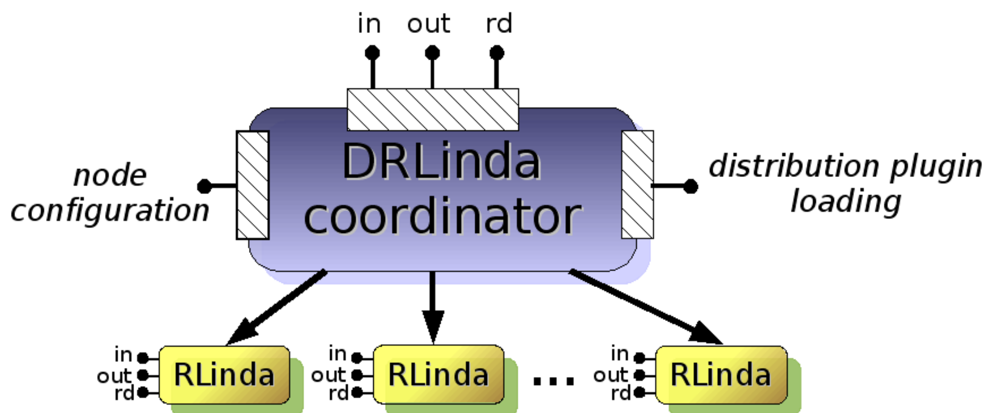


Fig. 3 Estructura de DRLinda y sus nodos

2.4 RLinda WS-Security

Desde el mismo momento en que RLinda es desplegado como servicio Web aparecen toda una serie de consideraciones de seguridad que deben de ser tenidas en cuenta. Por ejemplo, si la información que entra o sale de RLinda es confidencial o si queremos restringir el uso del coordinador a usuarios o entidades autorizadas. RLinda WS-Security [5] garantiza la interoperabilidad del servicio Web SOAP a través de estándar desarrollado por el W3C para la seguridad sobre SOAP: WS-Security. Esta especificación y sus implementaciones ofrecen todas las herramientas necesarias para la seguridad informática: firma digital, encriptación, certificados, etc.
Esta versión de RLinda ofrece una capa de seguridad utilizando el nombrado estándar WS-Security, de forma que se consigue un coordinador de procesos que integra las últimas tecnologías en cuanto a seguridad y criptografía se refiere.
RLinda WS-Security se implementa en dos servicios Web diferenciados, uno correspondiente a la entidad certificadora y otro al propio coordinador de RLinda. La entidad certificadora es propia del

sistema RLinda y se encarga de generar y firmar los certificados públicos de todos los clientes autorizados y del servidor de RLinda.

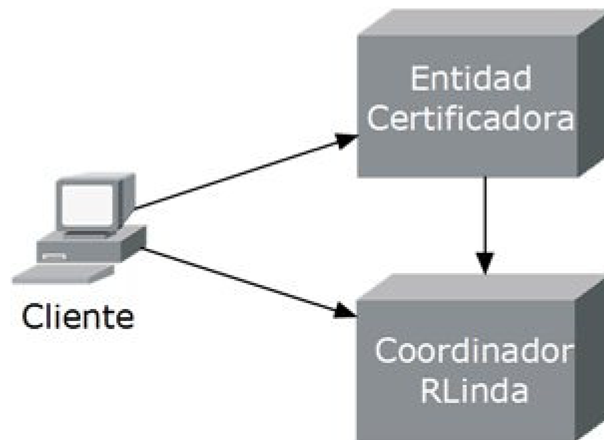


Fig. 4 Interacción de los servicios Web y el cliente en RLinda WS-Security

2.5 Otras soluciones comerciales basadas en Linda

El modelo conceptual de Linda es sencillo, pero no así su implementación. Aspectos como las operaciones de inserción, lectura y borrado de tuplas, las funciones de matching o la definición de las propias tuplas no tienen una solución obvia y la forma a afrontarlos influirá en las características del sistema y su rendimiento.

2.5.1 JavaSpaces

Java Spaces [6], de Sun Microsystems, es una de implementaciones más populares. Se presenta como un mecanismo de coordinación e intercambio de objetos distribuidos para la plataforma Java. La propia especificación de Sun dice:

JavaSpaces provee mecanismos de intercambio de datos y persistencia distribuida para código escrito en lenguaje Java. Los datos están escritos como "entradas" que proveen agrupamiento tipado de campos relevantes.

En un JavaSpace se escriben entradas, y se usan plantillas *templates* para leerlas. Los *templates* son entradas donde algunos o todos sus campos deben corresponderse con entradas que ya existen en el JS (el espacio de tuplas). Los campos dejados como nulos se consideran comodines.

Existen dos tipos de operaciones de lectura: *read* busca una entrada y devuelve una coincidencia, dejándola dentro del espacio mientras que *take* busca la entrada, devuelve la coincidencia, pero la remueve . Si no se encuentra una entrada, se produce una excepción que devuelve inmediatamente.
Además proporciona una operación *notify* que notifica cuando una tupla coincide con un patrón dado.

JavaSpaces soporta además un mecanismo de transacciones, que implica agrupar un conjunto de operaciones sobre un JS o una instrucción sobre varios JSs como una sola operación atómica.
También provee acceso protegido usando identidad.

2.5.2 GigaSpaces

La especificación de JavaSpace inspiró otras como GigaSpaces [7], focalizado en soportar aplicaciones distribuidas de gran tamaño que sean linealmente escalables. En esta aproximación una aplicación se divide en unidades de procesamiento autosuficientes donde la comunicación entre ellas se realiza mediante un espacio de tuplas compartido.

2.5.2 TSpaces

Es una aproximación de JavaSpaces llevada a cabo por IBM. Está pensado como método de interconexión de, textualmente de su página Web [25], cualquier cosa a cualquier cosa. El espacio de tuplas interconecta los diferentes elementos del sistema y posee una base de datos centralizada donde almacenar información permanente.

Incluye nuevas operaciones para las escritura y extracción de múltiples tuplas así como operaciones que especifican tuplas en términos de "*tuple ID*" en lugar de las funciones de matching clásicas.
Permite a las aplicaciones añadir nuevas operaciones al espacio de tuplas de forma dinámica.

2.5.3 Objective Linda

En el mundo académico han aparecido otras aproximaciones como es Objective Linda [9]. Este se presenta como un modelo de coordinación que combina programación orientada a objetos con comunicación generativa desacoplada. Entendemos por desacoplada que es independiente del lenguaje de programación.

Se provee un espacio global de objetos donde se depositan objetos C++. Existen cuatro operaciones disponibles, las tres ya explicadas en RLinda (*out*,*in*,*rd*) mas una operación *eval* que crea procesos Unix.

La correspondencia entre objetos del espacio global de objetos no se realiza de acuerdo al contenido de sus campos. El programador debe especificar un subgrupo de campos que serán tenidos en cuenta a la hora de la función de búsqueda y asignación, que se realiza de acuerdo a los principio de Linda.

2.5.4 ELLIS

ELLIS [10] es un sistema Linda desarrollado por EuLisp. La función de búsqueda y asignación es un método de la clase que modela el espacio de tuplas. Esto permite que el método de búsqueda y asignación sea sobrescrito, pero el mecanismo es complicado y requieren tratas con el espacio de tuplas a muy bajo nivel.

2.5.5 XMLSpaces

Se considera una extensión de TSpaces que proporciona medios de suportar datos expresados en formato XML. Este soporte a XML se realiza mediante las características del lenguaje Java. Extiende la clase Java que modela una tupla para soportar una tupla en XML, especialmente sobrescribiendo la función de búsqueda y asignación, que pasa a trabajar con tuplas en dicho formato XML.
Se basa en la capacidad de java para saber si un objeto es de una clase u otra y entonces aplicar la función de búsqueda y asignación normal o la sobrescrita.

2.5.6 WorkSpaces

WorkSpaces [13] emplea XMLSpaces para coordinar flujos de trabajo (*workflows*) distribuidos a través de la Web. La arquitectura de WorkSpaces se construye alrededor del llamado procesador XLS que recupera las descripciones necesarias o *steps* y documentos de trabajo específicos de una aplicación de un espacio de almacenamiento distribuido llamado XMLSpaces. El procesador ejecuta los pasos que se especifican en la información saca del espacio distribuido.
Como están en XML, cualquier herramienta puede generar nuevos *steps*.

2.6 Casos de aplicación del modelo de coordinación Linda

Para mostrar al lector posibles aplicaciones de este bróker de mensajes, se exponen algunos ejemplos específicos de usos de un espacio de tuplas. Varios de ellos emplean JavaSpaces como implementación de Linda y sería posible sustituirlo por RLinda.

2.6.1 Espacio de tuplas para redes sociales en teléfonos móviles

Una red ad hoc es aquella en la que todos sus nodos poseen la misma importancia. Hoy en día las personas utilizan sus móviles para conectarse en modo ad-hoc unas con otras y sincronizar sus aplicaciones, compartir información de sus contactos, conectarse al GPS… La tendencia en los próximos años es que aplicaciones sociales como las que actualmente funcionan en internet se trasladen a los entornos de este tipo de redes pequeñas.

Incluso siendo redes de pequeña escala construir y operar aplicaciones en rede ad hoc resulta complejo. Esta complejidad viene de la falta de una infraestructura o administración centralizada y una topología para la comunicación que es inestable.

Un espacio de tuplas abstrae a la red subyacente creando un espacio común donde las aplicaciones pueden comunicarse y transmitirse datos. En la aproximación ofrecida por el *Department of Computer Science* de la universidad de Zurich en [14] cada aplicación puede configurar su visión de "memoria compartida" a pesar de que todas las tuplas residen en un espacio común.
Las tuplas poseen atributos de versión y tiempo de vida. El primero sirve para identificar diferentes versiones de una misma tupla en el tiempo, mientras que el segundo establece una fecha de caducidad a las tuplas para que desaparezcan automáticamente del sistema cuando esta llegue.

A la hora de emplear patrones en las funciones de búsqueda y asignación se implementa el comodín descrito en Linda y un nuevo valor *LAST* (último) que devuelve la tupla con versión más reciente.

2.6.2 Coordinación en sistemas multiagente heterogéneos

En un sistema multiagente varios agentes interaccionan entre sí para lograr sus propios fines. Los agentes deben comunicarse y entenderse entre sí de manera que muchos de los esfuerzos a la hora de crear este tipo de sistemas se destinan a diseñar las reglas de la comunicación. Además la noción de agente normalmente intenta ser un concepto de alto nivel más que un proceso de sistema, de manera que resultaría ideal que los agentes no estén ligados a un lenguaje de programación o una tecnología concreta.

Un espacio de tuplas se presenta como el candidato perfecto para gestionar esta comunicación. RLinda no se preocupa de quién se está comunicando sino solo del mensaje que se deposita. Ofrece unas directivas básicas de comunicación con las que los agentes pueden coordinarse y el lenguaje XML como la sintaxis que deben usar. Todo el esfuerzo debe dirigirse pues a la semántica del lenguaje y las acciones que debe realizar cada agente.

2.6.3 Como construir un ComputeFarm

Algunos programas pueden hacerse más rápidos dividiéndolos en partes más pequeñas y ejecutando dichas partes en múltiples procesadores. Esta metodología se conoce como computación paralela y existe un gran número de sistemas hardware y software que la facilitan.

ComputeFarm [19] es un proyecto open *source* del entorno Java para desarrollar y ejecutar programas paralelos. En un ComputeFarm existe un proceso maestro que crea una serie de tareas que deben ser realizadas, los procesos trabajadores toman tareas de la colección, las realizan y pasan el resultado al proceso maestro. El sistema de intercambio de información es un espacio, entendido como un espacio común donde la comunicación está desacoplada.

Los trabajadores son todos y cada uno se ejecuta en una maquina distinta. Cuando un trabajador acaba una tarea elije otra del espacio y vuelve empezar. La carga está balanceada pues son los trabajadores los que escogen tareas cuando están ociosos. Un trabajador en una máquina mucho más rápida acabará antes sus tareas y podrá empezar nuevas, mientras que uno más lento tardará mayor tiempo en realizarlas.

De cara al proceso maestro, el espacio común no es más que un lugar donde el deposita tareas y encuentra resultados, no se preocupa acerca de quien las ha realizado.

2.6.4 Limbo: Plataforma basada en un espacio de tuplas para aplicaciones móviles adaptativas

Los entornos de la tecnología móvil se caracterizan principalmente por sufrir cambios rápidos en la infraestructura disponible, en particular la calidad del servicio QoS (*Quality of Service*) disponible en los canales de comunicación inferiores. Una aplicación que desee funcionar en este entorno y pueda superar y sacar ventaja de estos cambios en la calidad de servicio necesita de una plataforma que soporte sistemas distribuidos.

Actualmente en este tipo de plataformas se provee de paradigmas de programación orientados a conexiones síncronas. En determinados momentos la calidad del servicio puede ser escasa o nula y se recurre a sistemas de buffer que almacenan operaciones pendientes de ser enviadas cuando vuelva el servicio. Este paradigma no acaba de encajar. El *Distributed Multimedia Research Group* del departamento de informática de la universidad de Lancaster propone Limbo [20] como una alternativa a estas plataformas.

Limbo se apoya en el paradigma de un espacio de tuplas, basado en Linda, con cuatro operaciones: out, in y rd que son equivalentes a las descritas en el punto 2.1 sobre Linda y una operación *eval*, similar a out, que crea tuplas activas. Una tupla activa se diferencia del resto de tuplas (tuplas pasivas) en que se invocan procesos separados que realizan diferentes acciones en *background* hasta que finalmente depositan una tupla pasiva.

La plataforma en si ofrece múltiples espacios de tuplas que se pueden especializar para cumplir requerimientos a nivel de aplicación. Jerarquía de tuplas, tuplas con atributos explícitos de calidad de servicio y agentes del sistema que monitorizan la calidad del servicio.

2.6.5 Computación basada en espacios de tuplas para la Web Semántica

Las tecnologías semánticas prometen solucionar los problemas de las aplicaciones Web actuales. Cada vez son más aceptadas en el entorno empresarial pero esto su uso en entornos abiertos como la Web - respecto a temas de escalabilidad, dinamismo y distribución- requiere más información.
La Web semántica ha heredado el modelo de comunicación clásico de la Web basado tecnologías para el intercambio de mensajes de manera síncrona como llamadas a procedimientos remotos (RPC).

La computación semántica basada en espacios de tuplas (*Semantic tuplespace computing*) es un instrumento para mejorar las aplicaciones de la Web semántica, basándose en que la auténtica comunicación estilo Web con un servicio Web se debe basar, de forma análoga con la Web convencional, en el acceso compartido a datos persistentes publicados en lugar de en paso de mensajes. Un espacio de tuplas aporta esta forma de comunicación análoga a la que se realiza en internet.

En *"Tuplespace-based computing for the Semantic Web: a survey of the state-of-the-art"* [21] se analizan y comparan soluciones propuestas para dar una idea de el estado de desarrollo de este tipo de tecnologías.

Capítulo 3 | **Objetivos**

En resumen los objetivos de este PFC son:

- Analizar, diseñar e implementar los mecanismos necesarios para dotar al sistema de coordinación RLinda de persistencia de datos con independencia de la tecnología utilizada para su almacenamiento, permitiendo su volcado y recuperación.

- Analizar, diseñar e implementar los mecanismos necesarios para permitir la temporización de las operaciones y los datos del sistema de coordinación RLinda como un añadido a RLinda integrado con la capa de persistencia.

- Proporcionar Accesibilidad a la plataforma mediante el uso de estándares y tecnologías de servicios Web, concretamente se habilitará la interacción mediante los estilos SOAP y REST.

- Analizar los problemas de rendimiento observados en la comunicación RMI entre la interfaz que proporciona acceso mediante servicios Web y el núcleo de la aplicación, diseñar una solución e implementarla.

- Dotar de mecanismos de verificación de datos mediante esquemas de comprobación basados en los estándares usados.

- Extender la función de equivalencia de RLinda para que soporte tres criterios distintos de equivalencia de tuplas permitiendo configurar dicha función desde el exterior del sistema.

- Desarrollar un caso real de aplicación.

- Analizar las prestaciones del sistema desarrollado en un clúster, analizando los resultados con respecto a los obtenidos inicialmente para RLinda.

Capítulo 4 | **Diseño de WS-PTRLinda**

Una vez presentado RLinda y vistas sus aplicaciones es el momento de ver el diseño de esta nueva versión. WS-PTRLinda se compone de dos nuevas capas que se sitúan sobre el RLinda inicial añadiendo al sistema original nuevas funcionalidades. Estos módulos o capas son independientes entre sí y pueden usarse por separado. Ya hemos visto que RLinda se apoya en la implementación de las redes de Petri de Renew, las *Reference Nets [4]*. La *ReferenceNet* que modela RLinda se modifica con nuevas transiciones y elementos que soporten las nuevas funcionalidades que se añaden desde cada nueva capa. Dada esta arquitectura por capas se ha seguido una metodología iterativa e incremental comentada en el anexo correspondiente.

Existen tres fases claras en el diseño, una inicial que propone algunos cambios al sistema y dos posteriores que suponen una inversión de tiempo considerable y que corresponden a cada una de las nuevas capas de RLinda.

4.1 Fase uno: RLinda Modificado (WS-RLinda)

El punto de partida es el sistema RLinda con acceso mediante protocolos RMI y SOAP que ofrece las tres directivas básicas *out*, *in*, *rd*. La primera versión del sistema incorpora acceso al servicio Web mediante protocolo REST a través de un servidor Apache Tomcat embebido en la propia aplicación. Se eligió esta manera tras comprobar que RLinda sufre una bajada en su rendimiento a partir de cierta concurrencia de clientes debida a la comunicación entre el antiguo servidor Apache Tomcat externo y la aplicación (mediante RMI). Al ser Apache Tomcat externo reside en una máquina virtual Java distinta de RLinda y deben emplear protocolo RMI para comunicarse.
Apache Tomcat ofrece una versión embebida que se llama desde el propio código, de manera que puede lanzarse el servidor desde la misma máquina virtual. Al compartir memoria no es necesario recurrir al protocolo RMI para la comunicación entre el servidor Apache Tomcat y RLinda.

Comentar también que se añaden nuevos criterios a la función de *matching* y la posibilidad de configurar este parámetro ofrecida mediante una nueva operación *setMatching*. El diseño del sistema puede verse en la figura 5.

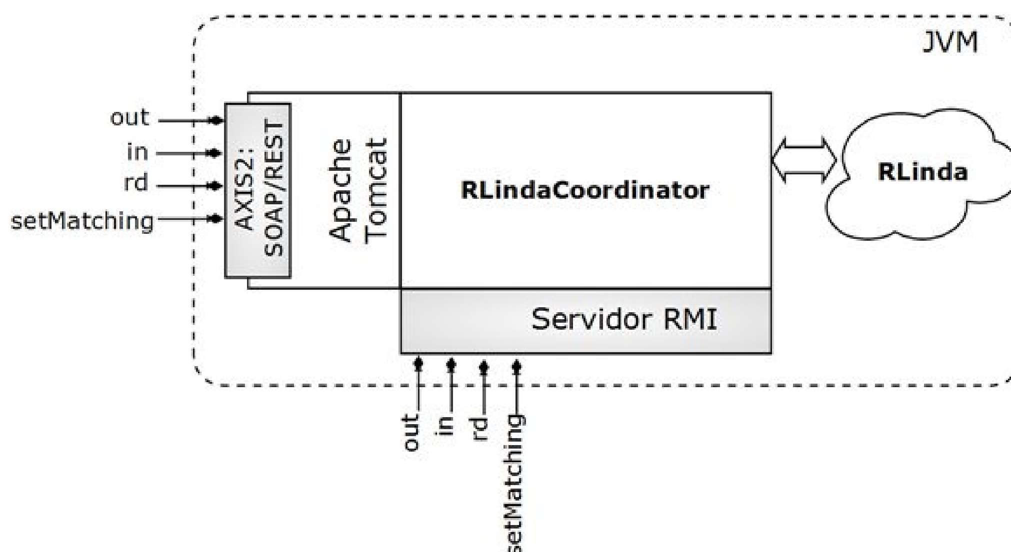


Fig. 5 Diseño de RLinda Modificado

La fig. 5 muestra cómo la comunicación con el exterior se hace mediante los servidores de servicios Web y RMI que tienen acceso al coordinador RLinda. Es este coordinador mediante los

anteriormente citados *stubs* el que pasa las tuplas de los servidores a la red de WS-RLinda. En las siguientes versiones nos apoyaremos sobre esta estructura para aplicar nuevas funcionalidades.

4.2 Fase dos: Persistent RLinda

Un proceso Web complejo se compone de varios procesos Web más sencillos que deben comunicarse y coordinarse entre sí para alcanzar sus fines. Estos procesos Web complejos pueden requerir de cierto tiempo hasta que son completados y pueden poseer varios puntos críticos. La información en RLinda está contenida en memoria lo cual aporta rapidez pero deja el sistema a merced de un fallo del sistema operativo o una pérdida de corriente por poner algún ejemplo.

La capa de persistencia de RLinda aporta la tolerancia a fallos, duplicando la información contenida en memoria en una base de datos. Se ofrecen dos formas diferentes de persistencia, una más conservadora que garantiza la tolerancia a fallos total pero que disminuye la velocidad del sistema y otra más rápida pero que admite ciertos riesgos a la hora de prevenir errores.

- **persistencia fuerte -** Cada operación que modifica el espacio de tuplas (*out()* escribe tuplas e *in()* las retira) produce una interacción con la base de datos. Estas acciones se realizan desde la propia capa de persistencia y la red de RLinda mantiene su funcionamiento habitual.

- **persistencia débil -** Las interacciones con la base de datos se posponen para más adelante, ofreciendo mayor velocidad de respuesta. Estas interacciones se realizan desde la propia red de RLinda atendiendo a criterios de tiempo o cantidad de tuplas sin ser persistidas.

Decimos que una tupla es persistente si existe coherencia entre la base de datos y el espacio de tuplas. Es decir, si la tupla ha sido retirada del espacio de tuplas y también ha sido borrada de la base de datos o si la tupla ha sido escrita en el espacio de tuplas y también en la base de datos.

Tanto el tipo de persistencia con la que se va a trabajar como los criterios de la persistencia débil son configurables desde el exterior.

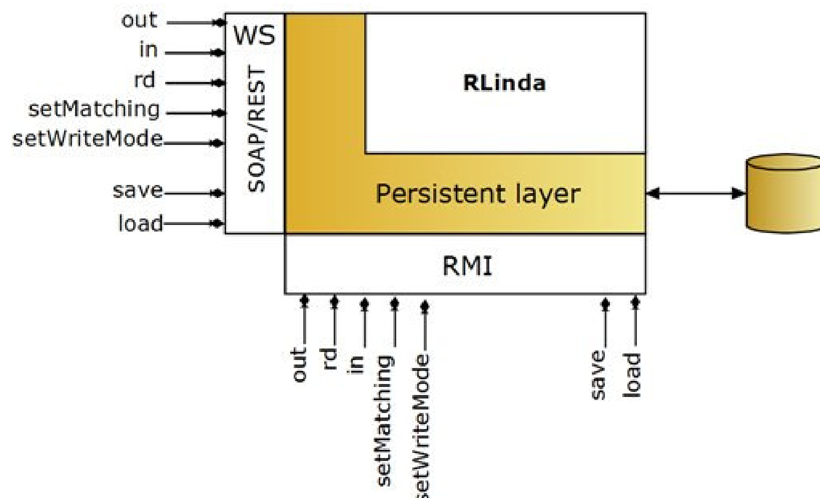


Fig. 6 Diseño de Persistent RLinda

El mecanismo de copia de seguridad reside en dos funciones *load* y *save* que crean una copia de seguridad y la envían al cliente o reciben dicha copia y restauran el estado de PRLinda .La figura 6 muestra el diseño de este nivel.

La capa de persistencia se sitúa entre los servidores de acceso y el coordinador de RLinda.

4.3 Fase tres: Temporized RLinda

En el contexto de una comunicación entre procesos Web que se coordinan y se envían datos entre sí, es probable que cierta información solo sea válida durante un tiempo determinado o, en otro caso, un proceso solo pueda permitirse esperarla durante un tiempo concreto. Pongamos como ejemplo un servicio Web diseñado para recibir información de un servicio de confianza pero que antes debe investigar durante un breve espacio de tiempo si existen otros servicios en el sistema más rápidos que le convengan más. Con RLinda como centro de la comunicación, sería estupendo poder lanzar una operación que mire si hay otros servicios disponibles pero que pasado un tiempo sin encontrar nada desbloquee al proceso para que pueda seguir su curso. Otra opción sería que si un servicio tiene una información que será solo valida durante un día, pueda de alguna forma indicarlo. Pasado un día otros servicios nunca podrán acceder a esa información (mejor, está obsoleta).

La capa de temporización aplica el concepto de tiempo tanto a datos como a operaciones. Las operaciones básicas de RLinda pasan a admitir un parámetro de tiempo o *timeout* que indica cuanto tiempo serán válidas en el sistema. Así mismo, las tuplas que se emplean en dichas operaciones también pueden ser temporizadas añadiéndoles una cabecera indicativa.

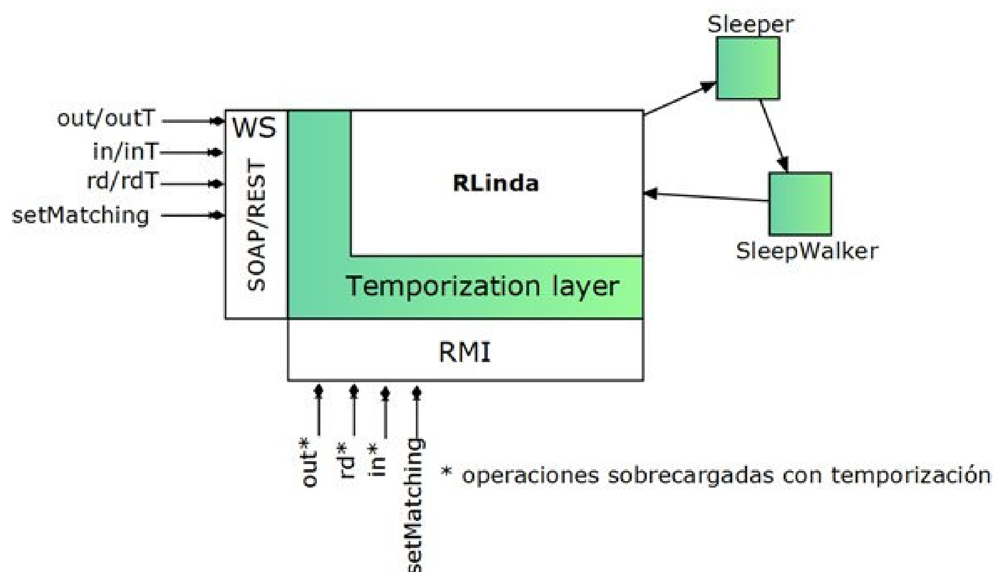


Fig. 7 Diseño de Temporized RLinda

Las operaciones de lectura *in* y *rd* son bloqueantes, lo que quiere decir que al expirar deben desbloquearse y recibir un resultado que indique al proceso que las ejecutó que se han desbloqueado porque han expirado y no porque existiese una tupla que se ajusta a su patrón. Para ello existen Los objetos *Sleeper* (durmiente) y *SleepWalker* (noctámbulo) encargados de gestionar los *timeouts* de operaciones y datos. Cuando un *timeout* expira, interaccionan con la red de TRLinda para proporcionar un resultado a la operación y que esta se desbloquee.

La operación de escritura no es bloqueante de manera que la temporización solo afecta a sus datos. Las tuplas temporizadas del espacio de tuplas son descartadas por la función de búsqueda y asignación una vez expiradas, así que nunca podrán ser devueltas a una operación de lectura.

Pongamos por ejemplo que un proceso A escribe una tupla ["A"] en el espacio de tuplas con la llamada al método *out(["A"],30)*. La tupla será válida durante 30 segundos. Pasados 31 segundos otro

proceso B quiere comprobar si el proceso A ha escrito una tupla en el espacio de tuplas pero no quiere estar bloqueado permanentemente esperando. Ejecuta una operación *in([“A”],10)*. Como la tupla [“A”] ya ha expirado el proceso B se desbloqueará después de los 10 segundos que dura el timeout de su operación.

En el capítulo correspondiente a la implementación pueden encontrarse algunas trazas de ejecución que muestran ejemplos de cómo la temporización afecta a los datos y a las operaciones.

4.4 El sistema completo: WS-PTRLinda

La versión final incorpora las dos capas anteriores para reunir juntas todas las nuevas funcionalidades. Pese a poder emplearse individualmente, en su uso conjunto la capa de temporización funciona siempre por encima de la capa de persistencia. La capa más superficial es una clase WS-PTRLinda que simplemente incorpora una referencia a la capa inferior y agrupa todas las operaciones que se ofrecen.

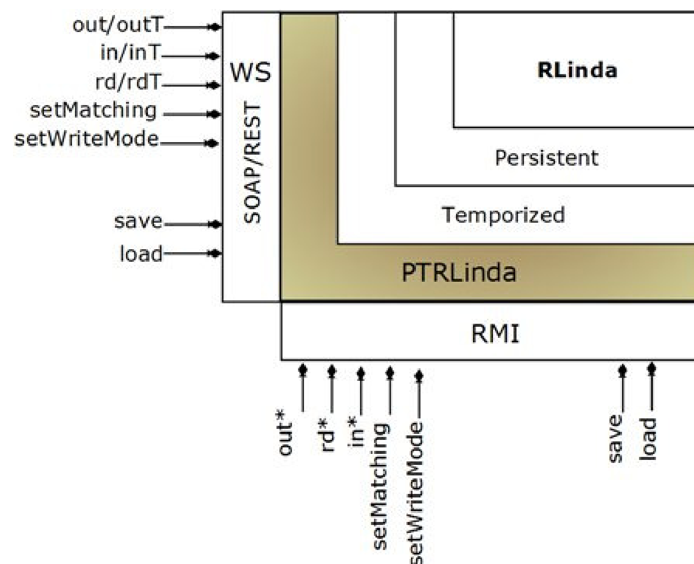


Fig. 8 Todas las capas del sistema

En el capítulo de implementación se analiza y explican las nuevas funcionalidades así como los elementos desarrollados que las soportan.

Capítulo 5 | **Implementación de WS-PTRLinda**

En el capítulo anterior se han descrito las capas que forman el sistema, las nuevas funcionalidades que estás aportan y se han dado a conocer los actores que intervienen en cada una. Ahora es el momento de ver que es necesario implementar y como se ha hecho para proporcionar a RLinda las nuevas funcionalidades que lo convierten en WS-PTRLinda.

5.1 *Persistent* RLinda

La capa de persistencia de RLinda aporta la tolerancia a fallos y lo hace mediante el uso de la herramienta Hibernate que utiliza para facilitar la comunicación con la una base de datos MySQL.

5.1.1 Funcionalidades añadidas

Las mejoras añadidas por esta capa son:

- Modo de trabajo con persistencia fuerte.
- Modo de trabajo con persistencia débil.
- Operaciones *load* y *sabe*.
- Métodos para configurar el tipo de persistencia y criterios relacionados con esta.

A lo largo de este capítulo se mencionan algunos nombres y conceptos que se aclaran a continuación:

5.1.2 Tecnologías utilizadas

La capa de persistencia de WS-PTRLinda se basa en la herramienta Hibernate para gestionar la comunicación con una base de datos MySQL. Estos elementos se describen a continuación quedando extendidos en el anexo correspondiente a la implementación de sistema extendida.

Hibernate

Hibernate es una herramienta de mapeo objeto-relacional para la plataforma Java, su objetivo es solucionar las diferencias entre dos modelos de información distintos entre sí, la base de datos tradicional y el modelo orientado a objetos. Esta herramienta permite abstraer al programador de la labor de escribir las consultas SQL correspondientes para almacenar los atributos de sus objetos en una base de datos, añadiendo una sobrecarga mínima al sistema.

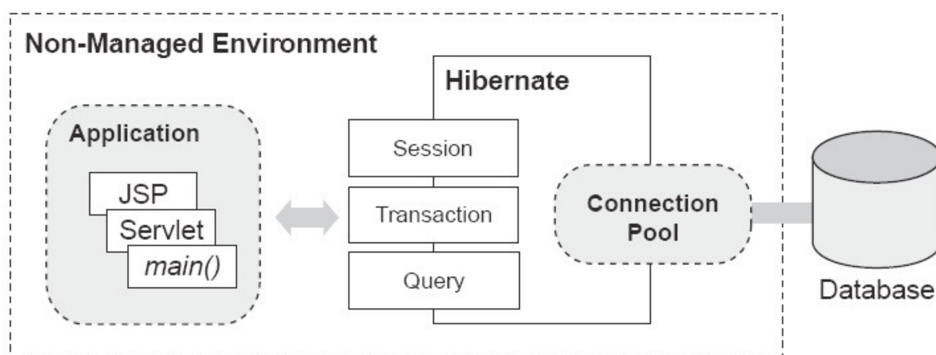


Fig. 9 Dónde se sitúa Hibernate

Hibernate utiliza esquemas *XML* definidos por el usuario para saber cómo debe convertir un objeto de una clase Java concreta a una tabla específica de la base de datos. Hibernate se configura mediante un fichero externo XML y posee un pool de conexiones para agilizar la interacción con la base de datos. En nuestro caso, emplearemos Hibernate para añadir persistencia a WS-PTRLinda guardando una copia actualizada en tiempo real de la información contenida en memoria.

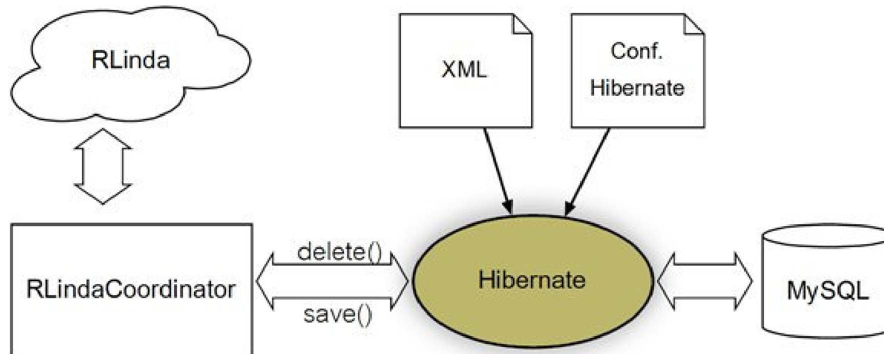


Fig. 10 Dónde se sitúa Hibernate en WS-PTRLinda

Hibernate no funciona como un objeto Java normal que puede ser creado y llamado desde cualquier otra clase, Hibernate funciona en *background*. Por ello, tanto para iniciarlo como para interactuar con él empleamos una clase sencilla de nombre **HibernateUtil**.

Hibernate es accedido desde el código Java mediante un objeto de tipo *SessionFactory*. Es trabajo de *HibernateUtil* construir este objeto y que sea accesible desde otros puntos del código.

```
Session session = HibernateUtil.getSessionFactory().openSession();
Transaction tx = session.beginTransaction();
try{
   tx.save(flarlar);
}
```

5.1.2 ¿Qué datos deben ser persistidos?

WS-PTRLinda trabaja en un entorno abierto en el que los clientes se conectan a través de internet. Esto quiere decir que la comunicación del sistema está regida por los protocolos de comunicación que se empleen y por sus características y limitaciones (por poner un ejemplo: el tiempo máximo de de conexión).

Si se produce un fallo y hay que restaurar WS-PTRLinda, las conexiones con los clientes habrán caído. Por ello no necesitamos preocuparnos de guardar una copia de los patrones de las operaciones de lectura (fig. 11). Una operación de escritura *out(t)* en cambio, introduce una tupla en el espacio de tuplas y el cliente sigue su curso. El espacio de tuplas contiene todos los mensajes y los datos y es la información que más nos urge duplicar para poder restaurarla, de manera que si que habrá que tenerla en cuenta a la hora de garantizar la persistencia.

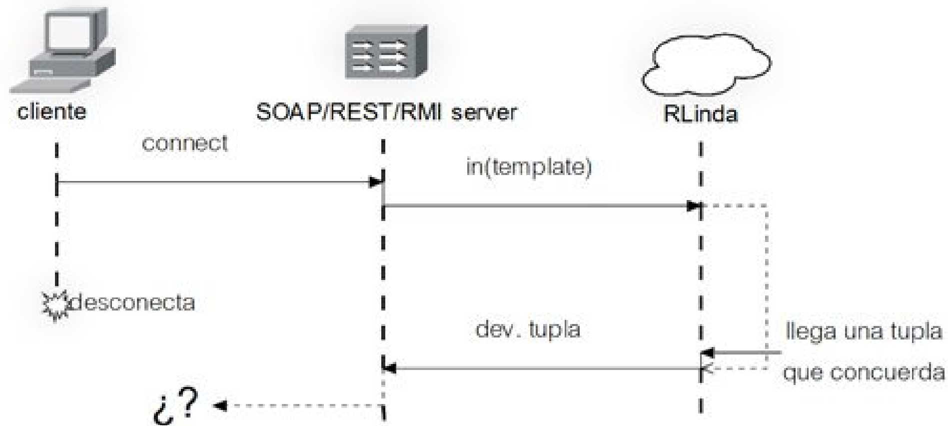


Fig. 11 Problema de desconexión de un cliente

Generador de persistencia

Todo objeto que vaya a ser manejado por Hibernate debe tener asignado un id único, o dejar que sea Hibernate quien lo genere. Este id funcionará como clave primaria en la base de datos. WS-PTRLinda requiere emplear dicho *id*s antes de que Hibernate los asigne (ver anexo de problemas y limitaciones) de manera que se desarrolla un generador propio de *id*s modelado en la red de Petri del coordinador al que se accede mediante una operación getId().

Cuando nos refiramos a este id hablaremos siempre de *persistence id* o *pid* ya que dentro de WS-PTRLinda las operaciones de lectura poseen ya un identificador de nombre *id*. En resumen cada tupla tiene un *pid*, que es único y sirve como clave primaria en la base de datos.

5.1.2 Organización de Persistent RLinda como servicio Web

PRLinda se presenta en dos servicios Web diferenciados, implementados y desplegados mediante Apache AXIS2 en el interior de un servidor Apache Tomcat embebido. Un servicio Web es el propio coordinador de PRLinda que ofrece las operaciones básicas y otro ofrece las funciones de creación y restauración de una copia de seguridad, enviándolas al cliente (o recibiéndolas).

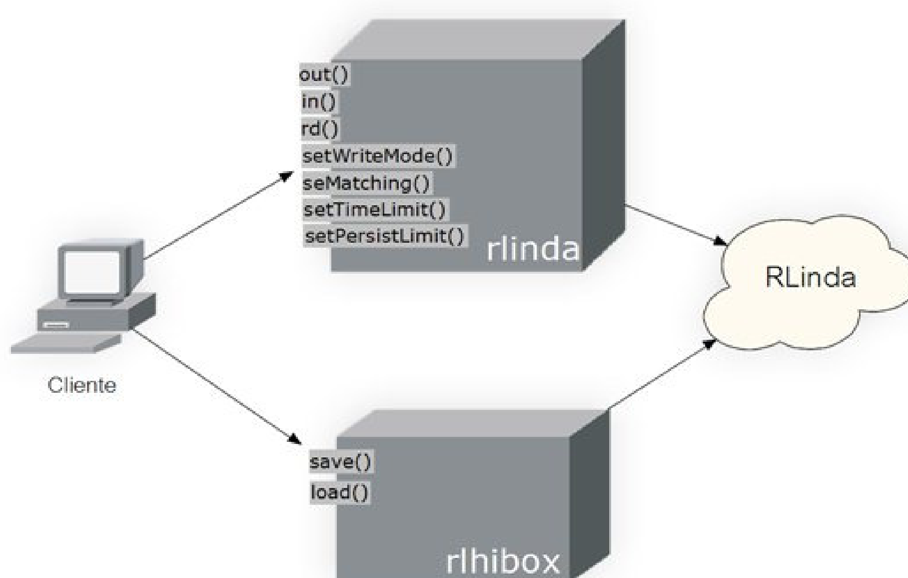


Fig. 12 Servicios Web en Persisten RLinda

El servicio Web encargado de las copias de seguridad, ***rlhibox***, accede al coordinador únicamente para hacer uso de las operaciones *load* y *save*. Estas operaciones bloquean las operaciones de escritura *out* o de lectura destructiva *in* mientras se crea o se restaura la base de datos para evitar problemas de coherencia. Las operaciones de lectura no destructivas *rd* no modifican el espacio de tuplas así que no se ven afectadas.

5.1.3 Arquitectura del sistema

La clase Java que representa la capa de persistencia recibe el nombre de HibernateBox ya que su función es comunicarse con Hibernate para guardar/borrar los datos necesarios de la base de datos y, una vez hecho, interaccionar el coordinador. La figura 14 muestra la arquitectura del sistema.

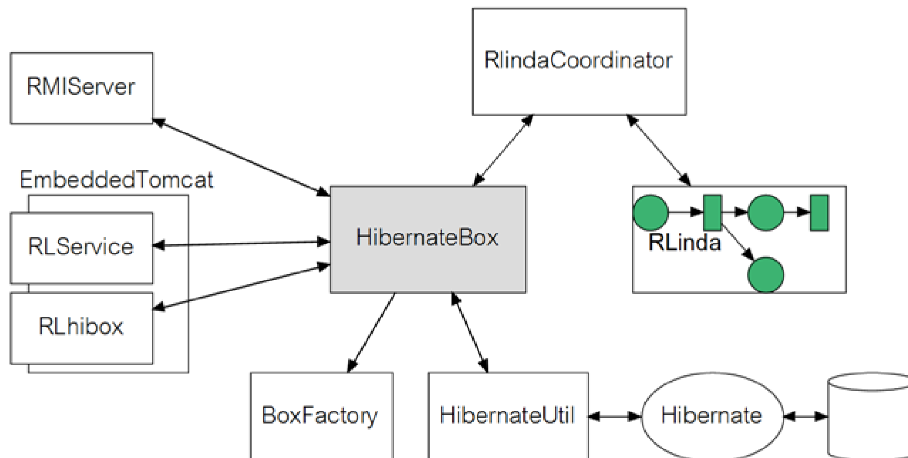


Fig. 13 Arquitectura de PRLinda

Hibernate Box

Hibernate box es la capa de presentación de la persistencia, por lo tanto encapsula toda la lógica de aplicación y ofrece las nuevas funcionalidades a través de sus métodos. Además de las operaciones básicas de Rlinda, estas son las operaciones que se ofrecen al exterior.

- Operaciones ***out,rd,in*** básicas de RLinda**.**
- Operación ***load*** - restaura la base de datos y el estado de PRLinda a través de una copia de seguridad *filedata* de nombre *filename*.
- Operación ***save*** - crea una copia de seguridad de la base de datos. Está función se utiliza desde el lado del cliente RMI y transmite el archivo de copia de seguridad a su directorio.
- ***setWriteMode(int mode)*** - Establece el modo de persistencia (1 – débil , 2- fuerte).
- ***setPersistLimit (int limit)*** - en persistencia débil, límite de tuplas que pueden permanecer sin persistir.
- ***setPersistTime(long time)*** - segundos cada los cuales se realizan operaciones de sincronización.

HibernateBox implementa la interfaz HibernateRemote de manera que puede ser publicado e instanciado mediante RMI.
La clase BoxFactory ofrece métodos para el empaquetado, desempaquetado y limpieza de archivos de la copia de seguridad.

5.1.4 Tipos de persistencia ofrecidos

Existen muchos tipos de procesos Web cada uno con diferentes necesidades. Mientras uno puede necesitar que la comunicación sea lo más rápida posible y no tenga grandes problemas en perder información si ocurre un fallo, otro puede tener varios puntos críticos y requerir estar seguro de que sus datos están garantizados y que, en caso de fallo, podrá recuperar su estado y seguir en el punto en el que estaba.

Con un escenario tan amplio de posibilidades se han desarrollado dos formas distintas de proporcionar persistencia, una fuerte que garantiza que en todo momento en la base de datos hay una copia exacta del contenido de la memoria y otra que realiza las tareas de escritura o borrado de información en la base de datos periódicamente y de acuerdo a criterios configurables. De esta forma el sistema es flexible en función de a qué escenario se enfrenta y se deja la puerta abierta a un trabajo futuro de un balanceador que decida en función de las condiciones de tráfico si debe aplicar una u otra forma.

El tipo de persistencia que se está empleando es también llamado modo de escritura (*writeMode*), que puede ser lento o rápido en función de si la persistencia es fuerte o débil respectivamente.

Persistencia fuerte

En este modo las operaciones de escritura *out* invocadas realizan la escritura de la tupla en la base d datos y solo una vez que esta se ha producido pasa a insertar la tupla en el espacio de tuplas.
Las operaciones de lectura *in* (*rd* no modifica el espacio de tuplas) que han extraído una tupla del espacio de tuplas acceden a la base de datos para eliminarla antes de devolverle dicha tupla al proceso que ejecutó la llamada.

En la figura 16 puede verse los pasos seguidos por una operación *in()* desde que entra en el sistema hasta que devuelve el resultado al proceso cliente. El diagrama de la operación *out()* es equivalente y puede encontrarse en el anexo extendido.

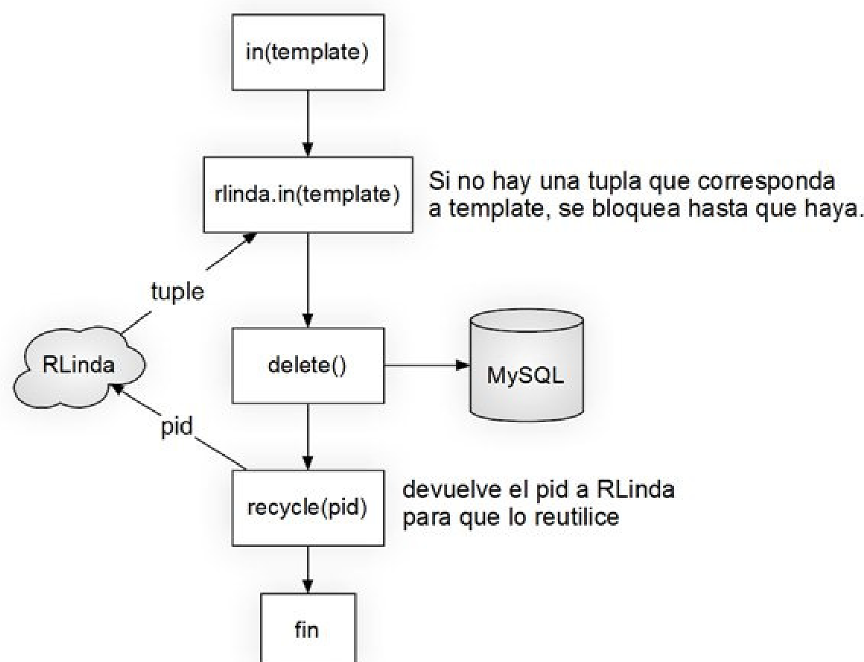


Fig. 14 Pasos de una operación in en persistencia fuerte

Persistencia débil

Este modo de persistencia permite una interacción con PRLinda rápida pero sin renunciar a proteger la información en una base de datos. A diferencia del modo fuerte, las labores de persistencia no se realizan en la propia capa de persistencia sino que se delegan a la red de PRLinda.
Esta realiza periódicamente, acorde a criterios configurables, el copiado o borrado de tuplas en la base de datos (labores de sincronización). De esta forma el cliente no se ve ralentizado por accesos a la base de datos. La contrapartida es obvia, las tuplas insertadas o borradas entre una de estas labores de sincronizado están expuestas a fallos.

La regularidad de estas operaciones de sincronización viene dada por dos parámetros configurables por métodos ofrecidos al exterior:

- **Tiempo** - Número de segundos cada los cuales se van a volcar las tuplas en la base de datos (en escritura) o en lectura.
- **Número de tuplas** - Cuando el numero de tuplas sin volcar en la base de datos (o sin borrar) sobrepase este valor frontera, realizar las operaciones de sincronizado.

Almacén intermedio de tuplas borradas y sistema de borrado

A continuación se muestran y se describe el mecanismo de borrado de tuplas en persistencia débil ligeramente más sencillo que el de escritura y perfecto para explicar al lector como se realizan las labores de sincronización. Pueden encontrarse todas las modificaciones realizadas en la red de PRLinda en el anexo de implementación extendida.

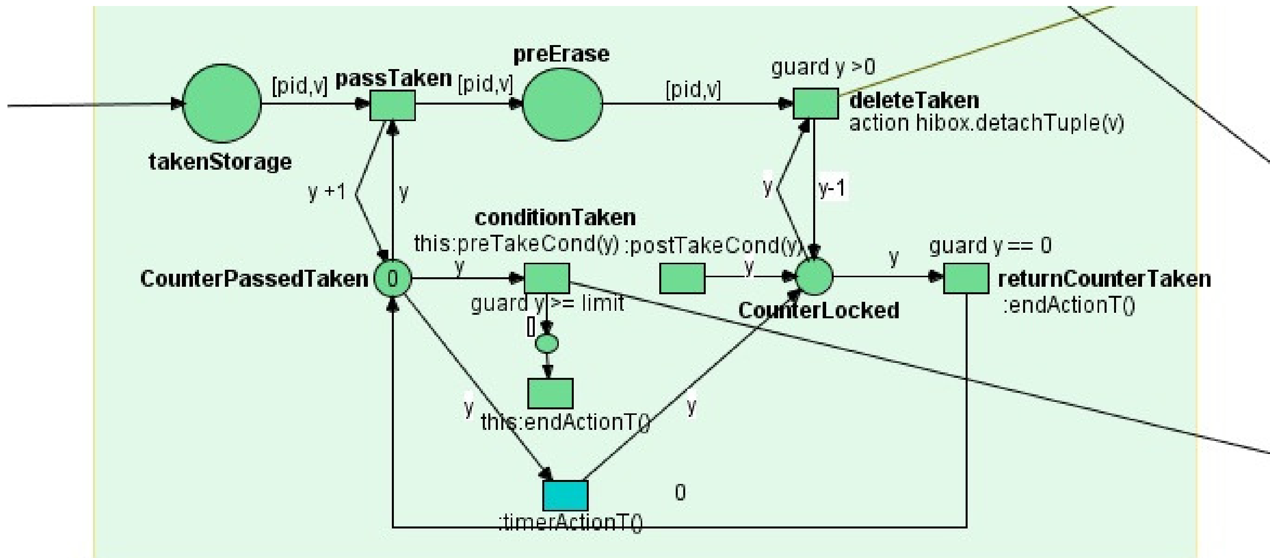


Fig. 15 Mecanismo de borrado de tuplas en persistencia débil

La fig.18 muestra como las tuplas llegan procedentes de la función de matching y se almacenan en el lugar *TakenStorage*. Las tuplas van pasando al espacio de *prePersistence* hasta que el contador (que representa el número de tuplas que hay en *prePersistence*) llega al valor frontera de límite de tuplas. Desde esta zona son leídas por la transición deleteTakens que se encarga de borrarlas de la base de datos. Para que pueda dispararse esta transición el contador ha tenido que sobrepasar el límite impuesto, una vez sucedido, el contador se desplaza hacia la derecha donde se produce un bucle (hasta que el contador vale cero) que dispara la transición que persiste una tupla una vez por cada tupla en preErase.

El mecanismo de escritura es similar, está dotado de una transición más cuya función es que antes de cada operación de borrado de tuplas.

Una situación problemática

La existencia de varias condiciones que inicien las operaciones de sincronización del espacio de tuplas con la base de datos puede generar una situación de falta de coherencia entre memoria y base de datos. A esto hay que añadirle que Renew, el programa sobre el que está diseñada la red de RLinda, dispara las transiciones de forma no determinística lo que añade aun más dificultad para prever algunos comportamientos. Como se muestra en la figura inferior, podría dar lugar a que una tupla intente ser borrada de la base de datos antes de ser escrita, produciendo una excepción y luego tarde o temprano sería escrita (¡cuando debía haber sido borrada!).

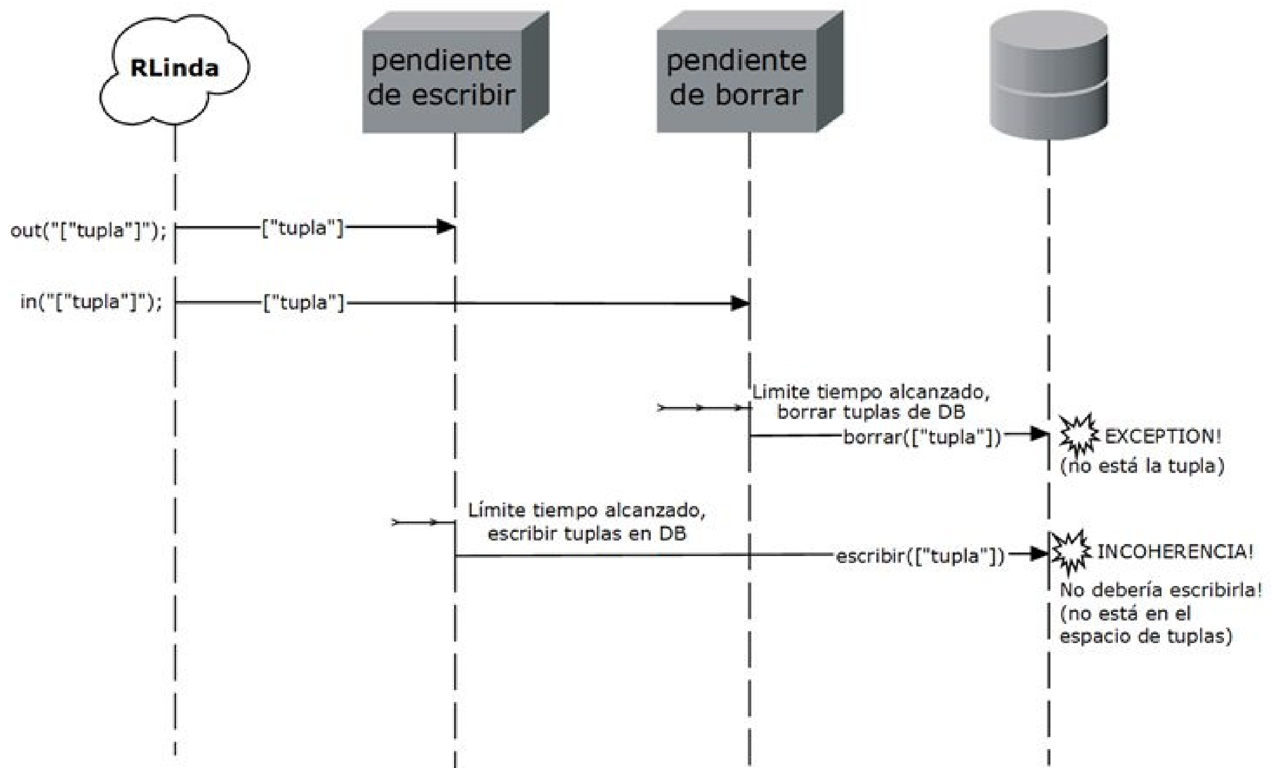


Fig. 16 Problema en la persistencia débil

Para solucionar este problema se emplean dos técnicas conjuntamente.

En el caso de que se cumpla la condición de tiempo para las tuplas pendientes de ser borradas, se guardan todas las tuplas pendientes de escribir antes de eliminar las pendientes de borrar. De esta forma se minimizan los casos en los que se intenta borrar una tupla aun no escrita.

Para prevenir los casos restantes, la función de borrado comprueba antes si existe la tupla que va a borrar, en caso afirmativo se borra normalmente, si no es escrita una tupla comodín *["FAKETUPLE"]* que servirá para alertar al método de escritura. Cuando una tupla va a ser escrita en la base de datos, antes comprueba que no haya una tupla comodín en el lugar correspondiente, si la encuentra la borra y no guarda la tupla (ya no está en el espacio de tuplas).

Como todas las tuplas poseen un id de persistencia único que actúa como clave primaria en la base de datos, podemos emplear este para saber si hay ya una tupla escrita en una posición determinada. De esta manera se solucionan los problemas de coherencia comentados.

5.1.5 Funciones Load / Save

Estos métodos son los responsables de realizar las copias de seguridad o restaurarlas y enviarle o recibir dichas copias del cliente. Un cliente puede acceder mediante RMI, donde el envío de archivos es sencillo de implementar, o desde un servicio Web SOAP, que no soporta directamente el envío de archivos sino que hay que hacer uso de *SOAP with Attachments* (*SwA*).

La clase HibernateBox implementa una serie de métodos destinados a realizar y restaurar las copias de seguridad. También ofrece los métodos empleados en RMI para enviar o recibir dichas copias mientras que en el caso de SOAP se implementan en el propio servicio. Las figuras 20 y 21 muestran como se realizan las operaciones *save* y *load* y los diferentes métodos implicados (y si está en el ámbito de HibernateBox o en el servicio Web).

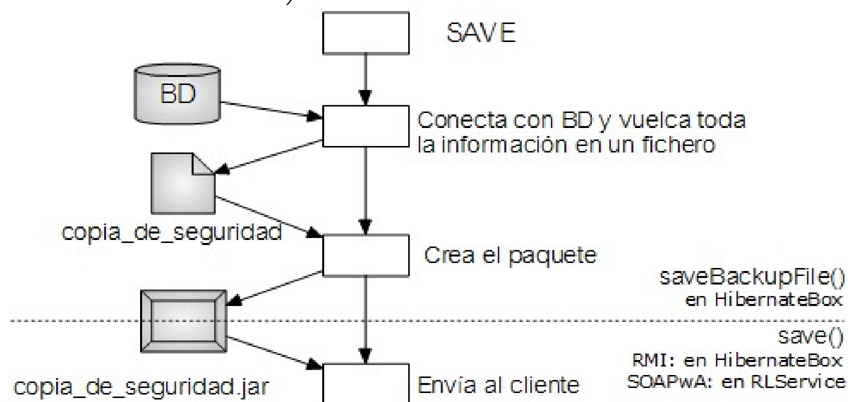


Fig. 17 Operación Save

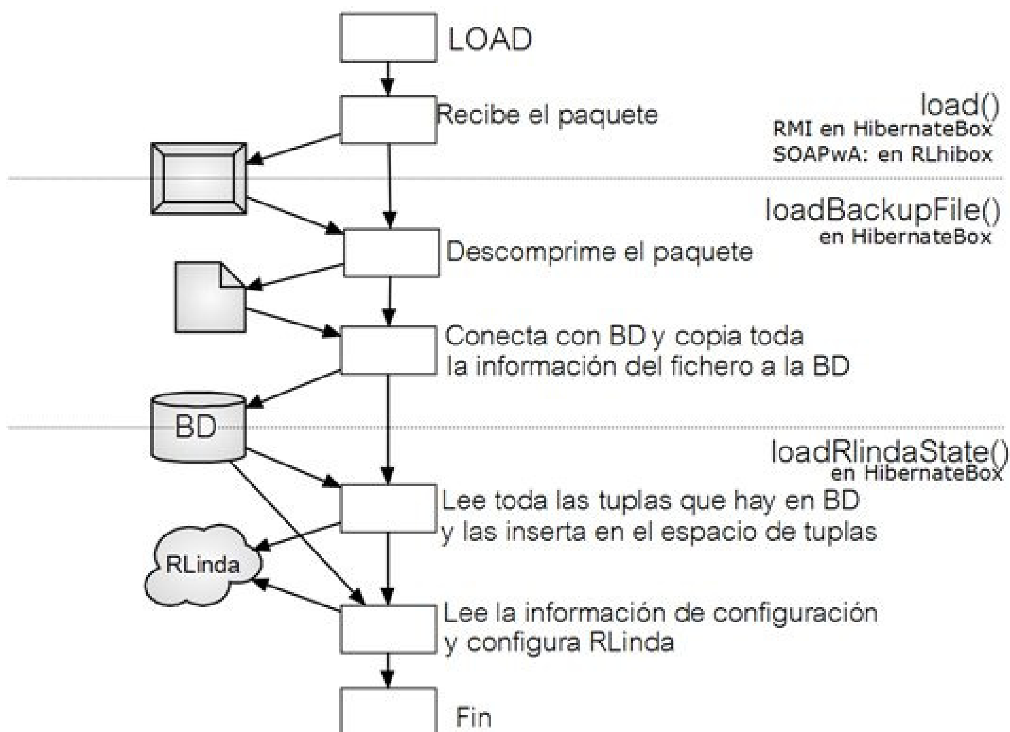


Fig. 18 Operación Load

5.2 *Temporized* RLinda

Aplicando el concepto de tiempo a las operaciones y los datos de RLinda aparecen nuevas posibilidades de uso del sistema, siendo más versátil pero manteniendo la filosofía de unas pocas directivas de comunicación. Veamos cómo afecta a las operaciones ya existentes el parámetro de tiempo, como se implementa en el interior de RLinda (qué cambios han sido introducidos en la red) y el mecanismo encargado de desbloquear las operaciones de lectura que no han obtenido respuesta.

5.2.1 Funcionalidades añadidas

- Las directivas básicas de comunicación de RLinda se sobrecargan para admitir un parámetro de tiempo*: out(tuple,timeout), in(pattern, timeout), rd(pattern,timeout),* (Temporización de operaciones)
- Las tuplas pueden incorporar una cabecera indicando el tiempo que serán válidas en el sistema (Temporización de datos).

Estas dos nuevas funcionalidades nos harán hablar en este capítulo de varios conceptos clave:

- valor de timeout (o timeout) - tiempo de validez de una operación o una tupla desde el momento que entra en el sistema.
- tiempo/fecha de expiración - punto en el tiempo a partir del cual la operación o la tupla ya no es válida.

Una tupla puede estar temporizada si tiene una cabecera ["TIMEOUT",valor] dónde valor indica el *timeout* en segundos. Se puede ejecutar una operación temporizada con una tupla temporizada.

5.2.2 Organización de TRLinda como servicio Web

Temporized RLinda se implementa como un servicio Web único que ofrece tanto operaciones temporizadas como sin temporizar. AXIS2 no soporta la sobrecarga de métodos de manera que las operaciones habituales se duplican para ofrecer la posibilidad de parámetros duplicados.

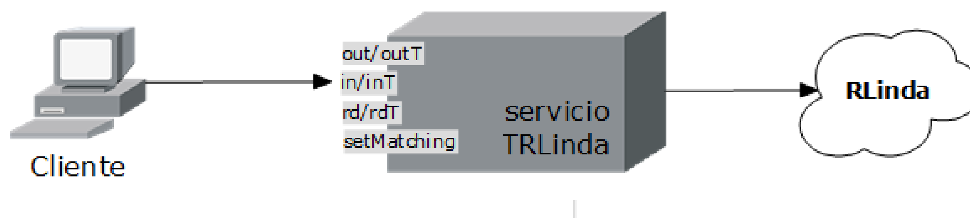


Fig. 19 Visión del servicio Web desplegado por Temporized RLinda

5.2.3 Arquitectura del sistema

Las nuevas funcionalidades de la capa de Temporización de WS-PTRLinda se implementan mediante los siguientes elementos.

TemporizationBox

La clase TemporizationBox es la clase que representa la capa de temporización de TRLinda. A diferencia de la capa de persistencia, esta es una clase muy sencilla que simplemente sobrecarga las operaciones básicas de RLinda (*out*,*in*,*rd*) con sus versiones temporizadas.

Implementa la interfaz TemporizationBoxRemote que permite accederla por RMI.

SleepWalker y Sleeper

Una operación de lectura se bloquea debido a que una transición no se puede disparar, de manera que no es trivial desbloquearla. Para ello se necesita una clase que funcione en *background* que lleve la cuenta del tiempo que queda a cada operación antes de que expire. Toda operación de lectura se almacena en un lugar de peticiones pendientes, uno para *in* (*pendingTakeQueries*) y otro para *rd* (*pendingReadQueries*), y permanece ahí hasta que la función de *matching* encuentra una tupla que se ajuste al patrón dado. Cada vez que una operación expira esta clase debe llamar a una transición que retire la tupla del espacio de pendientes. Este trabajo es responsabilidad de las clases SleepWalker y Sleeper.

Ambas clases comparten una lista de datos pendientes y otra de operaciones pendientes. Todas las operaciones de lectura poseen un *id*, este *id* junto con el valor del timeout se almacenan en la lista de manera que el *timeout* más bajo ocupa la primera posición. Sleeper consulta el *timeout* más bajo y se bloquea durante el número de segundos correspondientes. Al despertar emplea este id para desbloquear la operación. La figura 23 muestra la relación entre ambos.

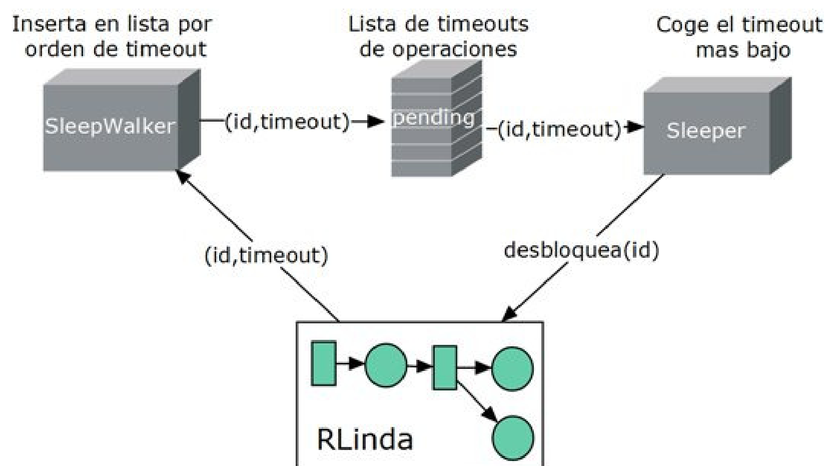


Fig. 20 Interacción de Sleeper y SleepWalker

SleepWalker

SleepWalker se crea en la red de TRLinda y es llamado cada vez que se produce una operación de lectura temporizada. Su misión es insertar la celda *id+timeout* en la lista de operaciones o datos pendientes. SleepWalker posee unas lista de datos y otra de operaciones, cada una gestionada por un Sleeper distinto. A su vez existen dos Sleeper, uno encargado de las operaciones *rd* u otro de las *in*. Ofrece un método *addToSleepWalker* que examina tupla y la asigna a una lista u otra.

Cómo se insertan operaciones en la lista de operaciones pendientes

Cuando un nuevo par id/timeout se inserta en la lista, el par que está en primera posición lleva un tiempo siendo gestionado por el Sleeper. Este tiempo debe de ser tenido en cuenta a la hora de decidir donde insertar el nuevo par, como muestra la figura 24.

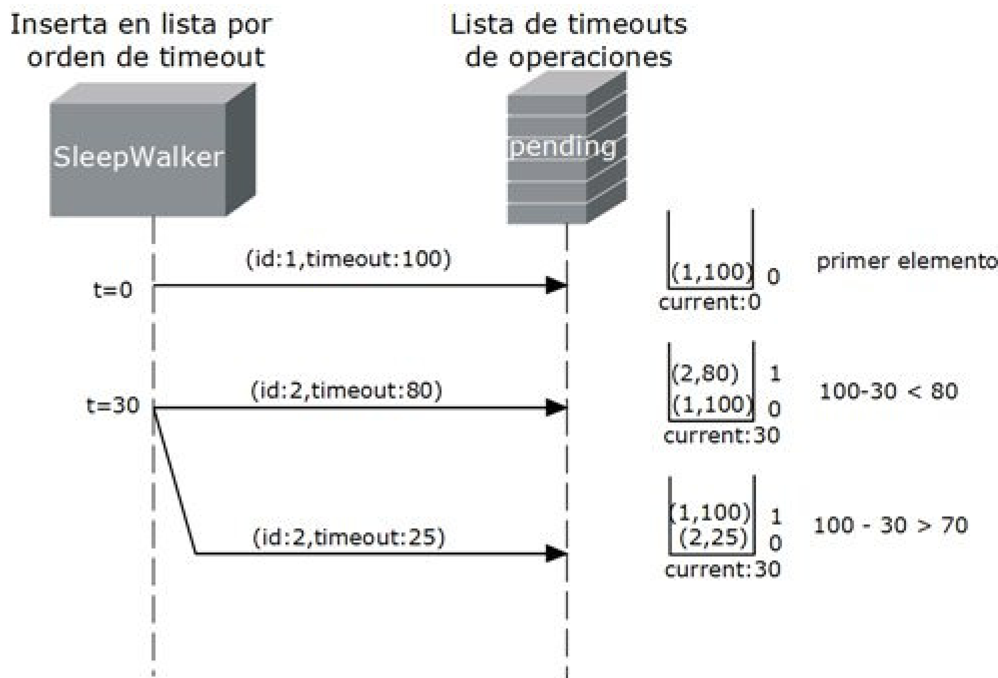


Fig. 21 Como se insertan elementos en la lista

En caso de *timeout:80* el *timeout* es mayor así que se añade en la posición dos. En cambio *timeout:25* tiene mayor prioridad (expirará antes que el que está actualmente en primer lugar) de manera que debe insertarse el primero de la lista.

Sleeper

Cada objeto Sleeper se ejecuta en un *thread* distinto y comparte una lista de pendientes son una instancia del objeto SleepWalker. Gestiona el timeout de la lista de pendientes y posee una instancia del coordinador de TRLinda para, una vez transcurra dicho *timeout* , desbloquear la operación de lectura correspondiente. Cuando llega una operación o dato con timeout menor que el que se está gestionando en ese momento, Sleeper prioriza al recién llegado pero recuerda el tiempo consumido hasta entonces por medio de las variables *delay* y *lastAwake*.

Sleeper implementa la interfaz Runnable le permite ser ejecutado en un *thread* aparte. Toda la gestión del *timeout* se produce en él método *run()*, que se describe la figura 25. El mecanismo se basa en que Sleeper lee el primer elemento de la lista que es el de menor *timeout*, realiza una llamada a *wait(timeout)* que lo dejará bloqueado durante *timeout* segundos. Al despertar, accede a TRLinda para desbloquear la operación correspondiente.

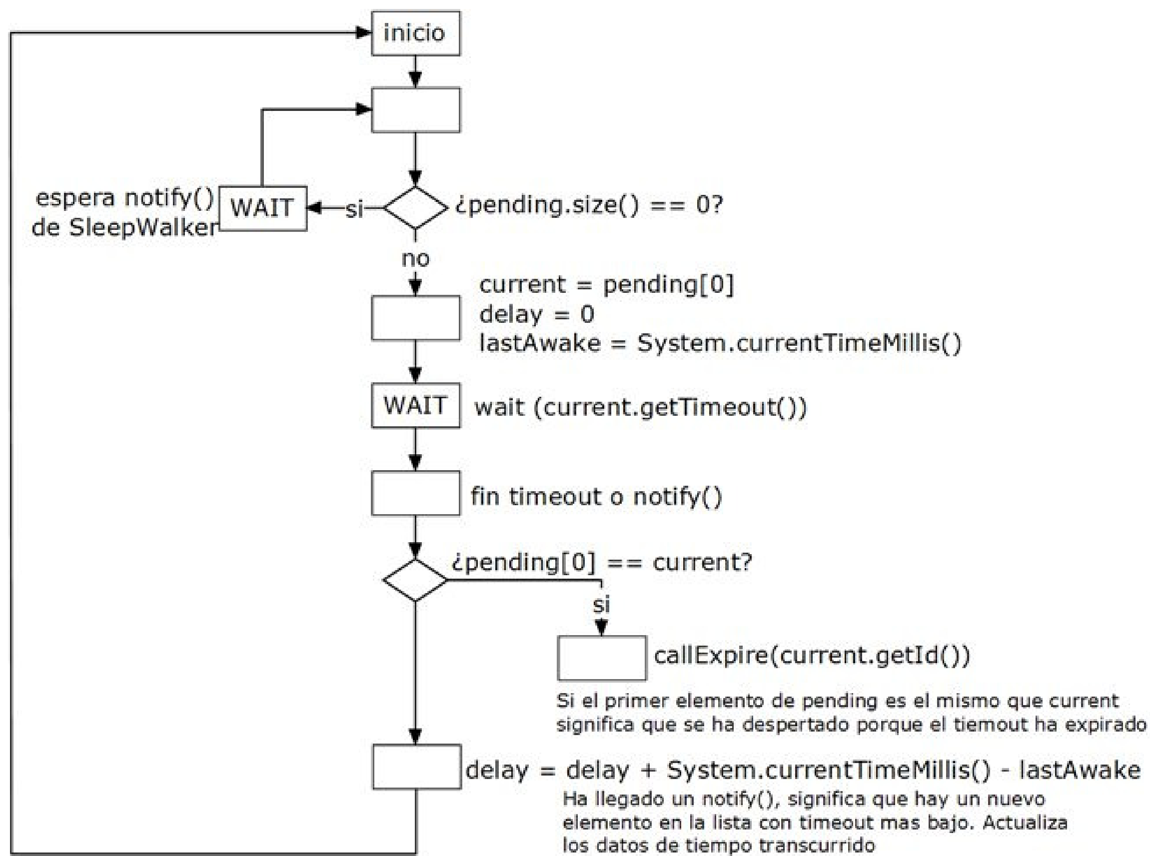


Fig. 22 Gestión de timeouts por parte de Sleeper

SleepWalker y Sleeper se comunican entre ellos por medio de la lista de pendientes y se sincronizan para evitar bloqueos y problemas de acceso simultáneo a la misma mediante llamadas *wait()* y *notify()*.

El coordinador y la red de TRLinda

Se realizan varias modificaciones en la red de TRLinda, de las cuales haremos hincapié en el sistema de desbloqueo de operaciones de lectura *rd*, análogo a *in*. El resto se presentan en el anexo extendido.

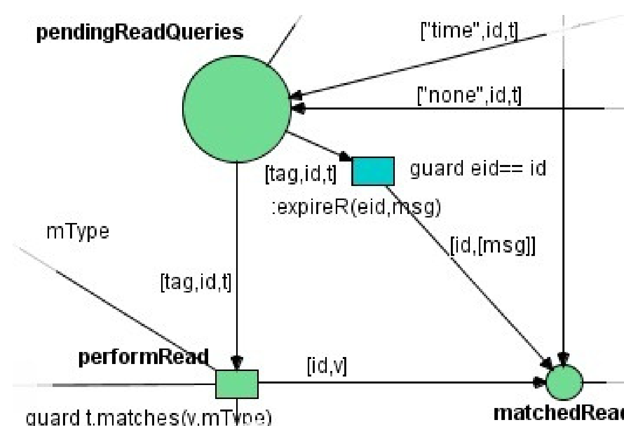


Fig. 23 Mecanismo de desbloqueo de operaciones

Una operación de lectura bloqueada espera en *pendingReadQueries*. Cuando su *timeout* expira Sleeper dispara la transición *expireR(eid,msg)* donde *eid* es el id de la operación a desbloquear. Solo se extrae la tupla con dicho id debido a la condición de guarda *guard id==eid*.
Al dispararse, la transición *expireR* deposita una tupla en el lugar *matchedRead* cómo si fuera un resultado normal de la operación de lectura.

La operación se desbloqueará (pues ha encontrado una tupla que se ajuste a su patrón, o eso piensa ella). La tupla devuelta está contenida en *msg* y será [“operation timeout”] o [“data timeout”] según hayan expirado los datos o la operación. De esta forma el proceso puede saber si se ha desbloqueado porque ha expirado el *timeout* o porque se le ha devuelto un resultado válido.

5.2.4 Aplicando la temporización a: las tuplas

Una tupla está temporizada si posee la cabecera [“TIMEOUT”,valor] donde *“TIMEOUT”* funciona como palabra reservada y *valor* es el tiempo en segundos que esa tupla permanecerá en el sistema antes de que expire. Temporizar una tupla es tan sencillo como el siguiente ejemplo:

[“tupla original”] pasa a estar temporizada así: [[“TIMEOUT”,15],[“tupla original”]] .

El valor de *timeout* de cero se comporta como si no estuviera temporizada.

Las función de *matching* se modifica para que compruebe y marque como expiradas las cuyo *timeout* ha expirado. La red de TRLinda posee una transición que elimina del espacio de tuplas las que han sido marcadas como expiradas.

Un patrón no deja de ser una tupla así que se temporiza de la misma forma. Los patrones temporizados sin embargo se tratan de una forma totalmente diferente ya que una vez expiran deben desbloquear al proceso que lanzó la operación de lectura. Estos timeouts se gestionan por objetos SleepWalker y Sleeper igual que las operaciones.

5.2.5 Aplicando temporización a: las operaciones

En RLinda el método *out(tuple)* provoca que la tupla *tuple* se escriba en el espacio de tuplas. No es bloqueante, por lo que añadir tiempo a la operación carece de sentido. Luego la operación de escritura temporizada es equivalente a una operación *out(tuple)* normal cuya tupla está temporizada.

Las operaciones de lectura de TRLinda son por definición bloqueantes. Al aplicar temporización a estas operaciones estamos diciendo que si no se encuentra ninguna tupla que corresponda con el patrón, pasado un tiempo *t* la llamada se desbloquee.

Una operación expirada devuelve como resultado al proceso que la ejecutó (y que permanecía bloqueado esperando un resultado) una tupla indicativa de lo sucedido: [“operation timeout”].

En el caso de operaciones de lectura existe la posibilidad de temporizar tanto operaciones como datos. Un dato representa información y tiene más importancia en el sistema que una simple operación, de forma que se considera preferente su timeout. Cuando expira el timeout del patrón dado por la operación de lectura, la tupla resultado devuelta será: [“data timeout”]

5.2.6 Traza de interacción SleepWalker – Sleeper

En la figura 28 se observan las interacciones, a través de la lista de pendientes, entre SleepWalker y Sleeper. Este primero inserta en orden los timeout en pendientes y notifica a Sleeper si ha insertado un *timeout* menor del tiempo que queda del que está gestionando actualmente.

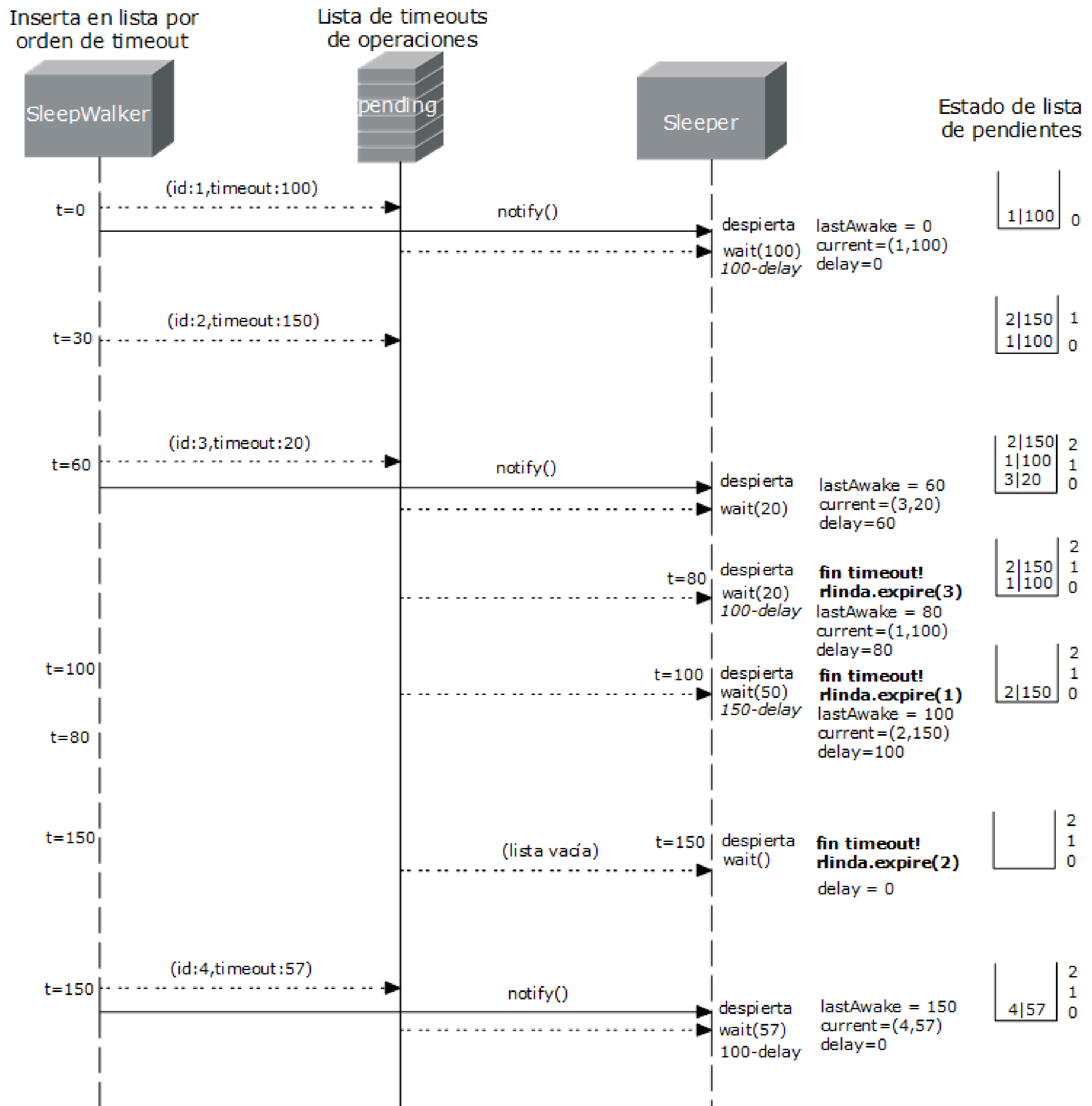


Fig. 24 Gestion de tiemouts de Sleeper y SleepWalker

La clase Sleeper gestiona el *timeout* del primer elemento de la lista de pendientes, que tiene el *timeout* más pequeño, mediante de *wait()* de *timeout* segundos. Cuando se despierta comprueba si lo ha hecho porque el timeout ha pasado o si ha sido despertado por SleepWalker. En caso primero Sleeper retirará el primer elemento de la lista, accederá a TRLinda para desbloquear la operación y empezará de nuevo con el siguiente elemento de la lista.

En el segundo caso significa que ha llegado un *timeout* con mayor prioridad (más pequeño y que ya ha sido puesto en primer lugar de la lista). Sleeper se bloqueará otra vez con *wait()* pero con el valor de este nuevo *timeout*.

Un caso problemático

Mientras Sleeper se despierta, accede a la lista de pendientes y lanza una nueva llamada a *wait()* con el valor de *timeout* que corresponda pasan unos pocos milisegundos. No hay manera de tener estos pocos milisegundos y puede producirse la situación en que los tiempos de los diferentes *timeouts* estén tan ajustados que esa pérdida de segundos suponga que un *timeout* ya ha expirado cuando entra en el Sleeper y lanza *wait()*.

En este caso lo que sucede es que *wait(tiempo)* se está llamando con un valor de *tiempo* negativo y genera una excepción. La solución es simple, si ocurre dicha excepción se captura y se emplea la referencia a RLinda para desbloquear la transición consecuente. La ejecución podrá continuar normalmente.

5.3 WS- Persistent Temporized RLinda

Una vez presentadas las dos capas que componen este PFC, se agrupan todas bajo una capa de presentación que ofrece todas las operaciones disponibles desde el exterior. Esta clase tiene por nombre PTRLinda e implementa la interfaz PTRLindaRemote que permite instanciarla en RMI.

La capa de persistencia y la de temporización se han desarrollado de forma modular para que puedan usarse de forma aislada, pero la herramienta final se presenta como una combinación de ambas, una que aporta persistencia y otra los conceptos de temporización. Visto desde la perspectiva del cliente Web, así se ofrece el sistema WS-PTRLinda.

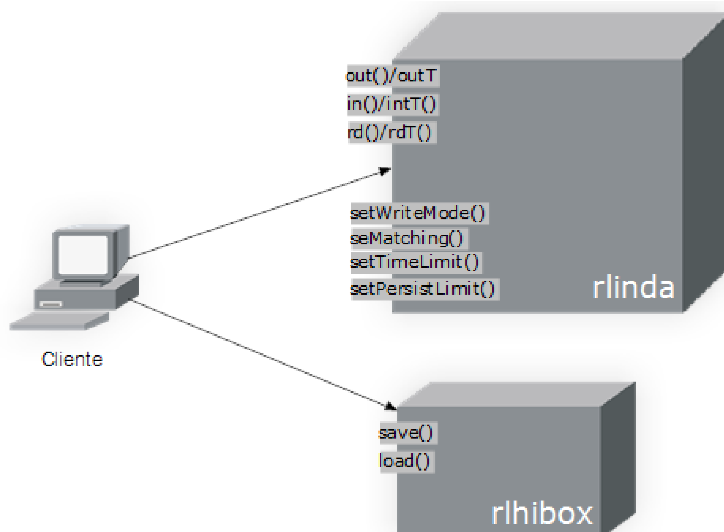


Fig. 25 Visión de los servicios Web desplegados por WS-PTRLinda

El servicio principal tiene ahora una referencia a la capa de temporización y ofrece las nuevas operaciones temporizadas. AXIS2 no permite sobrecarga de manera que se ofrecen con nombres distintos (por ejemplo *out()* y *outT()*). El servicio encargado de proporcionar las operaciones de copia de seguridad permanece intacto y sigue poseyendo una referencia a la capa de persistencia.

Kapfhammer propuso en [23] un *framework* para prestaciones de espacios de tuplas, referido como *Space bEnchmarking and TesTing moduLEs* (SETTLE) [23]. El *framework* permitió realizar la medición de JavaSpaces con el modelo descrito en [22]. El mismo *benchmark* se aplicó para comparar los resultados entre JavaSpaces y RLinda en [3].

En esta sección se comparan los resultados de RLinda con los obtenidos por la implementación de WS-PTRLinda. Para poder ejecutar el *benchmark* con garantías, se han vuelto a ejecutar los mismos experimentos utilizando RLinda sobre el mismo entorno hardware que WS-PTRLinda.

En los experimentos un conjunto de clientes Java acceden a un servidor RLinda/WS-PTRLinda. Cada cliente realiza una fase de precalentamiento para definir algunas variables locales y preparar los parámetros de ejecución del *benchmark*. A continuación, itera 1000 veces el siguiente ciclo: ejecuta una operación out, hace una pausa de un tiempo aleatorio *T_delay* en el rango de [200,250] ms) y, por último, obtiene la misma tupla mediante una operación in. Una vez ha finalizado el ciclo, el cliente termina ordenadamente. Una justificación detallada de todos los parámetro utilizados en el *benchmark* puede encontrarse en [23] mientras que la Figura 26 resume el experimento ejecutado. Para estudiar la influencia del tamaño de la tupla se han utilizado los mismos objetos que en [23,] objetos NullEntry (356 bytes), objetos StringEntry (503 bytes), objetos DoubleArrEntry (1031 bytes) y objetos FileEntry (3493 bytes). La función de emparejamiento aplicada corresponde a la función de *strong-matching*, lo que significa que dos tuplas son iguales cuando lo es la correspondencia de todos y cada uno de sus elementos.

En el caso de WS-PTRLinda se ha buscado medir la sobrecarga que supone la comprobación de la temporización de los datos, por lo que todas las tuplas se han escrito con un valor de temporización de 10000000 ms, lo que garantiza que la operación terminará antes, pero que la temporización debe comprobarse.

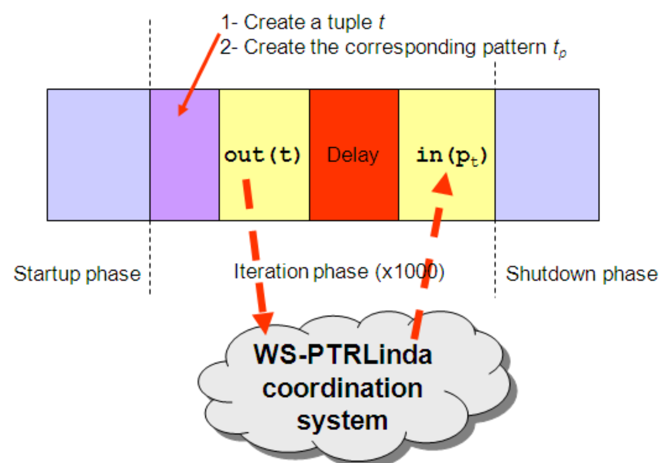


Fig. 26 Configuración del experimento según el propuesto en SETTLE

En el *benchmark*, q clientes se ejecutaron en q nodos distintos utilizando el software Condor para High Throughput.

Computing(HTC) sobre un clúster de 22 nodos más uno adicional dedicado a albergar el servidor RLinda. Concretamente, el clúster utilizado ha sido el del Instituto de Investigación en Ingeniería de Aragón, llamado Hermes. La configuración de cada nodo es una estación de trabajo con un procesador Dual Xeon 2.8Ghz montado con 2Gb de RAM y una arquitectura de 32 bits. El

subsistema de E/S está compuesto por un disco SATA por cada estación, y se cuenta con una Red interna GigaByte Ethernet en conexión directa al armario de comunicaciones del edificio. El sistema operativo utilizado fue Scientific Linux 5.4 (reempaquetado de Red Hat). La máquina virtual de Java 1.6 se ejecutó con la configuración por defecto y el servidor RLinda fue habilitado para utilizar la máxima cantidad posible de memoria física.

En el experimento se ha medido el *throughput* como número medio de operaciones *in/out* ejecutadas por segundo), tanto para RLinda como para WS-PTRLinda.

Cabe resaltar que todos los parámetros se miden en el lado del servidor para evitar los retrasos debidos a la latencia de la red. De esta forma, los resultados se han obtenido en un escenario similar al propuesto en [23].

La Figura 27 ofrece una representación gráfica del *throughput* RLinda en relación al número de clientes concurrentes q. Cabe destacar que los resultados obtenidos son mejores que los descritos en [3], principalmente debido a la mejora en el hardware utilizado para realizar los experimentos. Los resultados muestran que la operación más costosa siempre es una operación out o in utilizando un objeto del tipo FileEntry, ya que es el objeto más pesado y requiere una sobrecarga adicional de CPU para su proceso. Las mejores medidas se obtienen cuando q pertenece al intervalo [7,12] (excepto cuando se están procesando los objetos FileEntry, que alcanzan los valores máximos en el rango [10,13]). La existencia de un umbral a partir del cual las prestaciones comienzan a decrementarse ya fue descrito previamente para la implementación de JavaSpaces por Zorman en [22] Para valores más allá de 42 clientes concurrentes el tiempo de CPU fue tan elevado que muchas conexiones se cayeron, por lo que no tiene sentido mostrar valores a partir de este punto.

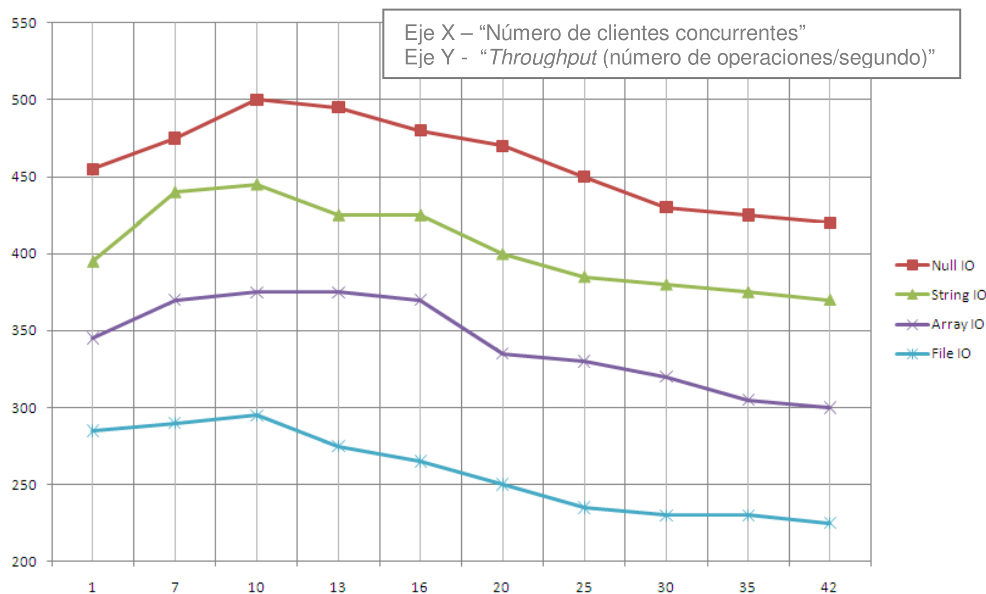


Fig. 27 *Throughput* para RLinda

La Figura 28 muestra los resultados obtenidos por la implementación de WS-PTRLinda. Como puede observarse, la implementación es mucho más estable y muestra el mayor *throughput* al principio, decrementando conforme aumentamos el número de clientes concurrentes atacando el espacio de tuplas. Este es el comportamiento normal, ya que en la implementación de WS-PTRLinda se ha arreglado el cuello de botella que provocaba que en la implementación de RLinda los resultados óptimos no se encontrasen al decrementar el número de nodos accediendo al espacio. Hay que destacar que las prestaciones son ligeramente peores que las obtenidas para RLinda, si bien se está realizando la temporización en este caso. Por tanto, los resultados muestran que la sobrecarga debida

a la comprobación de los datos temporizados es mínima, y que el sistema sigue ofreciendo unas excelentes prestaciones.

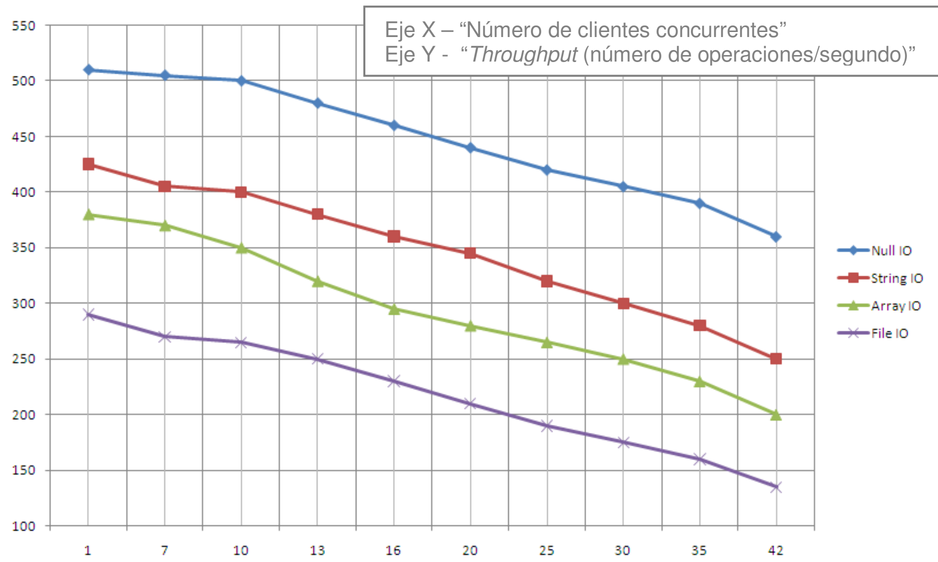


Fig. 28 *Throughput* para WS-PTRLinda

Finalmente, la Figura 29 muestra la comparativa entre los experimentos que han reportado las mejores y las peores medidas para los experimentos entre RLinda y WS-PTRLinda. Como puede apreciarse, inicialmente se consiguen unas mejores medidas con la nueva implementación, mientras que conforme aumenta la carga en el sistema, la sobrecarga de comprobar la temporización se hace notar. Sin embargo, no se aprecia un cambio notable en la tendencia de las curvas para WS-PTRLinda, por lo que podemos concluir que la implementación resulta muy eficiente para datos pequeños, mientras que es un poco más pesada para datos grandes (como puede apreciarse en el *benchmark* que utiliza los datos del tipo File IO).

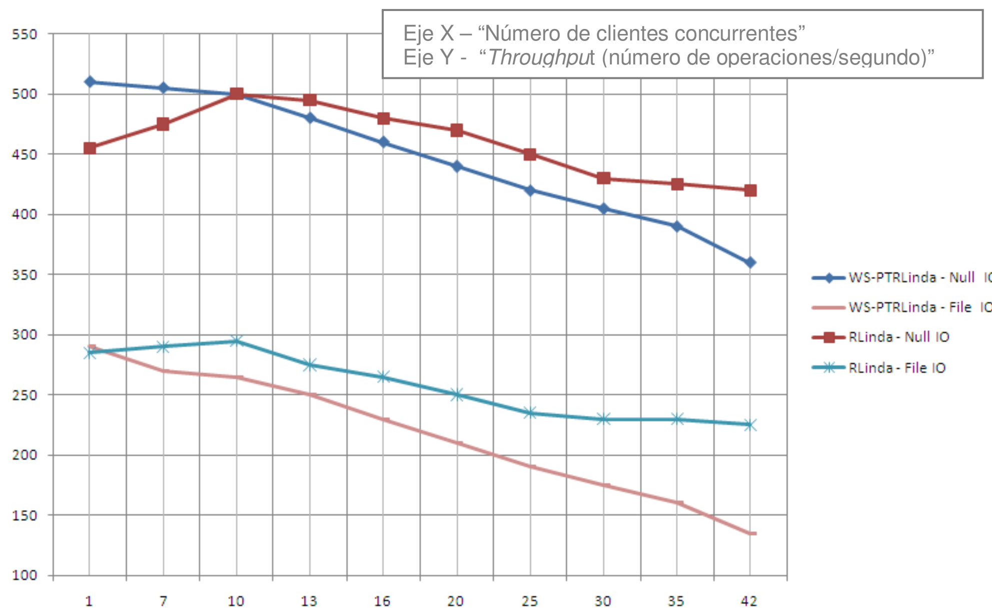


Fig. 29 *Comparativa del throughput de Rlinda y WS-PTRLinda en el mejor y en el peor caso*

Capítulo 7 | **Caso de uso: una aplicación de descarga P2P**

Una vez explicado WS-PTRLinda y vista la evaluación de su rendimiento, se va a mostrar un caso de uso. La aplicación realizada consiste en un programa de intercambio de archivos P2P que emplea el espacio de tuplas ofrecido por WS-PTRLinda para sincronizar a todos sus clientes para que puedan intercambiarse archivos.

7.1 Un nuevo cliente P2P: FLARLARdownloader

FLARLARdownloader es un programa cliente que se ejecuta en la máquina del usuario y que accede WS-PTRLinda a través de sus servicios SOAP.
La aplicación permite, por un lado, compartir archivos con otros usuarios. Estos archivos se dividen en partes que son descargadas individualmente por los diferentes clientes. Por otro lado, descargar archivos de otros usuarios.

Desde el punto de vista de este PFC la parte más interesante reside en la coordinación de los distintos clientes, que se realiza mediante la escritura y consumo de tuplas en el espacio de tuplas de WS-PTRLinda.

La aplicación saca partido de las operaciones temporizadas de WS-PTRLinda en dos formas:

- Evitando bloqueos. Las operaciones de consulta de datos en el espacio de tuplas poseen un tiempo límite de espera que evita que el cliente se quede bloqueado indefinidamente.
- Permite publicar archivos durante un tiempo establecido por el usuario.

Veamos como es el modelo de coordinación.

7.2 Modelo de coordinación

Con WS-PTRLinda coordinar un montón de clientes que quieren compartir información resulta muy sencillo. Para la aplicación FLARLARdownloader un archivo se divide en partes. Las partes se descargan individualmente y una vez que se han descargado todas se juntan para generar el archivo.
Las partes no tienen por qué ser descargadas del mismo cliente.

En nuestro caso el modelo se compone de dos tipos de tuplas:

- <u>Tuplas de archivo</u>. Tienen información general sobre un archivo completo: id, número de partes que lo compone y tamaño.
- <u>Tuplas de parte</u>. Identifican una parte de un archivo: poseen id (que será el mismo que el del archivo que forman parte), número de parte y dirección del cliente la tiene.

Cuando un cliente comparte un archivo, introduce una tupla de archivo en WS-PTRLinda y una tupla de parte por cada una de las partes que componen el archivo.

Un cliente que desee descargarse un archivo, se conecta a RLinda para consultar la tupla de archivo y ver cuántas partes tiene. Luego accede a RLinda para obtener tuplas de parte, que contienen la dirección del cliente que las posee, y se conecta al cliente para descargarse dicha parte. Y así hasta que completa el archivo. La figura 30 es una traza explicativa.

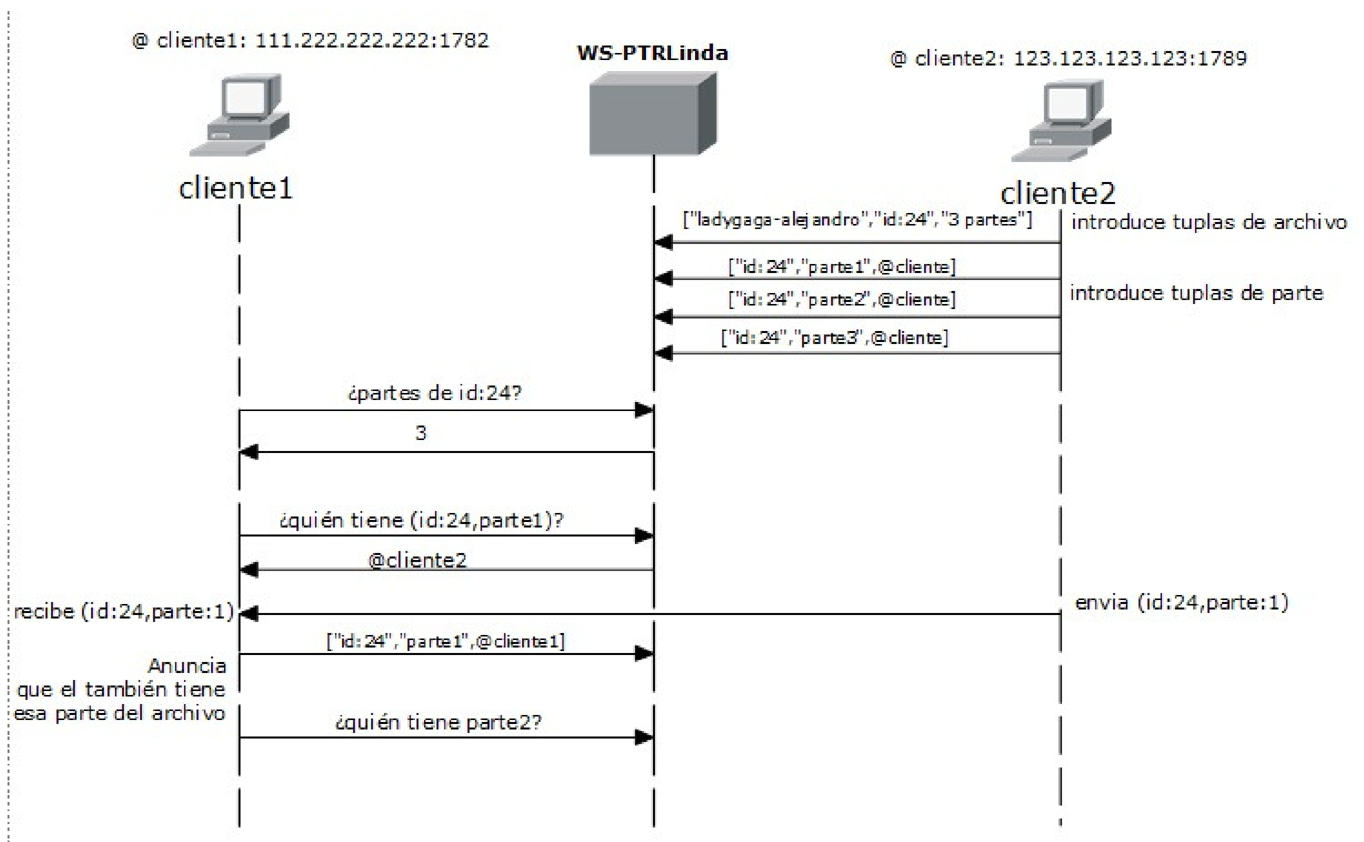


Fig. 30 Traza de coordinación de clientes

De esta traza de ejecución se destacan dos cosas. Todos los clientes que poseen partes de un archivo han depositado una tupla de parte. Al preguntarle a WS-PTRLinda quién tiene una parte cada vez, se consigue que me devuelva un cliente cualquiera de manera que estoy descargando cada parte de un cliente distinto.

Cuando un cliente ha descargado una parte automáticamente introduce una tupla de parte en WS-PTRLinda. De esta manera, aunque el cliente no posee el archivo completo sí que puede compartir una parte de él.

Desconexión de clientes

Cuando un cliente, por cualquier motivo, se desconecta no es necesario borrar las tuplas. Cuando un cliente que ha recibido una tupla de parte no consigue conectar con el propietario de dicha parte, remueve la tupla del sistema e intenta conseguir otra. De esta forma el espacio de tuplas se va actualizando bajo la demanda de los clientes.

Archivos temporales

Empleando las operaciones de temporización un cliente puede elegir el tiempo máximo que quiere que sus archivos sean compartidos. Una vez pase este tiempo, WS-PTRLinda expira y elimina las tuplas correspondientes de manera que otros clientes nunca podrán acceder a ella y por no tanto no podrán intentar descargarlas (de ese cliente al menos).

7.3 Descarga de tuplas

Como hemos comentado el verdadero reto de la aplicación es comunicar los clientes y sincronizarlos para que compartan archivos sin que estos tengan que estar centralizados.

Para descargar las diferentes partes, los clientes emplean RMI. No es la tecnología que se emplearía si la aplicación fuera a salir al mercado, pero nos basta para verificar el funcionamiento de una forma sencilla. Cada parte de un archivo se publica en el registro RMI del cliente y puede ser instanciada por un cliente en otra máquina. La parte posee un método de descarga que envía el archivo al cliente.

7.4 GUI (Interfaz gráfica)

FLARLARdownloader presenta una ventana principal dividida en tres partes, una de descargas dónde se introduce el id de un archivo para empezar a descargarlo, una central donde aparecen mensajes informativos del estado del programa y una tercera donde se pueden seleccionar archivos para compartirlos. A la hora de seleccionar un archivo para compartir aparece una ventana con un explorador de archivos para facilitar esta tarea. Las figura 31 es una muestra de la interfaz gráfica.

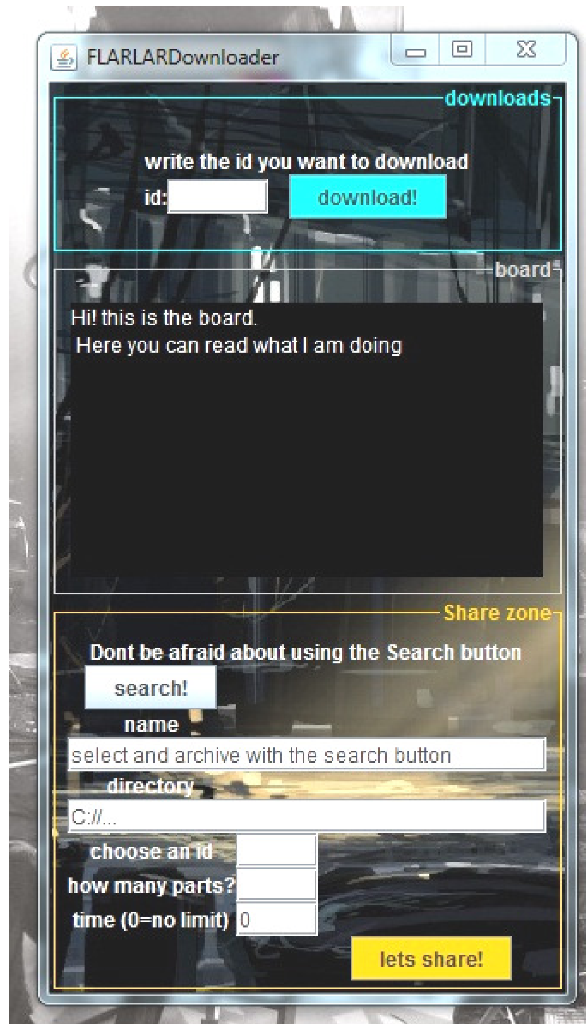


Fig. 31 ventana principal

Capítulo 8 | **Conclusiones**

El objetivo principal de este PFC era el diseño y desarrollo de una capa de persistencia que aporte al sistema la tolerancia a fallos por un lado y una capa de temporización que añada el concepto de tiempo a las operaciones y datos de RLinda por otro. Este objetivo se ha cumplido con éxito dejando atrás horas de búsquedas de información, diseños ante un folio en blanco y momentos más o menos dulces de programación.

8.1 Conclusiones a nivel técnico

Este PFC en realidad forma parte de una plataforma de más alto nivel, pero aun así creo que por sí solo es una buena muestra de un sistema listo para ser desplegado en un escenario real y competitivo como es internet.

WS-PTRLinda permite a sus clientes el acceso mediante la tecnología de los servicios Web REST y SOAP abarcando así un amplio espectro de posibles usuarios del sistema además de seguir la tendencia de los grandes proveedores de la Web 2.0 que están migrando (o ya lo han hecho) a REST.
Garantizar la tolerancia a fallos es una característica muy deseable en un sistema que va a situarse en el centro de la comunicación, este proyecto no solo aporta esto a RLinda sino que ofrece dos formas distintas de definir persistencia para que sea más flexible cuya configuración se realiza desde el exterior.

La capa de persistencia se sustenta sobre la tecnología de Hibernate que simplifica enormemente la comunicación con la base de datos. De cara a un proyecto futuro que parta desde esta capa de persistencia resultará fácil variar parámetros de la base de datos sin necesidad de acceso al código, así como sustituir la base de datos por otra distinta, por no decir que a la persona que venga le resultará mucho más sencillo entender el código.

Desde mi punto de vista el mundo de la informática ofrece unas posibilidades apasionantes. Puede que ya existan formas de hacer las cosas pero siempre se puede dar un paso más hacia algo nuevo que facilite todo. Para mi RLinda forma parte de eso nuevo, de una forma de coordinar y comunicar procesos que aun tiene que crecer hasta que la mayor parte de la comunidad la considere. La capa de temporización desarrollada añade a las operaciones ya existentes un concepto nuevo con montones de posibilidades. Una prueba es el caso de aplicación realizado, una aplicación P2P en el que el 95% del tiempo se ha invertido en luchar contra la interfaz gráfica y el envío de archivos, pero en la que diseñar la comunicación y la sincronización de los clientes necesitó apenas dos horas.

RLinda proporcionaba rendimientos superiores a la competencia directa JavaSpaces. Ahora, como puede verse en las gráficas de evaluación, no solo incorpora nuevas funcionalidades con una mínima sobrecarga sino que las horas invertidas en buscar y desarrollar una solución a la comunicación entre el nodo principal y el proveedor de acceso Web han conseguido que WS-PTRLinda mejore en aspectos de concurrencia de clientes.

Los datos en RLinda se codifican en XML, formato totalmente extendido y usado para intercambiar información. Después de este PFC se cuenta con un mecanismo de verificar que un dato XML tiene la sintaxis correcta de un dato de WS-PTRLinda. Este mecanismo son esquemas XMLSchema y modificar los *tags* aceptados es tan sencillo como cambiar un fichero de texto.
Las nuevas capas son totalmente modulares, de forma que puede usarse una, otra o ninguna. Considero que este proyecto es un buen punto de partida hacia nuevos desarrollos y mejoras de un

sistema, RLinda, que tiene enormes posibilidades en el futuro de hacerse un hueco en el inmenso mar de información de internet.

8.2 Trabajo futuro

A lo largo del desarrollo te vienen a la cabeza cosas que te gustaría añadir o nuevas posibilidades que no se relacionan directamente con tu proyecto. Otras si guardan relación pero la fecha está demasiado avanzada como para introducir según qué cambios en el sistema. Estas son algunas posibilidades de trabajo futuro.

Desarrollar un mecanismo de balanceo de persistencia que monitorize la carga del sistema de manera que pueda cambiar en un momento dado de modo de persistencia para mantener una relación de garantía de persistencia/rendimiento del sistema.

Permitir restaurar copias de seguridad de la base de datos "en caliente", es decir, restaurar espacio de tuplas y base de datos a un punto anterior en el tiempo mientras en sistema está funcionando.

Modificar la capa de persistencia para que cambiar la base de datos sea totalmente "*plug & play*". En WS-PTRLinda el único código que hay que modificar para sustituir la base de datos consiste en parte de las funciones de creación y restauración de la copia de seguridad.

Desarrollar un sistema de autentificación de usuarios de manera que algunas operaciones solo puedan ser realizadas por determinados tipos de usuario.

Proporcionar un mecanismo de envío y recepción de copias de seguridad a través de REST.

8.3 Conclusiones personales

Siempre me ha llamado la ingeniería software orientada al mundo online. Históricamente, cuando la gente imaginaba el futuro veía robots haciendo su trabajo pero nadie podía imaginar que existiría un mundo totalmente paralelo que nos interconectaría a este nivel. Mientras el mundo real tiene las limitaciones de la física el mundo virtual te permite crear casi cualquier cosa. Con este pensamiento en la cabeza siempre me he sentido atraído por saber cómo funciona todo esto, sobre todo a mayor nivel que las redes físicas. Me introduje en el mundo de los servicios Web en mi año Erasmus en Italia, en la Università da Torino, y la vuelta buscaba un proyecto enfocado en este campo. La propuesta de extender RLinda me atrapó en cuanto Javier me explicó qué era eso de RLinda y para qué servía.

Este proyecto me ha aportado muchas cosas desde el punto de vista personal como de mi forma de trabajar como ingeniero. Me encanta la fase inicial de diseño sobre papel así como empezar a transformar esas líneas de lápiz en líneas de código. Aun así solía lanzarme demasiado pronto a hacer cosas que necesitaban algo más de preparación. Este PFC me ha aportado poco a poco, a puro maldecir no haber hecho más trabajo previo, la costumbre de diseñar hasta el más mínimo detalle y el más extraño caso (alguno siempre se escapa) antes de pasar a la implementación, llegando a sorprenderme a mi mismo cuando añadía las últimas funcionalidades de la capa de temporización.
El PFC me ha proporcionado un buen conjunto de buenos hábitos en documentación, copias de seguridad, metodología de trabajo y ojo clínico para identificar tutoriales defectuosos antes de empezar a seguirlos.

A lo largo de estos meses mi productividad delante del ordenador ha ido aumentando hasta sorprenderme a mí mismo, me siento especialmente orgulloso de las últimas semanas de

implementación y el inicio de la documentación, cuando cada día avanzaba considerablemente a WS-PTRLinda.

WS-PTRLinda es un sistema complejo y amplio, muy diferente de cualquier cosa a la que me haya enfrentado antes. Ahora sé en qué consiste un proyecto maduro y como es realizarlo y enfrentarse a menudo a la frustración y a querer lanzar tu equipo por la ventana. Al final, cuando todo funciona, merece la pena.

Capítulo 9 | **Referencias**

[1] D. Gelernter (1985). Generative communication in Linda. ACM Transactions on Programming Languages and Systems, 7(1):80–121.

[2] D. Gelernter (1989). Parallel Architectures and Languages Europe –PARLE '89. Eindhoven, The Netherlands, June 12-16, 1989, volume 366 of Lecture Notes in Computer Science, chapter Multiple tuple spaces in Linda, pages 20–27. Springer Verlag.

[3] Fabra, J., Álvarez, P., Bañares, J.A., Ezpeleta, J.: RLinda: A Petri Net Based Implementation of the Linda Coordination Paradigm for Web Services Interactions. Lecture notes in Computer Science: E-Commerce and Web Technologies, Volume 4082, 2006, pages 183-192. 2006.

[4] Kummer, O., Wienberg, F., Duvigneau, M., Schumacher, J., K¨ohler, M., Moldt, D., R¨olke, H., and Valk, R. (2004). 25th International Conference on Application and Theory of Petri Nets – ICATPN 2004. Bologna, Italy, June 2004, volume 3099 of Lecture Notes in Computer Science, chapter An Extensible Editor and Simulation Engine for Petri Nets: Renew, pages 484–493. Springer Verlag.

[5] Alberto Olmo garcía, Francisco Javier Fabra Caro. Despliegue de la plataforma RLinda a través de un servicio web seguro basado en el estándar WS-Security, Febrero 2009, PFC en la Universidad de Zaragoza.

[6] Sun Microsystems, Inc. (2000). JavaSpaces Service Specification. Technical report, Sun Microsystems.

[7] GigaSpaces Technologies (2000). GigaSpaces. Technical report, Sun Microsystems.
[8] AlphaWorks - TSpaces (2003). www.alphaworks.ibm.com/tech/tspaces.
[9] Kielmann, T. (1997). Objective Linda: A Coordination Model for Object-Oriented Parallel Programming. PhD thesis, Dept. of Electrical Engineering and Computer Science, University of Siegen, Germany.

[10] P. Broadbery and K. Playford (1991). Using object-oriented mechanisms to describe Linda. Technical report, Edinburgh Parallel Computing Centre.

[11] Project 'jxtaspaces' (2004). Available at http://jxtaspaces.jxta.org/.

[12] Tolksdorf, R. and Glaubitz, D. (2001). 9th International Conference on Cooperative Information Systems. Trento, Italy, September 5-7, 2001, volume 2172 of Lecture Notes in Computer Science, chapter Coordinating Web-Based Systems with Documents in XMLSpaces, pages 356–370. Springer Verlag.

[13] Tolksdorf, R. (2002). Workspaces: a Web-based Workflow Management System. IEEE Internet Computing, 6(5):18–26.

[14] Emre Sarig¨ol,Oriana Riva and Gustavo Alonso. A Tuple Space for Social Networking on Mobile Phones.

[15] Renew User-Guide, Olaf Kummer, Frank Wienberg, Michael Duvigneau, Lawrence Cabac. University of Hamburg, August 28, 2009.

[16] J. Fabra, P. Álvarez, and J. Ezpeleta. DRLinda: A Distributed Message Broker For Collaborative Interactions Among Business Processes. In 8th International Conference on Electronic Commerce and Web Technologies - EC-Web'07, volume 4655 of LNCS, pages 212- 221. Springer Verlag, Sept 2007.

[18] Eric Freeman, Susanne Hupfer, and Ken Arnold, "JavaSpaces Principles, Patterns, and Practice".

[20] N. Davies, S.P. Wade, A. Friday and G.S. Blair. "Limbo: A tuple space based platform for adaptive mobile applications", Distributed Multimedia Research Group, Computing Department, Lancaster University.

[21] Lyndon J.B. Nixon, Elena Simperl, Reto Krummenacher, Francisco Martin-Recuerda," Tuplespace-based computing for the Semantic Web: a survey of the state-of-the-art".

[22] B. Zorman, G. M. Kapfhammer, and R. S. Roos (2002). Proceeedings of the 8th International Conference on Parallel and Distributed Processing Techniques and Applications –PDPTA '02. Las Vegas, Nevada, USA, June 24 - 27, 2002. Volume 3, chapter Creation and analysis of a JavaSpace-based genetic algorithm, pages 1107–1112. CSREA Press.

[23] D. Fiedler, K. Walcott, T. Richardson, G. M. Kapfhammer, A. Amer, and P. K. Chrysanthis (2005). Towards the Measurement of Tuple Space Performance. ACM SIGMETRICS Performance Evaluation Review, 33(3):51–62.

Referencias consultadas en internet

[17] Información sobre JavaSpaces , http://www.dcc.uchile.cl/~secastro/60f/JavaSpaces.htm.

[19] Tom White, "How to build a ComputeFarm", 21-04- 05, http://today.java.net/pub/a/today/2005/04/21/farm.html.

[24] Various time classes, http://www.ensta.fr/~diam/java/online/notes-java/other/10time/01time.html

[25] TSpaces, IBM, http://www.almaden.ibm.com/cs/TSpaces/.

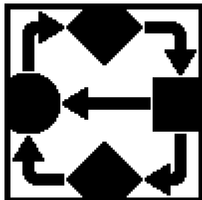
ANEXO 1 | **Tecnologías Utilizadas**

Este capítulo pretende nombrar y explicar para qué se ha utilizado cada una de las tecnologías y programas en la realización de este proyecto fin de carrera.

Renew2.2

Simulador de redes de Petri[] de alto nivel basado en lenguaje de programación Java creado en el departamento de informática de la universidad de Hamburgo. Dichas redes de Petri están basadas en Reference Nets.
Provee una herramienta gráfica que hace muy sencillo su modelado. Las Reference Nets son clases Java y pueden ser accedidas desde otras clases.

Renew viene con código fuente incluido de manera que puede ser extendido en la manera en que sea necesario.



Web: http://www.renew.de/

Hibernate 3.5.3

Una herramienta de Mapeo objeto-relacional para la plataforma Java que facilita el mapeo de atributos entre una base de datos relacional y el modelo de objetos. El objetivo de Hibernate es relevar al desarrollador del 95% de las tareas de persistencia de datos comunes en cualquier aplicación.
Soporta multitud de bases de datos del mercado y ahorra trabajo a la hora de consultar manuales específicos de bases de datos o de driver JDBC. Hibernate se configura mediante ficheros de configuración externos que le aportan mucha versatilidad.

En este PFC se encarga de la comunicación con la base de datos en las operaciones de persistencia.

Web: http://www.hibernate.org

MySQL

Uno de los sistemas de gestión de base de datos mas populares. Es propiedad de Sun Microsystems Gestiona el acceso a una base de datos relacional, multihilo y multiusuario. Soporta varios lenguajes de programación incluyendo C, C++, C#, Pascal, Lisp, Perl PHP, Javam Ruby y otros.

MySQL es muy utilizado en aplicaciones Web, en plataformas (Linux/Windows-Apache-MySQL-PHP/Perl/Python), y por herramientas de seguimiento de errores.

En este PFC Almacena una copia del espacio de tuplas de RLinda.

Web: http://www.mysql.com/

Java

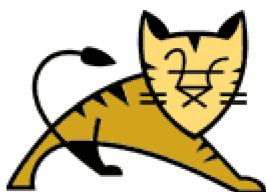
Java es un lenguaje de programación orientado a objetos, desarrollado por Sun Microsystems a principios de los años 90. El lenguaje en sí mismo toma mucha de su sintaxis de C y C++, pero tiene un modelo de objetos más simple y elimina herramientas de bajo nivel. La implementación original y de referencia del compilador, la máquina virtual y las bibliotecas de clases de Java fueron desarrollados por Sun Microsystems en 1995.

Web: http://www.java.com

Apache Tomcat

Tomcat es un servidor Web con soporte de *servlets* y JSPs. Tomcat puede funcionar como servidor Web por sí mismo. En sus inicios existió la percepción de que el uso de Tomcat de forma autónoma era sólo recomendable para entornos de desarrollo y entornos con requisitos mínimos de velocidad y gestión de transacciones. Hoy en día ya no existe esa percepción y Tomcat es usado como servidor Web autónomo en entornos con alto nivel de tráfico y alta disponibilidad.
Dado que Tomcat fue escrito en Java, funciona en cualquier sistema operativo que disponga de la máquina virtual Java.

En este PFC se utiliza como contenedor de aplicaciones para desplegar y ejecutar AXIS2 como una aplicación Web (Web-App).



Web: http://tomcat.apache.org/

Apache AXIS 2

Framework para la creación, ejecución y despliegue de servicios Web disponible en lenguaje C y Java. Soporta SOAP 1.1 , SOAP 1.2 así como REST. Algunas de sus características son su velocidad, uso de poca memoria, un modelo de objetos propio (AXIOM) y la posibilidad de desplegar servicios en caliente (en plena ejecución, solo hace falta copiar el archivo del servicio Web en la carpeta correspondiente de AXIS2 para que despliegue automáticamente.)

En este PFC se utiliza para crear y desplegar los servicios Web.

Web: http://ws.apache.org/axis2/

NetBeans

Un entorno de desarrollo visual de código abierto para aplicaciones programadas mediante Java. NetBeans es un proyecto de código abierto de gran éxito con una gran base de usuarios, una comunidad en constante crecimiento, y con cerca de 100 socios en todo el mundo. Sun MicroSystems fundó el proyecto de código abierto NetBeans en junio de 2000 y continúa siendo el patrocinador principal de los proyectos.

La Plataforma NetBeans es una base modular y extensible usada como una estructura de integración para crear aplicaciones de escritorio grandes. Empresas independientes asociadas, especializadas en desarrollo de software, proporcionan extensiones adicionales que se integran fácilmente en la plataforma y que pueden también utilizarse para desarrollar sus propias herramientas y soluciones.
La plataforma ofrece servicios comunes a las aplicaciones de escritorio, permitiéndole al desarrollador enfocarse en la lógica específica de su aplicación. Entre las características de la plataforma están:

- Administración de las interfaces de usuario (ej. menús y barras de herramientas)
- Administración de las configuraciones del usuario
- Administración del almacenamiento (guardando y cargando cualquier tipo de dato)
- Administración de ventanas
- Framework basado en asistentes (diálogos paso a paso)

En este PFC, se ha empleado para desarrollar los clientes (de servicios Web y RMI) así como las primeras versiones de las diferentes clases Java que componen cada capa del sistema.

Web: http://netbeans.org/

Apache Ant

Es una herramienta usada para la realización de tareas mecánicas y repetitivas, normalmente durante la fase de compilación y construcción. Tiene la ventaja de no depender de las órdenes del *shell* de cada sistema operativo, sino que se basa en archivos de configuración XML y clases Java para la realización de las distintas tareas, siendo idónea como solución multi-plataforma.

En nuestro caso se usa para hacer más sencilla la compilación y ejecución del sistema.

Web: http://ant.apache.org/

ANEXO 2 | **Diagrama de esfuerzos**

En este anexo se muestran las distintas tareas realizadas durante el desarrollo del proyecto agrupadas en grupos estimándose el tiempo invertido en cada uno. Cada grupo se desglosa en subtareas para dar una mejor idea de las fases en que ha consistido el proyecto.

Grupos de Tareas

- Estudio previo: Engloba el trabajo de búsqueda de información previa, el estudio de las tecnologías a utilizar y desarrollo de aplicaciones sencillas y ejemplos para aprender el funcionamiento de las diferentes tecnologías.

- Diseño: Incluye la definición de la arquitectura del sistema, de las dos opciones diferentes de persistencia que se van a ofrecer y de las interacciones con datos y operaciones temporizadas.

- Implementación: Las tareas relacionadas con la propia implementación de la aplicación, es decir, el desarrollo de la propia aplicación.

- Testeo: Elaboración y ejecución de pruebas de las diferentes capas y funcionalidades del sistema, tanto para asegurar el buen funcionamiento como el rendimiento y solución de errores encontrados.

- Documentación: Elaboración de documentación, desde esta memoria y sus anexos hasta otros documentos empleados para mantener constancia de diferentes aspectos del sistema o control de versiones.

Tareas Realizadas

Estudio previo (230 h)

Estudio de apuntes y realización de las prácticas de la asignatura sobre servicios Web (Javier Fabra) del máster de la Universidad de Zaragoza, estudio del manual de Renew y pruebas con el programa, búsqueda de información sobre Apache Tomcat y Axis2.

Estudio y búsqueda de soluciones del problema de rendimiento al lanzar Apache Tomcat en una máquina virtual Java diferente.

Búsqueda de información sobre Hibernate, instalación y realización de tutoriales y test de correcto funcionamiento.

Estudio de prestaciones entre SQLite y MySQL, instalación y aprendizaje de ambas.

Instalación y configuración del entorno de trabajo, es decir, de todas las aplicaciones y software a utilizar y testeo del mismo.

Búsqueda y pruebas sobre características del lenguaje de programación Java empleados.

Diseño (161 h)

Diseño de la interface y de la arquitectura de la primera versión.

Diseño de la capa de persistencia de RLinda: cambios en la red, clases involucradas en la capa de persistencia, interacción con la base de datos, integración de Hibernate.

Diseño de la interface de la capa de temporización de RLinda: cambios en la red, clases involucradas en la capa de persistencia, formas de implementar el concepto de timeout, problemas de interacción con la capa de persistencia.

Diseño de la capa final de presentación, servicios Web finales y configuración de un solo módulo.

Diseño de clientes de prueba para todas las versiones.

Diseño del caso de uso

Implementación (232 h)

Desarrollo de la primera versión: RLinda como servicio Web SOAP/REST, nueva función de búsqueda y asignación, validador de tuplas y Apache Tomcat embebido.

Desarrollo de la versión persistente: empleando Hibernate para comunicarse con la base de datos.

Desarrollo de la capa de temporización.

Modificaciones en la capa de persistencia para interaccionar con la capa de temporización.

Desarrollo de la capa de presentación, servicios Web finales y clientes.

Desarrollo del caso de uso.

Testeo (165 h)

Pruebas de la primera versión: función de búsqueda y asignación, acceso REST y SOAP con el servidor Embebido.

Pruebas persistencia: coherencia base de datos-memoria.

Pruebas temporización: funcionamiento correcto de los timeouts de operaciones de lecturas y de tuplas temporizadas.

Pruebas de interacción de tuplas temporizadas con la persistencia.

Simulación en el clúster.

Documentación (120 h):

Documentación interna.

Elaboración de esta memoria y de sus anexos.

Gráfico de tarta

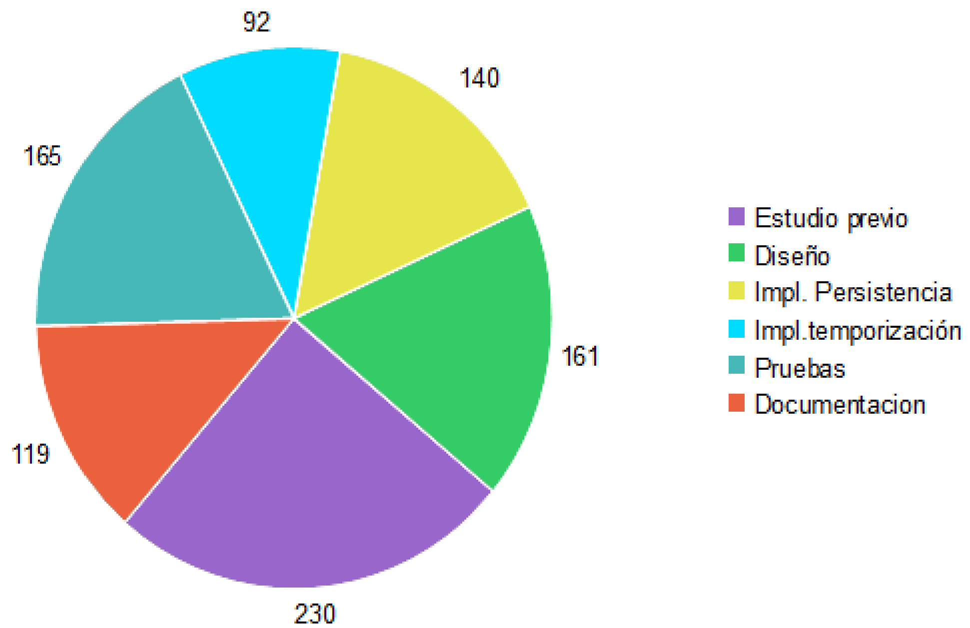


Figura 1 Distribución del tiempo empleado

Diagrama de GANTT

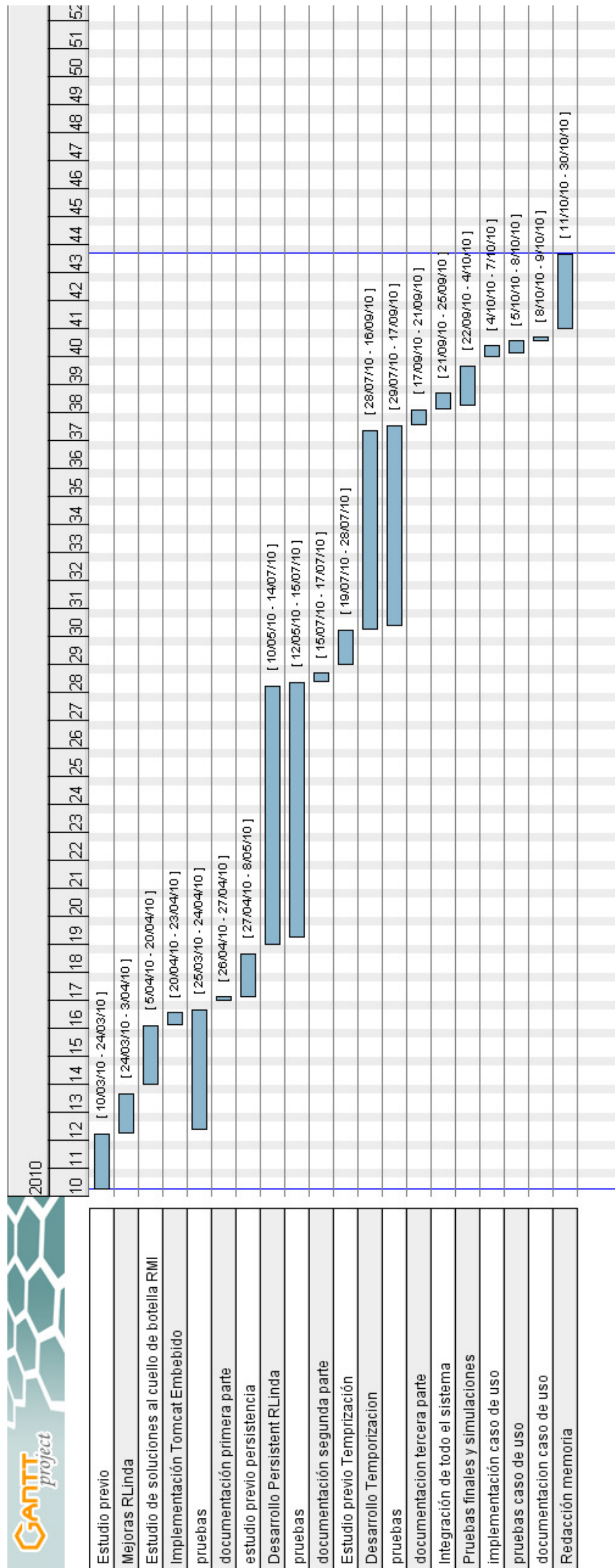


Figura 2 Diagrama de Gant

ANEXO 3 | **Manual de usuario de WS-PTRLinda**

Este Anexo describe el manual de usuario de WS-PTRLinda en el que se detalla la instalación de la aplicación, la ejecución y una descripción de los métodos ofrecidos por los servicios Web. También se muestra cómo modificar la herramienta para que emplee sólo una de las capas, la configuración de parámetros que afectan a la forma de funcionar de WS-PTRLinda, como crear un cliente sencillo con NetBeans, emplear algunas de las herramientas de WS-PTRLinda desde el lado del cliente y cómo funciona el mecanismo de las copias de seguridad.

1 Instalación

1.1 Pasos previos

Antes de instalar RLinda asegúrate de que tu ordenador tiene Java JDK versión 6 o superior instalado. Si tienes Java pero desconoces que versión, puedes averiguarlo fácilmente visitando este enlace:

http://www.java.com/es/download/installed.jsp

Si no tienes Java SDK instalado puedes bajarte la última versión desde el siguiente enlace:

http://www.oracle.com/technetwork/java/javase/downloads/index.html

La capa de persistencia requiere tener MySQL 5.1 instalado. Puedes descargarlo gratuitamente de la página Web oficial de MySQL. A partir de la versión 5.1 MySQL incorpora un instalador "wizard"que instala y configura el sistema gestor de bases de datos. Incluye una utilidad para crear una base de datos de forma sencilla. En todo caso puedes encontrar manuales de instalación y configuración en la propia Web de MySQL.

http://www.mysql.com/downloads/mysql/

1.2 El paquete de RLinda

WS-PTRLinda se distribuye en un archivo comprimido RAR que incluye, entre otras cosas, el código fuente. Antes de poder ejecutarlo libremente es necesario compilar este código, pero veamos antes lo que encontramos cuando descomprimimos el archivo:

WS-PTRLinda - que contiene el directorio con los fuentes de WS-PTRLinda y las librerías necesarias para su utilización.

Rlinda clients - Con los clientes de ejemplo tanto *Web Service* como RMI

Renew tools - con las herramientas a instalas para que funcione Renew, el programa sobre el que se construye el coordinador de WS-PTRLinda.

Ficheros de texto sobre cómo iniciar WS-PTRLinda y cómo configurar sus capas.

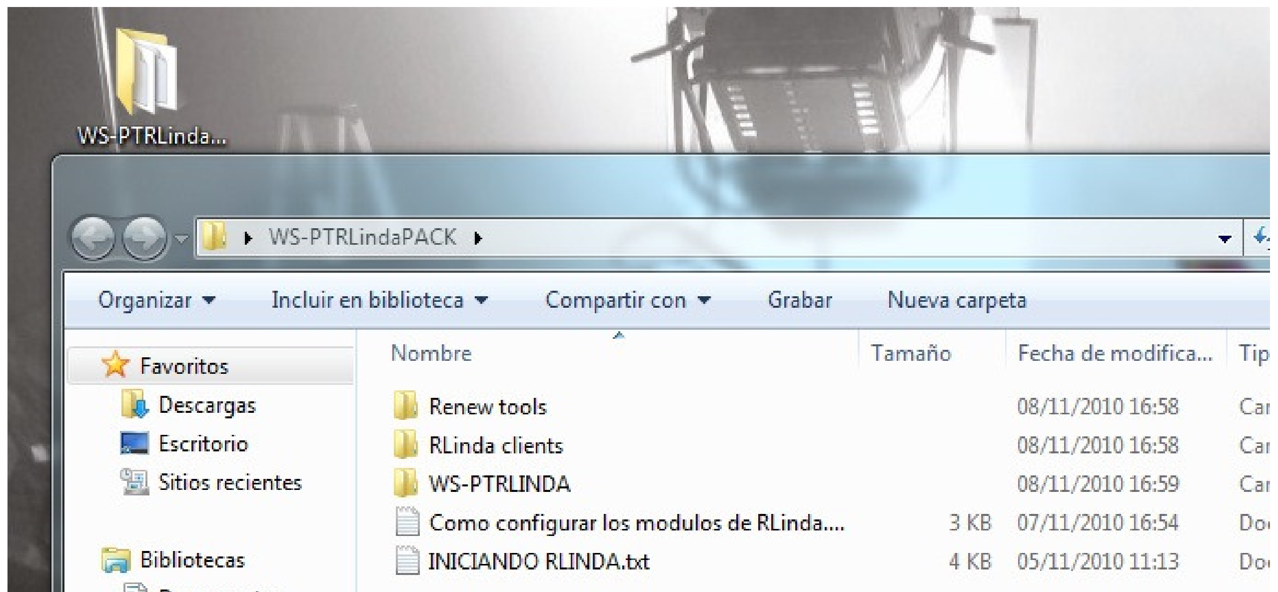


Figura 3

1.3 Compilando WS-PTRLINDA

A la hora de compilar RLinda es necesario añadir algunas herramientas y bibliotecas al sistema que son necesarias para su funcionamiento. Todas estas vienen incluidas en el directorio "Renew tools" del pack de distribución. Los pasos a seguir son los siguientes:

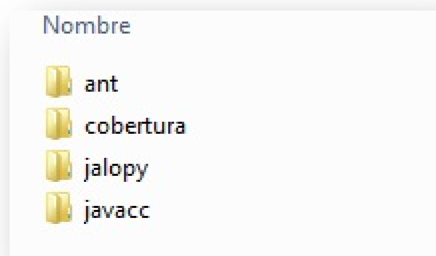


Figura 4

1.3.1 Copiar los directorios que se encuentran en Renew tools en el directorio C:/src del equipo. Las rutas de dichos directorios quedarán de la siguiente manera:

```
C:/src/cobertura/…
C:/src/ant/…
C:/src/jalopy/…
C:/src/javacc/…
```

1.3.2 Añadir las variables de entorno necesarias para Ant y Axis2:

```
$ANT_HOME="C\src\ant"
$PATH="C\src\ant"
$AXIS2_HOME="C:/src/WS-PTRLinda/AXIS2/axis2-1.5.1/lib"
```

1.3.3 El siguiente paso consiste en compilar la herramienta Renew, sobre la que se implementa WS-PTRLinda, que también se ofrece con sus códigos fuentes. Si has realizado los pasos anteriores correctamente solo deberes ejecutar la tarea de compilación, para ello abre el intérprete de comandos, sitúate en el directorio raíz de WS-PTRLinda y ejecuta :

```
ant compile
```

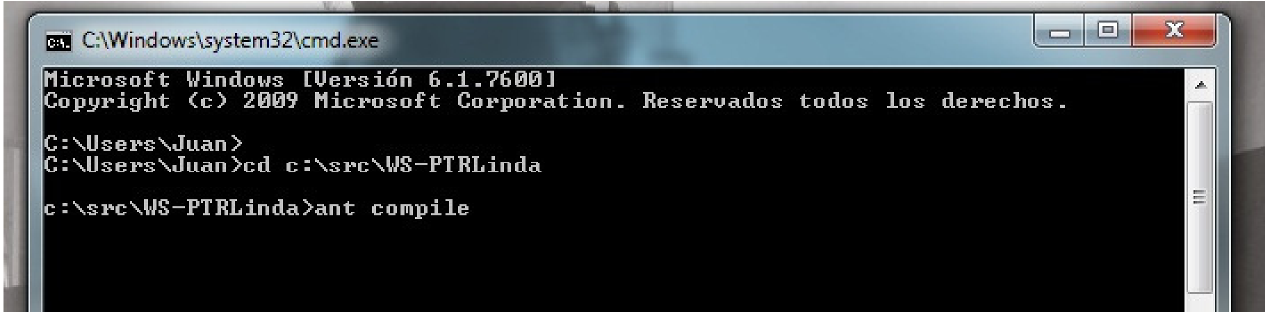


Figura 5

Aparecerá un mensaje de BUILD SUCCESSFUL indicando que la compilación se ha realizado correctamente.

1.3.4 Finalmente compilaremos la herramienta WS-PTRLinda ejecutando en el intérprete de comandos la tarea de compilación:

```
ant megacomp
```

Se mostrará de nuevo un mensaje de BUILD SUCCESSFUL indicando que la compilación se ha realizado correctamente.
¡WS-PTRLinda ya está instalado!

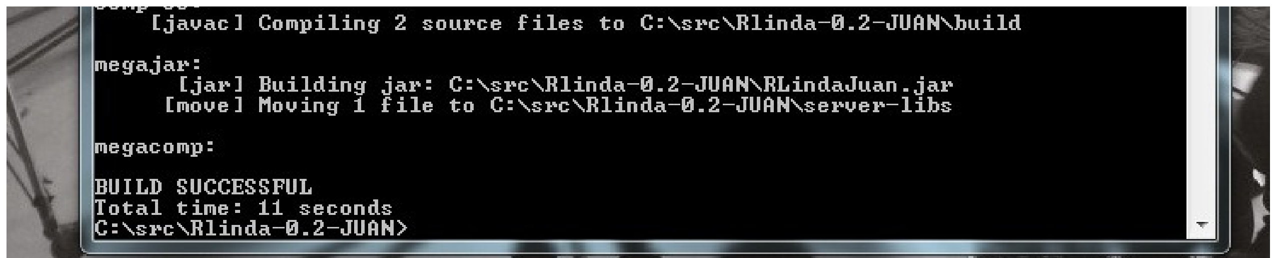


Figura 6

Existe la posibilidades de compilar y lanzar la ejecución automáticamente mediante una única tarea:

```
Ant doAll
```

2 Ejecución de WS-PTRLinda

La ejecución de WS-PTRLinda se realiza en dos partes. EN primer lugar es necesario ejecutar la herramienta Renew. Una vez abierta Renew muestra dos redes : el coordinador de WS-PTRLinda y el el lanzador.
El primero modela el funcionamiento del sistema y no debe de ser modificado. La segunda red se encarga de lanzar todo lo necesario para que WS-PTRLinda se y una vez está todo, inicia el propio WS-PTRLinda.

2.1 Iniciar Renew.

Para lanzar Renew simplemente abre el intérprete de comandos y sitúate en el directorio raíz de WS-PTRLinda. Una vez allí ejecuta la tarea “run-gui” que abre la herramienta usando la interfaz gráfica.

```
ant run-gui
```

Aparecerán las dos redes que forman RLinda.

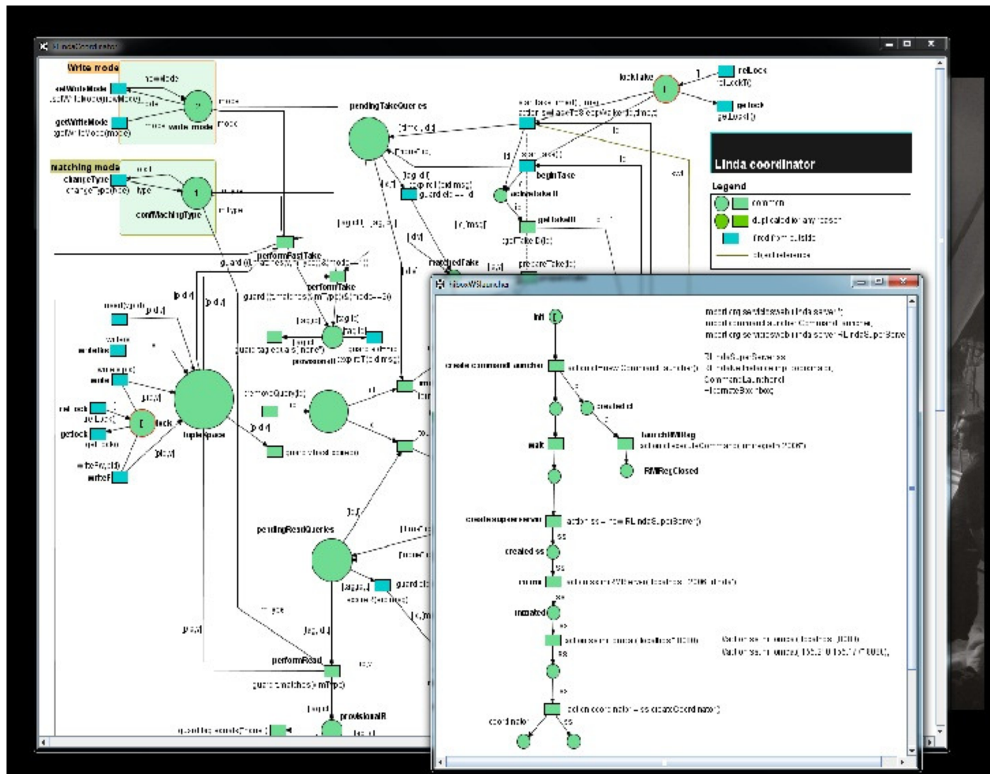


Figura 7

2.2 Configurar los parámetros de red

En la red lanzadora de RLinda se configuran los parámetros relativos a el acceso a la máquina desde el exterior, tanto mediante protocolo RMI como por servicios Web. Es necesario configurar el host y el puerto para el servidor RMI y para el contenedor de aplicaciones Apache Tomcat. Configura estos parámetros en las llamadas:

```
action ss.iniRMIServer(“localhost”,2006,”rlinda”)
action ss.iniTomcat(“localhost”,8080)
```

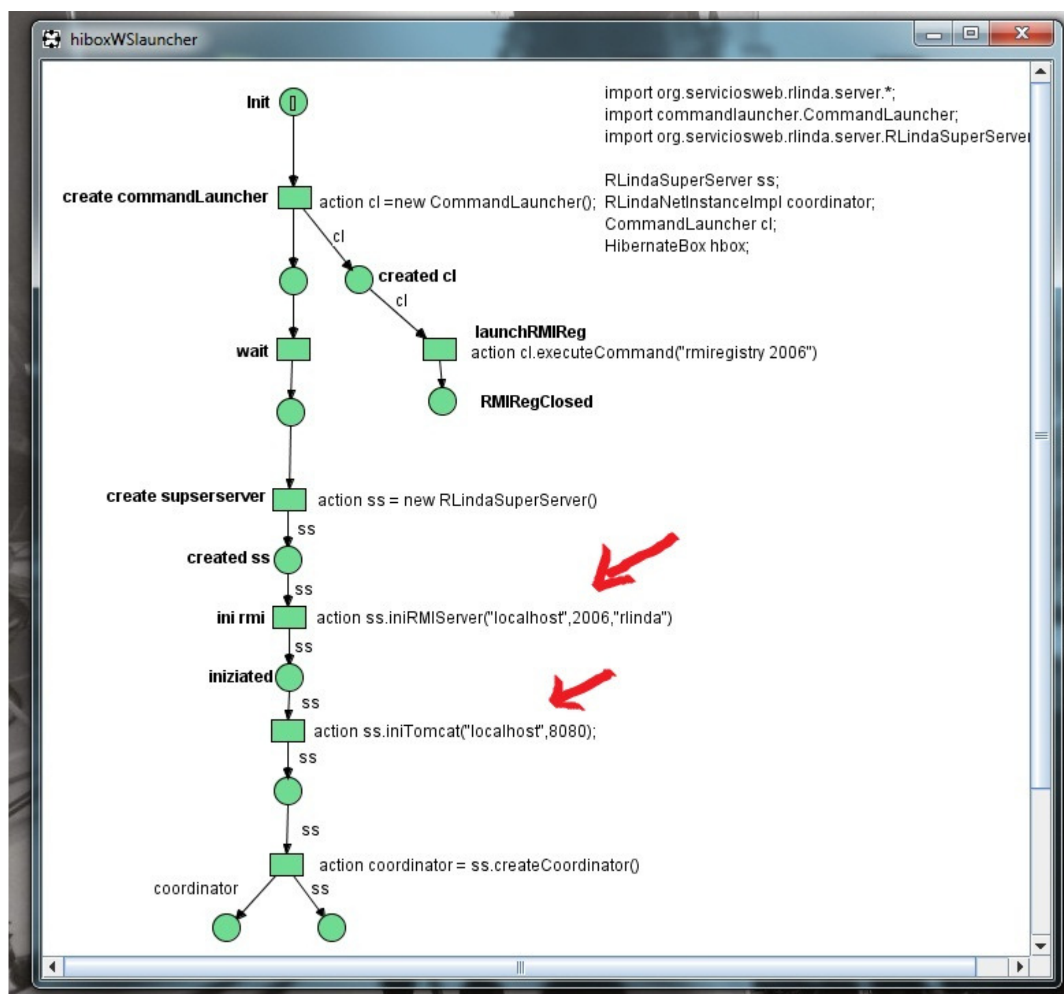


Figura 8

2.3 Configurar parámetros de Hibernate

Para que la capa de persistencia sepa donde encontrar la base de datos se deben configurar algunas propiedades del fichero de configuración de Hibernate. Este fichero puede encontrarse en la ruta:

```
WS-WTRLinda\src\org\servcicioswebb\rlinda\persistence\resources
\hibernate.cfg.xml
```

En dicho fichero modificar las siguientes propiedades con la información relativa a: la URL de la base de datos, nombre a usuario, contraseña, nombre de la base de datos:

```
<property name="connection.url">jdbc:mysql://localhost/RLindaDB</property>
<property name="connection.username">root</property>
<property name="connection.password">rlinda</property>
<property name="rlinda.dbname">RLindaDB</property>
```

Si no va a emplearse la capa de persistencia puede saltarse este paso.

2.4 Iniciando WS-PTRLinda al fin

Una vez llegados hasta aquí solo necesitas ubicarte en la red del lanzador de RLinda y pulsar “CTRL+R” o en el menú de Renew “Simulation/run simulation”.

Comprobarás que la red del lanzador se pone en color azul/morado (indica que está en ejecución) y que poco a poco la red va avanzando. Una vez que La red de petri marca los dos lugares inferiores con un 1, significa que WS-PTRLinda está listo para empezar a comunicar y coordinar clientes!

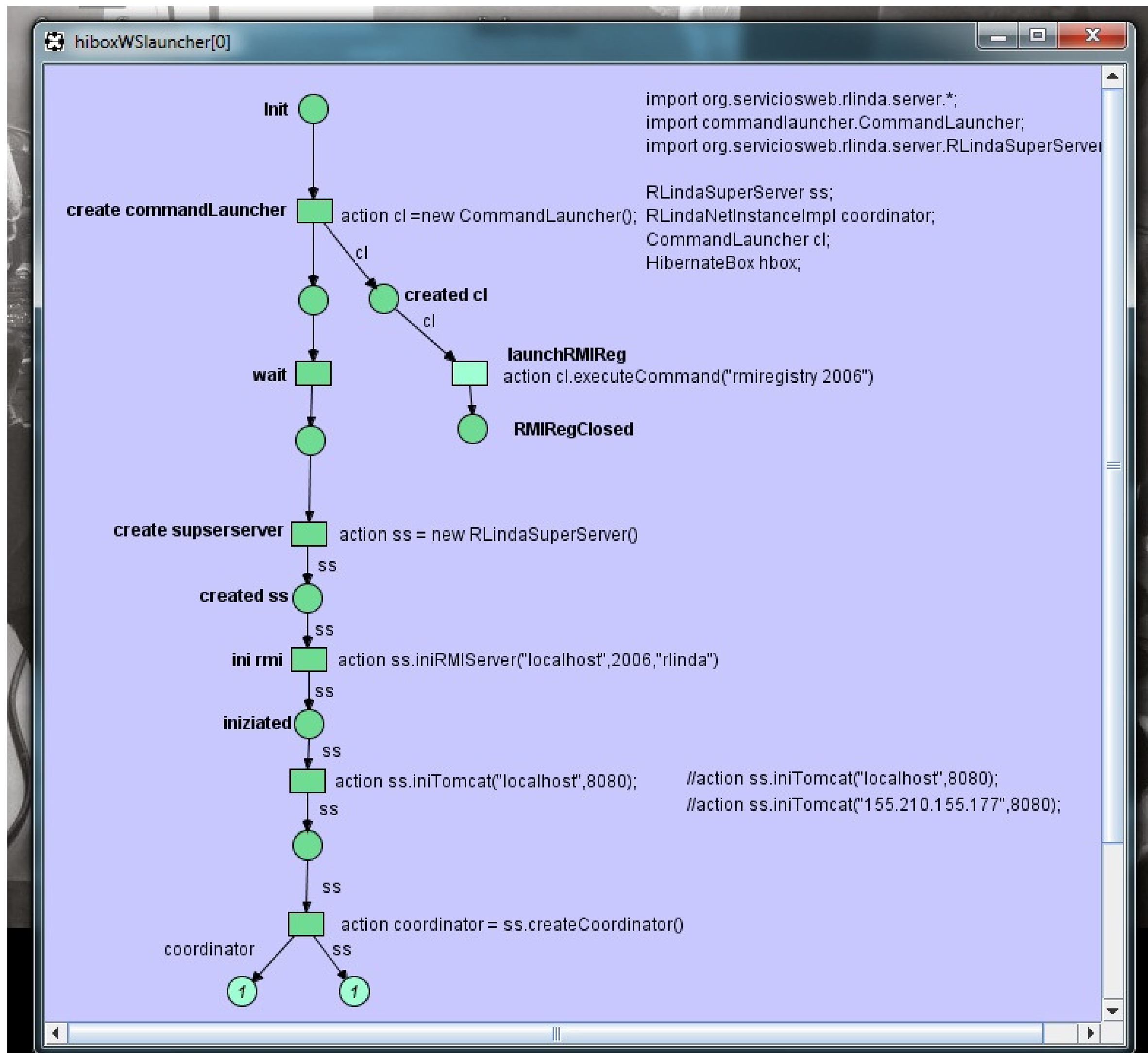


Figura 9

2.5 Comprobando que los servicios Web están accesible

Abre tu navegador e introduce la dirección que hayas configurado en la red lanzadora. Supongamos que lo has dejado tal y como estaba (localhost):

http://localhost:8080/axis2/services/listServices

Verás que ”rlinda” (por simplificar, no aparece como WS-PTRLinda) aparece en la lista de servicios desplegados, mostrando la dirección donde esta publicado , descripción y sus operaciones disponibles.

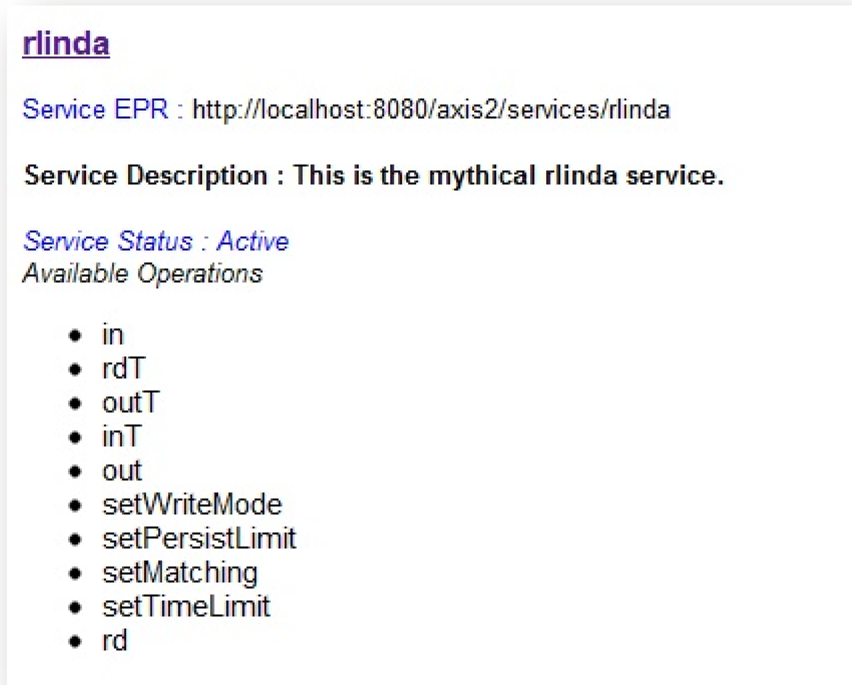


Figura 10

3 Crear un cliente sencillo con Netbeans

En la carpeta de distribución de RLinda se ofrece un cliente ejemplo para servicios Web así como un cliente ejemplo RMI. En este manual vamos a mostrar lo sencillo que es desarrollar un cliente desde el entorno gráfico de desarrollo NetBeans, totalmente gratuito.

Crear un proyecto nuevo en NetBeans : New Proyect/new Java Application

Seleccionar el nombre de nuestro proyecto: en nuestro caso elegiremos "WSClient".

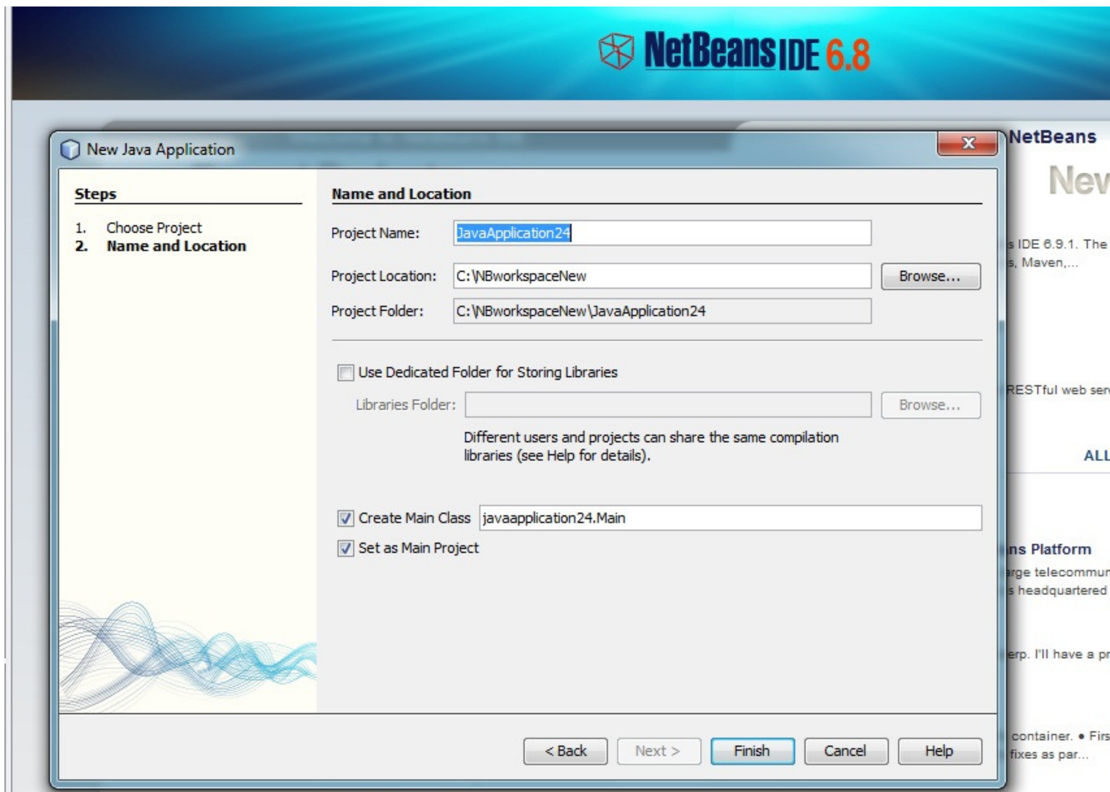


Figura 11

Ahora es necesario agregar la referencia al servicio Web:
botón derecho sobre el nombre del proyecto/new/Web Service Client.

Aparecerá la siguiente ventana dónde debemos introducir la dirección del servicio Web rlinda (WS-PTRLinda). Volviendo al punto 2.5 de comprobación de que los servicios se han desplegado correctamente, haciendo *click* en "rlinda" accedemos a su WSDL. La dirección de este WSDL es lo que nos interesa ingresar en el cuadro que sigue a continuación

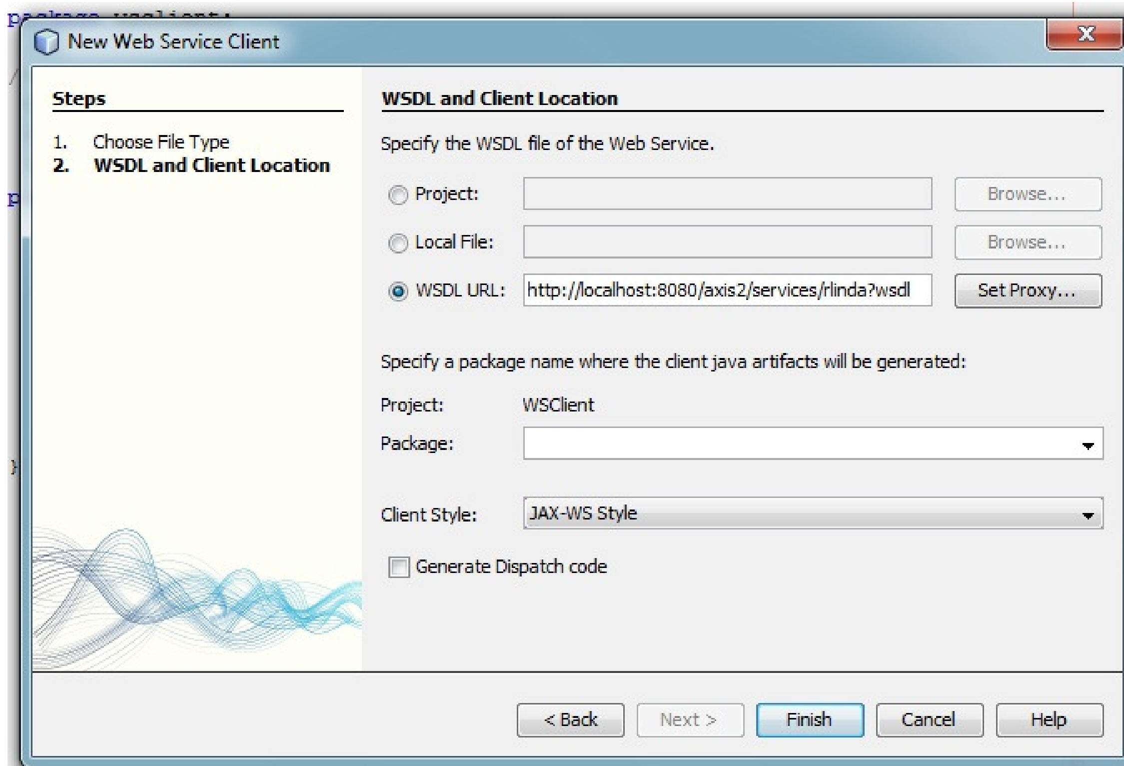


Figura 12

Pulsando en "finish" NetBeans realizará una serie de operaciones internas de forma que en nuestro árbol de directorios aparecerán nuevos elementos.

Navegando en ellos podemos llegar a las operaciones que nos ofrece WS-PTRLinda y basta con seleccionar una y arrastrarla para que NetBeans genera automáticamente el código necesario. Para su ejecución.

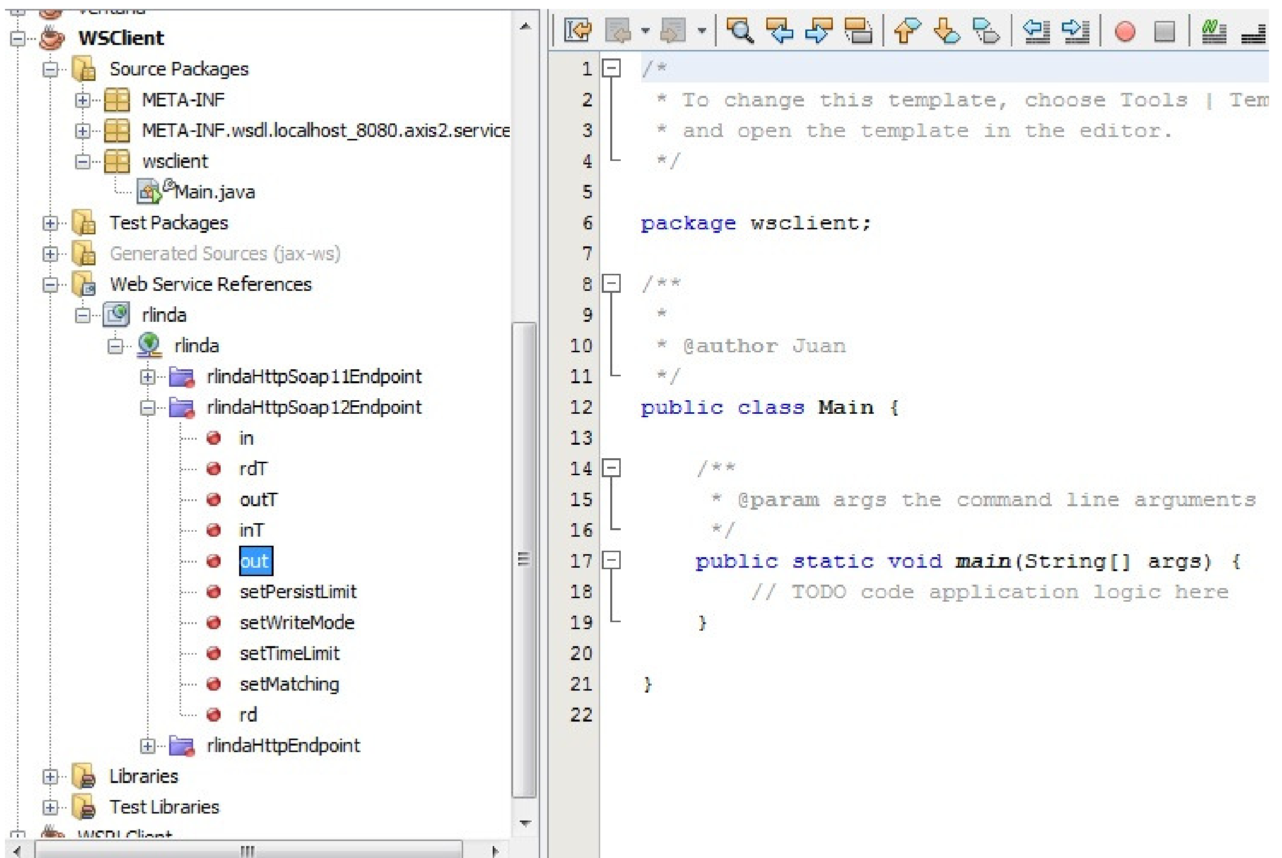


Figura 13

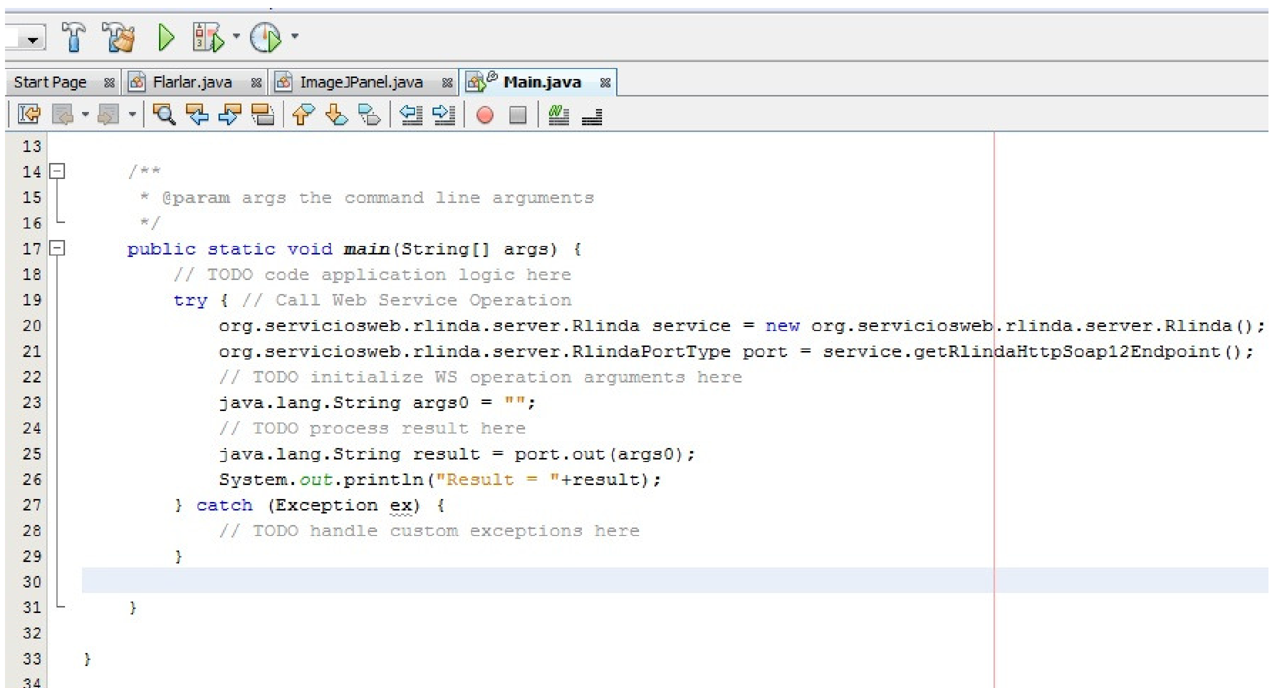


Figura 14

Introduce un valor para los parámetros y ejecuta la aplicación.

4 Métodos ofrecidos por los servicios Web

WS-PTRLinda se divide en dos servicios Web diferentes. Uno de ellos ofrece las operaciones básicas de RLinda , las versiones temporizadas y los métodos que sirven para configurar el sistema.
Un segundo servicio proporciona las operaciones de load y sabe para realizar y restaurar copias de seguridad de la base de datos.

4.1 Servicio Web principal: rlinda

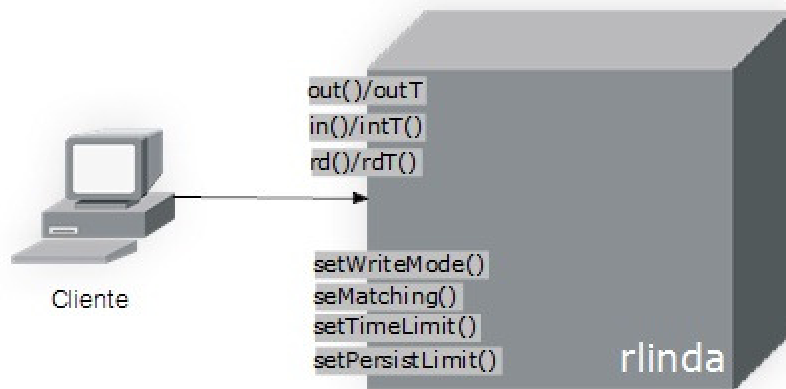


Figura 15

out(String tuple) - escribe una tupla en el espacio de tuplas.

outT(String tuple,long timeout) - escribe una tupla en el espacio de tuplas que será válida durante *timeout* segundos.

in(String pattern) - devuelve una tupla del espacio de tuplas que concuerde con el patrón *pattern* dado. La tupla se borra del espacio de tuplas. Si no hay ninguna se bloquea hasta que aparezca una.

inT(String pattern, long timeout) - devuelve una tupla del espacio de tuplas que concuerde con el patrón *pattern* dado. La tupla se borra del espacio de tuplas. Si no hay ninguna se bloquea hasta que se escriba una o pasen *timeout* segundos. En el caso segundo se devuelve una tupla indicativa *["operation timeout"]*.

rd(String pattern) - devuelve una tupla del espacio de tuplas que concuerde con el patrón *pattern* dado. La tupla se mantiene del espacio de tuplas. Si no hay ninguna se bloquea hasta que se escriba una.

rdT(String pattern, long timeout) - devuelve una tupla del espacio de tuplas que concuerde con el patrón *pattern* dado. La tupla se borra del espacio de tuplas. Si no hay ninguna se bloquea hasta que se escriba una o pasen *timeout* segundos. En el caso segundo se devuelve una tupla indicativa *["operation timeout"]*.

setMathing(int type) - cambia el *tipo de matching* a emplear. (1-weak_matching,2-strong_matching,3-attributte_matching).

setWriteMode(int mode) - cambia el modo de persistencia (modo de escritura). El modo de persistencia solo debe ser cambiado al inicio de la ejecución sin que ose haya comenzado a escribir o borrar tuplas. (1-persistencia débil,2-persistencia fuerte).

setPersistLimit(int limit) - Configura el valor frontera de tuplas en la persistencia débil. Este valor indica el número máximo de tuplas que pueden permanecer sin ser volcadas a la base de datos.

setTimeLimit(long limit) - Configura el valor frontera de tiempo en la persistencia débil. Este valor indica que cada *limit* milisegundos las tuplas que no han sido persistidas se vuelquen en la base de datos.

4.2 Servicio Web de creación y restauración de copias de seguridad: rlhibox

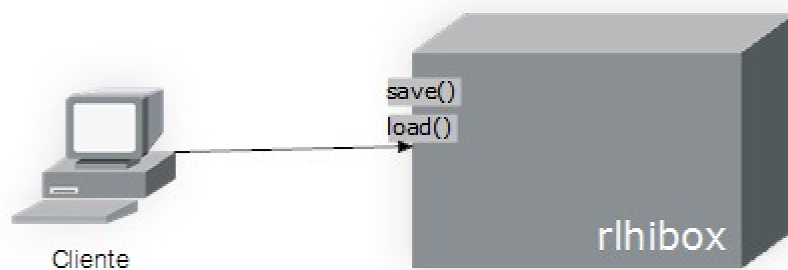


Figura 16

Estos métodos emplean *SOAP with Attachments* por lo que el cliente debe de estar correctamente configurado para poder recibir correctamente las copias de seguridad o enviarlas. Los códigos de el cliente receptor y el emisor se adjuntan a continuación.

load(String filename) - Se crea una copia de la base de datos de WS-PTRLinda y se envía al cliente. En el directorio raíz del cliente aparecer un archivo de nombre *filename.rar*.

save(String filename) - Se recibe el archivo de copia de seguridad de la base de datos y se restaura el estado de WS-PTRLinda. El fichero ha de llamarse *filename.rar*.

4.3 Ejemplo de cliente LOAD

RLindaSOAPClient.java

```
public class RLindaSOAPClient{
private static EndpointReference targetEPR;
      private static String axis2fileSystem;
  public RLindaSOAPClient(String endpoint, String axis2repository) {
      targetEPR = new EndpointReference(endpoint);
      axis2fileSystem = axis2repository;
  }
public static void load(String nombreBackup) throws Exception {
         File file = new File(nombreBackup);
         if (file.exists()){
                  System.out.println("archivo encontrado");
                  transferFile(file, nombreBackup);
```

```
            }else
                throw new FileNotFoundException();
}
public static void transferFile(File file, String destinationFile)
                throws Exception {

        Options options = new Options();
        options.setTo(targetEPR);
        options.setProperty(Constants.Configuration.ENABLE_SWA,
                        Constants.VALUE_TRUE);
        options.setSoapVersionURI(SOAP11Constants.SOAP_ENVELOPE_NAMESPACE_URI);

        // Increase the time out when sending large attachments
        options.setTimeOutInMilliSeconds(10000);
        options.setTo(targetEPR);
        options.setAction("urn:restoreBackup");

        // assume the use runs this sample at
        // <axis2home>/samples/soapwithattachments/ dir
        ConfigurationContext configContext = ConfigurationContextFactory
                        .createConfigurationContextFromFileSystem(axis2fileSystem,
                                        null);
        ServiceClient sender = new ServiceClient(configContext, null);
        sender.setOptions(options);
        OperationClient mepClient = sender
                        .createClient(ServiceClient.ANON_OUT_IN_OP);
        MessageContext mc = new MessageContext();
        FileDataSource fileDataSource = new FileDataSource(file);
        // Create a dataHandler using the fileDataSource. Any implementation of
        // javax.activation.DataSource interface can fit here.
        DataHandler dataHandler = new DataHandler(fileDataSource);
        String attachmentID = mc.addAttachment(dataHandler);
        SOAPFactory fac = OMAbstractFactory.getSOAP11Factory();
        SOAPEnvelope env = fac.getDefaultEnvelope();
        OMNamespace omNs = fac.createOMNamespace(
                        "http://server.rlinda.serviciosweb.org", "swa");

        OMElement uploadFile = fac.createOMElement("uploadFile", omNs);
        OMElement nameEle = fac.createOMElement("name", omNs);
        nameEle.setText(destinationFile);
        OMElement idEle = fac.createOMElement("attchmentID", omNs);
        idEle.setText(attachmentID);
        uploadFile.addChild(nameEle);
        uploadFile.addChild(idEle);
        env.getBody().addChild(uploadFile);
        mc.setEnvelope(env);
        mepClient.addMessageContext(mc);
        mepClient.execute(true);
        MessageContext response = mepClient
                        .getMessageContext(WSDLConstants.MESSAGE_LABEL_IN_VALUE);
        SOAPBody body = response.getEnvelope().getBody();
        OMElement element = body.getFirstElement().getFirstChildWithName(
        new QName("http://server.rlinda.serviciosweb.org","return"));
        System.out.println(element.getText());
    }
}
```

4.4 Ejemplo de cliente SAVE

RLindaSAVEClient.java

```
public class StatisticsServiceClient{
private  EndpointReference targetEPR;
  public StatisticsServiceClient(String endpoint) {
      targetEPR = new EndpointReference(endpoint);
  }
public  void callLoad(String name) throws Exception {
         Options options = new Options();
         options.setTo(targetEPR);
         options.setAction("urn:getBackup");
         options.setSoapVersionURI(SOAP11Constants.SOAP_ENVELOPE_NAMESPACE_URI);

         // Increase the time out to receive large attachments
         options.setTimeOutInMilliSeconds(10000);

         ServiceClient sender = new ServiceClient();
         sender.setOptions(options);
         OperationClient mepClient = sender.createClient(ServiceClient.ANON_OUT_IN_OP);

      MessageContext mc = new MessageContext();
      SOAPEnvelope env = createEnvelope(name);  //nombre demandado por el usuario
      mc.setEnvelope(env);

         mepClient.addMessageContext(mc);
         mepClient.execute(true);

         // Let's get the message context for the response
         MessageContext response =
mepClient.getMessageContext(WSDLConstants.MESSAGE_LABEL_IN_VALUE);
         SOAPBody body = response.getEnvelope().getBody();
         OMElement element = body.getFirstChildWithName(new
QName("http://server.rlinda.serviciosweb.org","getBackupResponse"));
      if (element!=null) {
                  processResponse(response, element);
      }else{
          throw new Exception("Malformed response.");
      }
}
private static void processResponse(MessageContext response, OMElement element) throws
Exception {

         String fileName = element.getFirstChildWithName(new
QName("http://server.rlinda.serviciosweb.org","fileName")).getText();
         System.out.println("File Name : " + fileName);
         OMElement graphElement = element.getFirstChildWithName(new
QName("http://server.rlinda.serviciosweb.org","file"));
      //retrieving the ID of the attachment
         String backupFileID = graphElement.getAttributeValue(new QName("href"));

      //remove the "cid:" prefix
      backupFileID = backupFileID.substring(4);

         //Accesing the attachment from the response message context using the ID
      System.out.println(backupFileID);
         DataHandler dataHandler = response.getAttachment(backupFileID);
      if (dataHandler!=null){
                  // Writing the attachment data (graph image) to a file
                  File file = new File(fileName);
                  FileOutputStream outputStream = new FileOutputStream(file);
                  dataHandler.writeTo(outputStream);
                  outputStream.flush();
                  System.out.println("Backup file downloaded to :" + file.getAbsolutePath());
      }else {
          throw new Exception("Cannot find the data handler.");
      }
}
```

```
private static SOAPEnvelope createEnvelope(String destinationFile) {
         SOAPFactory fac = OMAbstractFactory.getSOAP11Factory();
         SOAPEnvelope env = fac.getDefaultEnvelope();
         OMNamespace omNs = fac.createOMNamespace("http://server.rlinda.serviciosweb.org",
                           "swa");
         OMElement statsElement = fac.createOMElement("getStats", omNs);
         OMElement nameEle = fac.createOMElement("fileName", omNs);
         nameEle.setText(destinationFile);
         statsElement.addChild(nameEle);
         env.getBody().addChild(statsElement);
         return env;
      }
}
```

5. Usando solo la capa de Persistencia o la de Temporización

Las capas de WS-PTRLinda son independientes y pueden usarse por separado. Esta configuración no es automática y resulta necesario realizar algunos pequeños cambios en el código de la aplicación, en lo referente a la declaración de variables y su inicialización al comienzo de las clases que representan las capas.

El siguiente texto se puede encontrar en un archivo *.txt* incluido en el directorio de WS-PTRLinda.

5.1 Uso normal: Temporized Persistent RLinda

Modificar en PTRLinda.java :

1 - declaración de variables de la clase:

TemporizationBox tebox;
HibernateBox hibox;

2 - constructor de PTRLinda().

tebox = new TemporizationBox();
hibox = new HibernateBox;

Modificar en TemporizedBox.java:

1 - declaración de variables de la clase:

RLindaCoordinator rlinda; - HibernateBox rlinda; 2 - constructor - rlinda = new RLindaCoordinatorImpl();
rlinda = new HibernateBox();

Red RLindaCoordinator.rnv:

Añadir ”[]” para que se cree la referencia a hibox (en la parte de declaraciones).
OPCIONAL: dar valor a WriteMode = 1.

5.2 Solo Temporización: Temporized RLinda

Modificar en PTRLinda.java :

1 - declaración de variables de la clase:
TemporizationBox tebox;
Comentar: HibernateBox hibox; 2 - constructor de PTRLinda(). - tebox = TemporizationBox(); - comentar: hibox = HibernateBox;

Modificar en TemporizedBox.java:
1 - declaración de variables mde la clase:
RLindaCoordinator rlinda;
HibernateBox rlinda; 2 - constructor - rlinda = new RLindaCoordinatorImpl(); - rlinda = HibernateBox();

Modificar red RLindaCoordinator.rnv:
- Quitar la referencia a hibox (declaraciones derecha)
- Cambiar el modo de persistencia (writeMode) a 2.

5.3 Solo persistencia: Persistent RLinda

Modificar en PTRLinda.java :
1 - declaración de variables de la clase:
TemporizationBox tebox; - HibernateBox hibox; - HibernateBox tebox; 2 - constructor de PTRLinda(). - tebox = TemporizationBox();
hibox = HibernateBox;
tebox = hibox;

Modificar en TemporizedBox.java:
NADA

Modificar red RLindaCoordinator.rnv:
Quitar la referencia a hibox (declaraciones derecha).

ANEXO 4 | **Modelo de desarrollo de software**

Para la realización de este PFC se ha seguido una metodología iterativa e incremental. La razón es simple, WS-PTRLinda está compuesto por varias capas de manera que cada nueva se sitúa sobre la anterior. En esta metodología, partiendo de de una base sencilla se van añadiendo nuevas funcionalidades y probando que funcionan correctamente en cada etapa. El desarrollador saca ventaja de lo aprendido a lo largo de la fase anterior incrementando versiones entregables del sistema. El aprendizaje viene de dos fuentes: el desarrollo del sistema, y su uso.

El punto de partida es la versión original de RLinda con acceso mediante protocolos SOAP y RMI [3]. Existe una fase de formación seguida de tres claras fases de desarrollo que proporcionan diferentes versiones, la primera es una simple versión con varios añadidos mientras que las dos siguientes ya corresponden a las capas de persistencia y temporización. En la primera fase de desarrollo las tareas a realizar con aisladas entre sí. En cambio las fases segunda y tercera se componen de una primera parte de estudio previo para familiarizarse con el entorno, una fase de diseño, una fase de implementación y una fase de pruebas. De aquí adelante se comentan aspectos del desarrollo que no han sido mencionados antes en la memoria por falta de espacio o simplemente porque no cuadraban en ninguna sección.

Fase 0: formación

Como preparación previa se realizaron las prácticas sobre servicios Web del máster que se imparte en la universidad de Zaragoza con el fin de adquirir conocimientos sobre servicios Web, diferencias entre los protocolos REST y SOAP y los elementos clave de cada uno de ellos. Se estudió el manual de usuario de la herramienta Renew, recordaron conceptos sobre redes de Petri y realizaron algunos ejercicios básicos sobre integración de código Java con la implementación de redes de Petri de Renew y acceso desde el exterior a las mismas (RMI). Adicionalmente se desplegó un servidor Apache Tomcat y se desplegó un servicio Web con acceso a una red de Renew.

Fase 1: mejoras en RLinda

En esta primera fase se añaden diferentes mejoras a la herramienta aprovechando así a coger experiencia con las tecnologías que se usarán a lo largo del resto del proyecto. Dichas mejoras no guardan una estrecha relación entre ellas de manera que para cada una hay que realizar una pequeña búsqueda de información para luego pasar a su diseño e implementación.
Las mejoras añadidas son:

- Acceso por protocolo REST.
- Validador de tuplas en formato XML basado en esquemas XMLSchema.
- Conversor de tuplas en formato String a formato XML.
- Función de matching con varios criterios.

Cada una de estas mejoras obliga a enfrentarse a una parte distinta de la herramienta, de manera que al realizarlas se adquiere destreza y experiencia en la modificación y el uso del sistema.

Una vez llegado a este punto se había que enfrentarse a la solución de la bajada de rendimiento de RLinda con mucha concurrencia de usuarios. Hubo que estudiar varias soluciones y finalmente decantarse por una e implementarla. Se ha hecho uso de una distribución embebida del servidor de aplicaciones Apache Tomcat que se integra en el propio código de la aplicación. El diseño y la

implementación de este servidor embebido copan la mayor parte de los recursos en esta primera fase. Una vez finalizada, RLinda está listo para soportar las capas siguientes.

Fase 2: capa de persistencia

La fase de estudio previo ha consistido en el estudio de manuales y realización de tutoriales de la herramienta *Hibernate* pues la curva de aprendizaje de Hibernate requiere cierto esfuerzo en los comienzos. Se valoraron las prestaciones de dos bases de datos, MySQL y SQLite. Finalmente se optó por la segunda aunque las pruebas finales de la fase de persistencia mostraron resultados inesperados y hubo que cambiar a MySQL. Puede encontrarse información sobre este problema en el anexo destinado a limitaciones y problemas encontrados.

Al emplear Hibernate para comunicarse con la base de datos el cambio de SQLite a MySQL resultaba sencillo y solo obligó a cambiar los métodos que realizan la copia de seguridad y la restauran, pues la forma de hacerlo depende de cada base de datos.

Fase 3: Capa de temporización

En la capa de temporización no interviene ninguna nueva tecnología de manera que estudio previo queda reducido a las diferentes formas de representar el formato de tiempo del lenguaje de programación Java y como representarlo en WS-PTRLinda

Las fase diseño adquiere especial importancia, fue necesario dejar muy claro cómo reaccionaría el sistema al aplicar el concepto de tiempo y ver cómo desarrollar un mecanismo que desbloquee los procesos bloqueados al realizar operaciones de lectura que no encuentran ninguna tupla que se corresponda con el patrón que proporcionan como parámetro. Al trabajar con el concepto de tiempo y *threads* hay que prever situaciones de bloqueo y tratar de que el sistema mida el tiempo de una forma real. Para ello se dibujaron trazas de ejecución de los casos más problemáticos para ver cómo tratarlos.

Una vez previsto todo se paso a la fase de implementación y más tarde a la de pruebas.
Una vez acaba da la parte de temporización hubo que realizar algunos cambios en la capa de persistencia para que soportara la capa de temporización. Una vez realizados se preparó una batería de pruebas sencilla que verificara que todo funciona correctamente.

Simulación final

Se ejecutaron pruebas en el clúster de la Universidad de Zaragoza para obtener resultado sobre el rendimiento WS-PTRLinda y poder compararlo con la versión anterior. Los resultados de dichas pruebas se comentan en el anexo correspondiente.

Diagrama del modelo de desarrollo

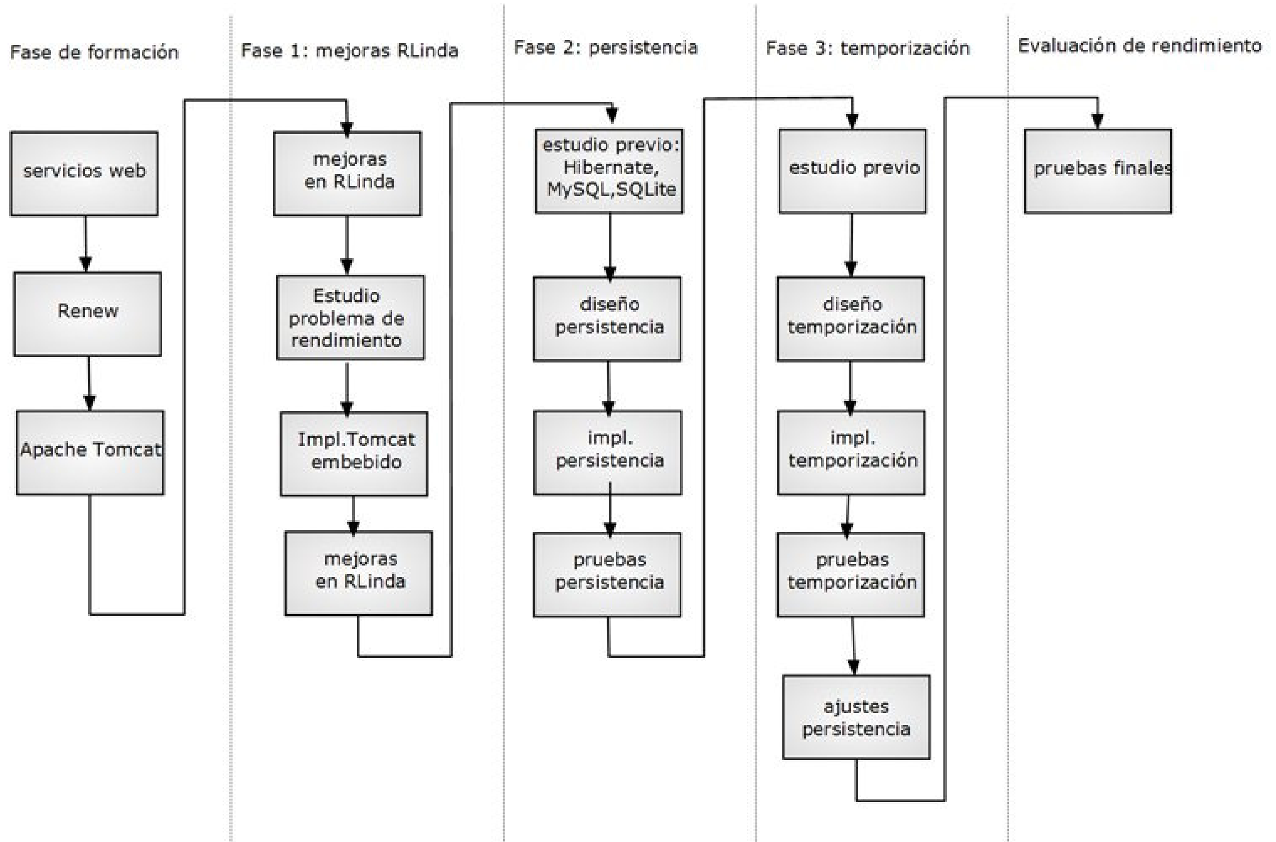


Figura 17

ANEXO 5 | **Limitaciones y problemas encontrados**

En este anexo se mencionan y comentan las limitaciones y problemas encontrados a lo largo del desarrollo del PFC, tanto a nivel conceptual como tecnológico.

SQLite

A la hora de aplicar persistencia se valoraron dos opciones diferentes para la base de datos: MySQL y SQLite [REFERENCIAS A CADA UNO]. Tras un pequeño estudio previo se decidió por emplear la segunda. Sus bazas eran claras, SQLite proporciona una base de datos basada en un fichero (frente a la necesidad de instalación e infraestructura de MySQL) y las bibliotecas para su uso se incluyen con la aplicación WS-PTRLinda de manera que no es necesaria una instalación previa por parte del usuario. Al almacenar todos los datos en un fichero, realizar una copia de seguridad es tan sencillo como copiar este fichero con otro nombre. Restaurar una base de datos simplemente requiere reiniciar SQLite indicándole que el fichero de la base de datos es otro (la copia de seguridad).

La capa de persistencia se implementó utilizando como base de datos SQLite y funcionaba perfectamente. Fue cuando se realizaron unas pruebas más severas en concurrencia de accesos cuando comenzó a aparecer una excepción procedente de SQLite que bloqueaba la base de datos. En internet encontré información sobre dicho problema además de la noticia de que no hay forma de solucionarlo pues reside en la propia base de SQLite.

SQLite posee un sistema de *locks* para garantizar el acceso en exclusión mutua a la base de datos pero como esta no deja de ser un fichero normal y corriente ubicado en el sistema operativo, está sujeto a todas las restricciones de acceso que este imponga. WS-PTRLinda es un sistema con usuarios concurrentes de manera que una mínima carga del sistema provocaba que varios clientes quisiesen escribir y se lanzase dicha excepción.

Este problema supuso muchas horas y quebraderos de cabeza hasta que finalmente se decidió cambiar a MySQL. Afortunadamente el cambio de base de datos es muy sencillo al emplear *Hibernate* y sólo requiere rediseñar las funciones de creación y restauración de copias de seguridad.

Pérdida de datos de tuplas en Renew

Tanto en la capa de persistencia como en la capa de temporización se sobrecarga el tipo de datos *Tuple* de Renew. El objetivo es guardar en su interior cierta información, en caso de persistencia el identificador de persistencia o *pid*. El identificador de persistencia funciona como clave primaria de la tupla en la base de datos. Mientras se probaban las primeras aproximaciones de la persistencia *Hibernate* producía de vez en cuando una Excepción explicando que está intentado acceder a una clave primaria que no existe.

Esto sucede porque en Renew cuando en un mismo lugar coinciden dos tuplas iguales, Renew las unifica como una única tupla y anota delante que hay dos instancias. A nivel de objeto Java, esto significa que donde había dos objetos cada uno con un *pid* diferente, ahora pasaba a haber solo uno (con valor de *pid* uno de los dos).
Varios problemas se derivan de esta característica de diseño de Renew, una de ellas la siguiente: en las operaciones de persistencia, cuando la primera copia de la tupla es extraída se accede a la base de

datos para borrar la tupla con *pid* cinco (por ejemplo). Cuando tarde o temprano se extraiga la otra copia de la tupla, se accederá a la base de datos a esa misma posición para intentar borrarla y saltará la excepción.

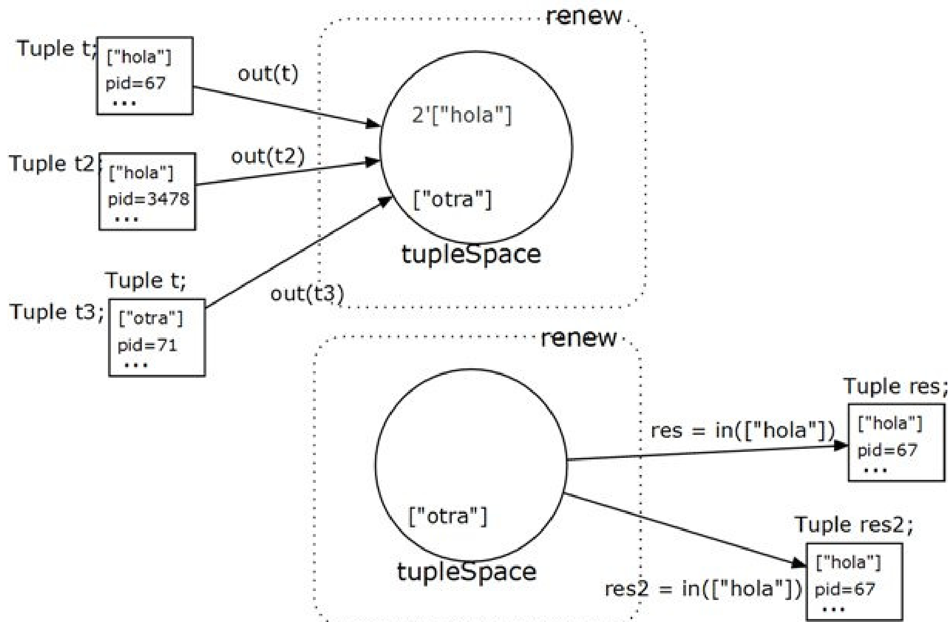


Figura 18

Se solucionó incorporando dentro de la red a cada tupla una cabecera con su *pid*. Como el *pid* es único por definición nunca existirán dos tuplas iguales y los campos internos del objeto *Tuple* no se sobrescribirán.

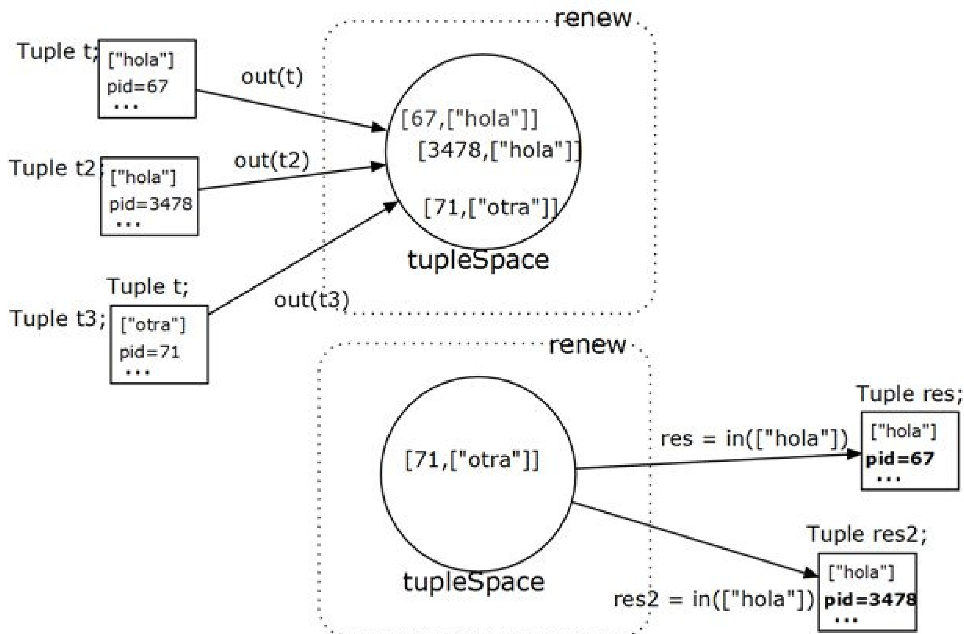


Figura 19

Almacenamiento en la base de datos de las tuplas expiradas

Las tuplas temporizadas guardan su valor de *timeout* en un campo interno dentro del propio tipo *Tuple* además de estar en forma de cabecera *["TIMEOUT",valor]*. La base de datos se sitúa en la capa de persistencia así que no guardaba ningún tipo de información del tiempo de expiración restante de las tuplas a excepción de la tupla en sí en formato String (que contendrá su cabecera). Esto supone que al realizar una copia de seguridad y restaurarla más adelante la única información temporal de la tupla está en la cabecera y resulta obsoleta (sería necesario saber cuando entró esa tupla en el sistema para ver si ha caducado o no).

Esto motivó varios cambios:

El primero de ellos, guardar la fecha de expiración *expirationTime* que indica el tiempo de sistema en milisegundos a partir del cual la tupla ya no es válida.

La base de datos almacena este *expirationTime* en la base de datos (0 si no está temporizada).

Al realiza la copia de seguridad se emplean llamadas SQL que vuelcan las tuplas en bloque a un fichero de manera que no se pueden controlar las tuplas temporizadas. En cambio al restaurar la base de datos si que es necesario ir introduciendo una a una las tuplas en el espacio de tuplas de manera que es momento ideal de comprobar si han expirado mirando el tiempo actual de sistema y el *expirationTime*.

El método toString() de Renew vs MySQL

Realizando las pruebas de restauración de la base de datos con tuplas temporizadas, se encontraron resultados extraños cuando se trataba de leer alguna de las tuplas restauradas. Al ejecutar operaciones de lectura con patrones que normalmente debían devolver como resultado una tupla de espacio de tuplas, permanecían bloqueados como si la tupla no existiese. Esto solo sucedía con tuplas restauradas con campos de tipo cadena (*String*).

En Renew las cadenas se muestran con los caracteres de las comillas al principio y al final. Al guardar una tupla en la base de datos se emplea el método *toString()* que devuelve la tupla en formato texto pero suprime las comillas. Al restaurar la base de datos, las tuplas son devueltas sin las comillas y Renew no da signos de error. El problema es que los patrones en las operaciones de lectura nunca pueden concordar con estas tuplas.

La mayor parte del tiempo se empleó en acotar el problema y localizarlo, la solución fue sencilla y consistió en modificar el método *toString()* para que incluya también las comillas. Se realizaron pruebas para ver que la solución no dañaba otras partes del código.

Envío de copias de seguridad mediante REST

A la hora de enviar/recibir los archivos de copia de seguridad RMI no supone ningún problema y SOAP permite adjuntarlos en el propio mensaje. En cambio REST utiliza *HTTP*, que no posee un sistema de incluir archivos en el propio *HTTP* directamente.

Por este motivo, únicamente pueden accederse a las operaciones de creación y restauración de la base de datos mediante protocolo SOAP o RMI.

Tomcat Embebido

El desarrollo de los servicios Web de persistencia ha venido acompañado de un error por parte de Axis2. Esta excepción era muy genérica y no tenía sentido, de manera que era realmente difícil de localizar. El problema se producía con las operaciones que hacen uso de *SOAP with attachments*, es decir, el envío y la recepción de los archivos de copia de seguridad.

De forma experimental se comprobó que separando estas operaciones en un servicio aparte se reducía bastante este problema aunque seguía apareciendo sin atender aparentemente a ningún criterio.

La versión de Apache Tomcat utilizada es una versión embebida del mismo. Se ofrece como alternativa para cubrir ciertas necesidad pero apenas incluye soporte y no se garantiza que funcione igual que la versión oficial. Su uso en la red es escaso de manera que las referencias y la posibilidad de conseguir ayuda es muy limitada. Si le añadimos que la Excepción devuelta corresponde a un error típico resultaba imposible encontrar una sola referencia del problema en internet.

Tras muchas pruebas se consiguió acotar el error y encontrar una metodología de desplegar los servicios Web para no haya ningún problema, aunque la causa exacta siga siendo desconocida. Los servicios Web han de desplegarse de la siguiente manera (se indica en el manual de usuario):

El servicio Web de WS-PTRLinda se compila junto con el resto del sistema y se despliega automáticamente.

Una vez el sistema está funcionando, se compila el servicio de persistencia rlhibox, se crea el paquete hibox.arr y se despliega.

El problema surge cuando los servicios se despliegan a la vez. Se ha comunicado esta situación a los desarrolladores de Apache, y en la actualidad se ha abierto el hilo correspondiente en el sistema de seguimiento de fallos.

ANEXO 6 | **Implementación de WS-PTRLinda extendida**

Este anexo extiende el capítulo de implementación introduciendo algunos elementos que no tienen tanto peso o que simplemente se quedaron fuera de la memoria por falta de sitio.

5.1 Persisten RLinda

Un proceso Web complejo se compone de varios procesos Web más sencillos que deben comunicarse y coordinarse entre sí para alcanzar sus fines. RLinda se sitúa en el centro permitiendo esta comunicación y coordinación. Los procesos Web complejos pueden requerir de cierto tiempo hasta que son completados y pueden poseer varios puntos críticos. La información en RLinda está contenida en memoria lo cual aporta mucha rapidez pero deja el sistema a merced de un fallo del sistema operativo o una pérdida de corriente por poner algún ejemplo. Perder toda la información supone tener que reiniciar el proceso Web complejo.

La capa de persistencia de RLinda aporta la tolerancia a fallos y lo hace mediante el uso de la herramienta Hibernate que utiliza para facilitar la comunicación con la una base de datos MySQL. De esta manera la información se duplica en una base de datos y podrá recuperarse.

5.1.1 Funcionalidades añadidas

Las mejoras añadidas por esta capa son:

Modo de trabajo con persistencia fuerte.
Modo de trabajo con persistencia débil.
Operación Load
Operación Save
Método de selección del tipo de persistencia: setWriteMode(mode)
En la persistencia débil, métodos de configuración de sus parámetros: setPersistLimit(lim), getPersisteLimit(), setTimerValue(time) y getTimerValue().

A lo largo de este capítulo se mencionan algunos nombres y conceptos que se aclaran a continuación:

Tuplas escritas: Aquellas tuplas que ya han sido escritas en el espacio de tuplas de RLinda.

Tuplas borradas: Al igual que con las tuplas escritas, llamamos tuplas borradas a aquellas que tras una operación ***in(template)*** ya han sido extraídas del espacio de tuplas.

Tupla persistente: Una tupla es persistente si cuando existe en el espacio de tuplas también existe una copia suya en la base de datos o si cuando ha sido borrada del espacio de tuplas también lo ha sido de la base de datos.

Operaciones de sincronizado o de sincronización: en persistencia débil, las operaciones que la red de RLinda realiza para guardar en la base de datos las tuplas que ya han sido escritas y borrar las que han sido borradas.

5.1.2 Tecnologías utilizadas

La capa de persistencia de WS-PTRLinda se basa en la herramienta Hibernate para gestionar la comunicación con una base de datos MySQL. Estos elementos se describen a continuación quedando extendidos en el anexo correspondiente a la implementación de sistema extendida.

Hibernate

Hibernate es una herramienta de mapeo objeto-relacional para la plataforma Java, su objetivo es solucionar las diferencias entre dos modelos de información distintos entre sí, la base de datos tradicional y el modelo orientado a objetos. Esta herramienta permite abstraer al programador de la labor de escribir las consultas SQL correspondientes para almacenar los atributos de sus objetos en una base de datos, añadiendo una sobrecarga mínima al sistema.

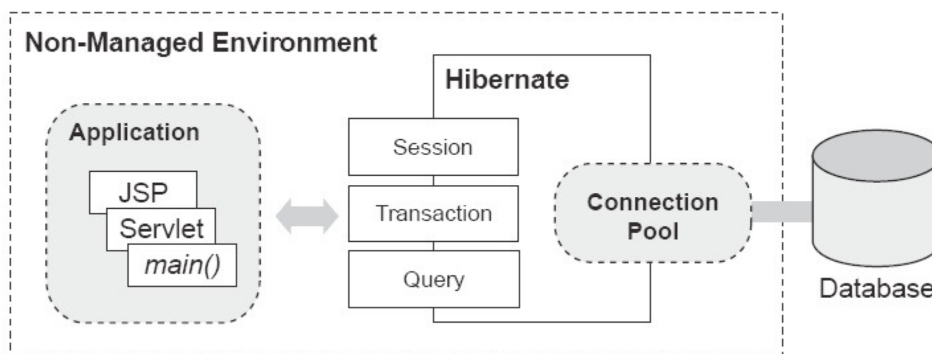


Fig. 26 Dónde se sitúa Hibernate

Hibernate utiliza esquemas XML definidos por el usuario para saber cómo debe convertir un objeto de una clase Java concreta a una tabla específica de la base de datos. Soporta las principales bases de datos que existen empleando para comunicarse con ellas unas clases intermedias llamadas dialectos (Dialect). Un Dialect no es más que una clase Java, de manera que está permitido que el usuario cree sus propios dialectos para permitir la conexión con bases de datos inicialmente no soportadas. Hibernate ofrece también su propio lenguaje de consultas HQL, para interacciones más complejas con la base de datos. La herramienta es software libre y está distribuida bajo los términos de la licencia GNU LGPL.

En nuestro caso, emplearemos Hibernate para añadir persistencia a WS-PTRLinda guardando una copia actualizada en tiempo real de la información contenida en memoria.

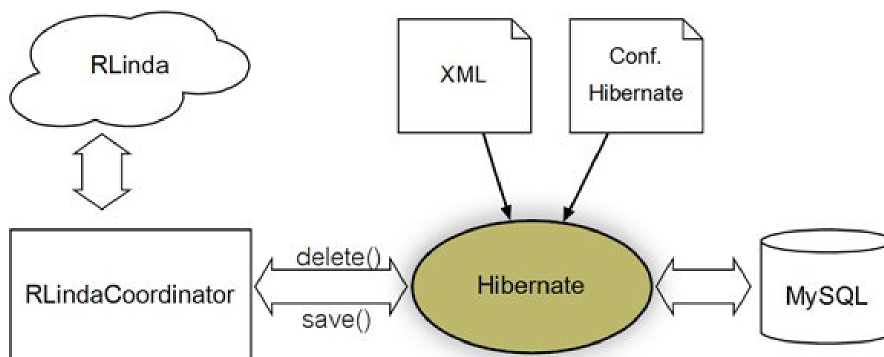


Fig. 27 Dónde se sitúa Hibernate en WS-PTRLinda

Configuración de Hibernate
Hibernate se configura mediante un fichero de configuración externo de nombre *hibernate.cfg.xml*. En él se definen las propiedades básicas de Hibernate así como el driver a utilizar para conectar con la base de datos, el pool de conexiones que se empleará, usuario y contraseña de la base de datos y otros.

Esta forma de configuración aporta una gran versatilidad a Hibernate y por lo tanto a nuestra capa de persistencia. Por ejemplo, para cambiar la base de datos que se está empleando solo sería necesario modificar unas pocas propiedades en el fichero de configuración sin llegar a tocar una sola línea de código.

A continuación se muestra parte el fichero de configuración de Hibernate para Persistent RLinda.

```
<hibernate-configuration>
   <session-factory>
     <property name="show_sql">true</property>
     <property name="format_sql">true</property>
     <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
     <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
     <property         name="connection.url">jdbc:mysql://localhost/RLindaDB</property>
     <property name="connection.username">root</property>
     <property name="connection.password">rlinda</property>
     ...
```

Este primer conjunto de propiedades hace referencia a la base de datos que empleamos.

- dialect: el dialecto de Hibernate que vamos a utilizar. En nuestro caso, el de MySQL. Aunque Hibernate es capaz de detectar por si mismo que dialecto debe emplear, se prefiere que quede plasmado en el archivo de configuración.
- connection.driever.class: Driver de conexión con la base de datos: JDBC para MySQL.
- connection.url: URL de acceso a la base de datos.
- connection.username: nombre de usuario en la base de datos.
- connection.password: contraseña de dicho usuario.

El siguiente bloque de propiedades hace referencia al pool de conexiones empleado, lo veremos en el punto siguiente.

Pool de conexiones de C3P0
En la comunicación con la base de datos la parte más costosa supone establecer la conexión, en términos de código Java, obtener un objeto de tipo *Connection*. Normalmente se aprovecha una misma conexión con la base de datos para realizar varias interacciones pero el caso de RLinda es diferente por la naturaleza del servicio, en el que muchos clientes depositan o extraen información y desaparecen. En entornos con muchos accesos a la base de datos, aun mas cuando son para realizar unas pocas operaciones sencillas, resulta muy costoso tener que solicitar una conexión cada vez. Por ello se emplean los pool de conexiones. Un pool de conexiones se sitúa entre medio del cliente y la base de datos (como puede verse en fig. 9) y mantiene siempre un número dado de conexiones abiertas. De esta manera el pool de conexiones proporciona al cliente una conexión que ya está abierta ahorrando todo el proceso de establecer una nueva.

Hibernate emplea su propio pool de conexiones por defecto aunque nos avisa de que no se recomienda para la fase de producción de la aplicación. Por ello también incluye el pool de conexiones C3P0, que se configura en el mismo archivo externo que Hibernate.

hibernate.cfg.xml

```
<property
name="connection.provider_class">org.hibernate.connection.C3P0ConnectionProvider</property>
<property name="c3p0.acquire_increment">3</property>
<property name="c3p0.idle_test_period">100</property> <!-- seconds -->
<property name="c3p0.max_size">100</property>
<property name="c3p0.min_size">10</property>
<property name="c3p0.max_statements">0</property>
<property name="c3p0.timeout">5000</property> <!-- seconds -->
```

- c3p0.max_size: máximo número de conexiones que puede mantener
- c3p0.min_size: mínimo número de conexiones.
- c3p0.timeout: el tiempo en segundos que una conexión puede permanecer en el *pool* sin ser descartada.
- c3p0.acquire_increment: cuantas conexiones intentará adquirir C3P0 cuando todas las demás estén en uso.
- c3p0.idle_test_period: cada este número de segundos *C3P0* chequeará las conexiones ociosas.
- c3p0.max_statement: *C3P0* permite también mantener una caché de sentencias SQL preparadas (clase PreparedStatements), en nuestro caso está desactivada.

Con estos parámetros el *pool de conexiones* mantendrá unas pocas conexiones abiertas (un mínimo de 10) cuando el uso del sistema sea bajo pero puede aumentar hasta 100 si hay una gran concurrencia de clientes. Al tener la configuración en un fichero externo se pueden variar estos parámetros para ajustarse a un escenario más concreto.

La clase HibernateUtil

Hibernate no funciona como un objeto Java normal que puede ser creado y llamado desde cualquier otra clase, Hibernate funciona en *background*. Por ello, tanto para iniciarlo como para interactuar con él empleamos una clase sencilla de nombre HibernateUtil.
El objetivo de HibernateUtil es:
Iniciar Hibernate.
Limpiar la base de datos al comenzar la ejecución de RLinda.
Ofrecer un método de comunicación con Hibernate.

Hibernate es accedido desde el código Java mediante un objeto de tipo *SessionFactory*. Es trabajo de ***HibernateUtil*** construir este objeto y que sea accesible desde otros puntos del código.

HibernateUtil.java

```
 private static SessionFactory buildSessionFactory(String cfgFileName, String databaseName) {
     try {
         iniMySQLDB(databaseName);

         // Create the SessionFactory from cfgFileName configurationFile.
         SessionFactory sf = new Configuration().configure(cfgFileName).buildSessionFactory();
   return sf;
       }
       catch (Throwable ex) {
         System.err.println("Initial SessionFactory creation failed." + ex);
```

```
            throw new ExceptionInInitializerError(ex);
        }
    }

  public static void start(String cfgFileName,String DBName) throws Exception{
        if (sessionFactory.isClosed()){
            sessionFactory = buildSessionFactory(cfgFileName,DBName);
        }else{
            logger.debug("[HIBERNATE UTIL] SessionFactory is already open!!");
        }
  }
```

Desde cualquier punto del código Java se puede llamar a HibernateUtil para que nos proporcione la referencia a *SessionFactory*. Este *SessionFactory* nos permitirá abrir una conexión con la base de datos en forma de objeto *Session*. Una vez obtenida es necesario crear un objeto de tipo *Transcation* a partir del cual podemos o utilizar uno de los métodos que ofrece Hibernate o lanzar una consulta en el lenguaje SQL propio de Hibernate: HQL.
El siguiente fragmento de código muestra cómo se interacciona con Hibernate.

```
Session session = HibernateUtil.getSessionFactory().openSession();
 Transaction tx = session.beginTransaction();
 try{
    tx.save(flarlar);
 }
```

Mapeo de Objetos mediante ficheros XML
Hibernate convierte la información contenida en el modelo orientado a objetos al modelo relacional empleado en las bases de datos. Estos dos modelos son muy diferentes entre sí y esta traducción no es para nada obvia. Por ello Hibernate se apoya en ficheros XML externos escritos por el desarrollador para saber cómo debe llevar a cabo esta conversión.

La clase Tuple de Renew es una clase complicada basada en listas y en otros componentes. En cambio, este clase Tuple puede expresarse en forma de String pudiendo hacer el camino de vuelta de nuevo. Por ello se desarrolla una clase TupleSpace Tuple que envuelve a la clase Tuple a la hora de ser manejada por Hibernate. De esta forma podemos definir un patrón XML sencillo (TupleSpaceTuple.xml) que se muestra a continuación:

TupleSpaceTuple.xml

```
 <?xml version="1.0" encoding="UTF-8"?>
 <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
 "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
 <hibernate-mapping>
    <class name="org.serviciosweb.rlinda.persistence.TupleSpaceTuple" table="TUPLE_SPACE">
       <id name="id" column="id" type="long"> </id>
       <property name="tuple" column="tuple" type="string"></property>
       <property name="expirationTime" column="expirationTime" type="long"></property>
  </class>
</hibernate-mapping>
```

La clase TupleSpaceTuple posee un campo *String* tuple donde se inserta la tupla y un campo *pid* que se corresponde con el id mostrado en el fichero XML y métodos get/set para acceder a los campos. Más tarde en la capa de Temporización se añadió un nuevo campo *expiratión time* que se explica más adelante.

TupleSpaceTuple.java

```
public class TupleSpaceTuple {
   long id;
   String tuple;
   long expirationTime;
   public long getId() {
      return id;
   }
   public void setId(long id) {
      this.id = id;
  }
  public String getTuple() {
      return tuple;
   }
   public void setTuple(String tuple) {
      this.tuple = tuple;
   }
   public long getExpirationTime() {
      return expirationTime;
   }
   public void setExpirationTime(long expirationTime) {
      this.expirationTime = expirationTime;
   }
}
```

Por lo tanto, para que Hibernate almacene una tupla en la base de datos antes debe ser envuelta en una objeto de tipo TupleSpaceTuple y posteriormente comunicar con Hibernate para que la escriba en la base de datos. Hibernate empleará el siguiente fichero de configuración TupleSpaceTuple.xml para saber cómo realizar esta traducción:

Volviendo con el fichero de configuración XML, podemos observar que se indica de forma clara como debe traducirse cada elemento:

name - nombre del campo del objeto.
column - columna de la tabla de la base de datos en donde se debe insertar.
type - tipo de la columna (String, Integer…).

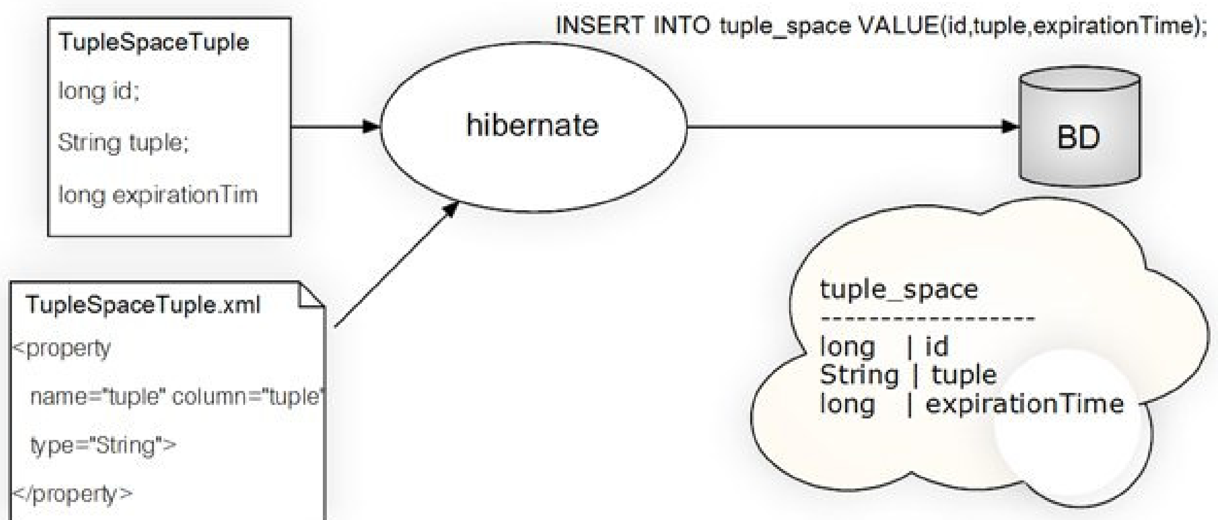


Fig. 28 Traducción de tuplas

5.1.2 ¿Qué datos deben ser persistidos?

En WS-PTRLinda circulan diferentes tipos de datos pero no se debe olvidar que no estamos trabajando en un entorno cerrado sino en un sistema abierto en el que los clientes se conectan a través de internet. Esto quiere decir que la comunicación del sistema está regida por los protocolos de comunicación que se empleen y por sus características y limitaciones (por poner un ejemplo: el tiempo máximo de de conexión).
Las diferentes fuentes de información en el sistema WS-PTRLinda son:

- tuplas residentes en el espacio de tuplas procedentes de operaciones de escritura ***out()***.
- patrones de operaciones ***rd()*** pendientes de encontrar una tupla que se ajuste a ellos.
- patrones de operaciones ***in()*** pendientes de encontrar una tupla que se ajuste a ellos.
- datos de configuración de persistencia y tipo de *matching*.

Debemos recordar que WS-PTRLinda se sitúa en el centro de la comunicación y que a él se accede mediante protocolos SOAP, REST o RMI. Los clientes se conectan al sistema y después ejecutan una operación. En el caso de las operaciones de lectura, los clientes permanecen bloqueados hasta que una tupla que corresponde con el patrón proporcionado es encontrada. Si se produce un fallo en WS-PTRRLinda y es necesario restaurarlo, todas esas conexiones se cierran y se pierden, de manera que no sería posible devolver el resultado. La figura 11 muestra un ejemplo de desconexión repentina de un cliente que sirve de ayuda a la explicación del caso anterior.

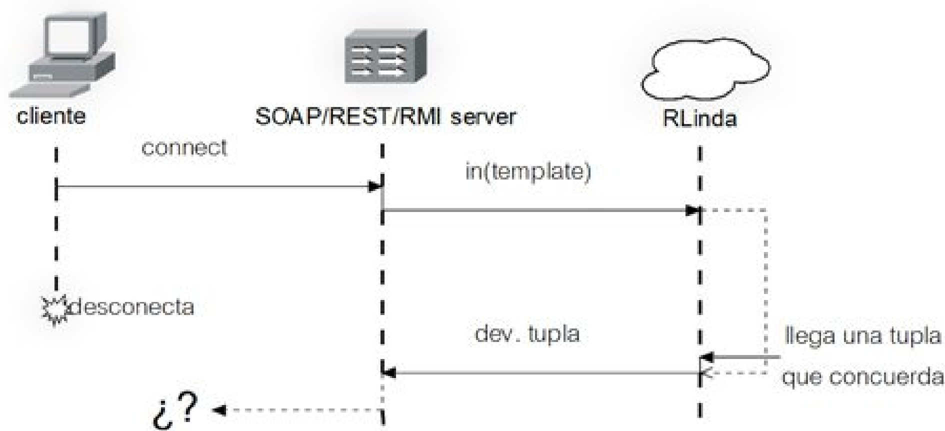


Fig. 29 Problema de desconexión de un cliente

Si se produce un fallo y hay que restaurar WS-PTRLinda, las conexiones con los clientes habrán caído. Por ello no necesitamos preocuparnos de guardar una copia de los patrones de las operaciones de lectura. Una operación de escritura *out(t)* en cambio, introduce una tupla en el espacio de tuplas y el cliente sigue su curso. El espacio de tuplas contiene todos los mensaje y los datos y es la información que más nos urge duplicar para poder restaurarla, de manera que si que habrá que tenerla en cuenta a la hora de garantizar la persistencia.

Estado de un objeto en Hibernate

Hibernate define y soporta los siguientes estados de objeto:

- Transitorio - un objeto es transitorio si ha sido recién instanciado utilizando el operador new, y no está asociado a una *Session* de Hibernate. No tiene una representación persistente en la base de datos y no se le ha asignado un valor identificador. Las instancias transitorias serán destruidas por el recolector de basura si la aplicación no mantiene más una referencia.
- Persistente - una instancia persistente tiene una representación en la base de datos y un valor identificador. Puede haber sido guardado o cargado, sin embargo, por definición, se encuentra en el ámbito de una *Session*. Hibernate detecta cualquier cambio realizado a un objeto en

estado persistente y sincroniza el estado con la base de datos cuando se complete la unidad de trabajo.

- Separado - una instancia separada es un objeto que se ha hecho persistente, pero su *Session* ha sido cerrada. La referencia al objeto todavía es válida, por supuesto, y la instancia separada podría incluso ser modificada en este estado. Una instancia separada puede ser re-unida a una nueva *Session* más tarde, haciéndola persistente de nuevo (con todas las modificaciones).

Las operaciones que emplea WS-PTRLinda son:

- save() - escribe el objeto en la base de datos haciéndolo persistente.
- delete() - Borra un objeto de la base de datos.
- load() - trae un objeto desde la base de datos.

Una operación *out()* producirá una llamada a *save()*, para que la tupla se copie en la base de datos.
Una operación *in()* que encuentra una tupla que concuerde con su patrón, llamará a *load()* para cargar el objeto correspondiente a dicha tupla desde la base de datos y posteriormente lo borrará con *delete()*.

Generador de persistencia

Todo objeto que vaya a ser escrito por Hibernate en la base de datos debe tener asignado un id, en caso de que no será Hibernate quien genere uno. Este id debe de ser único ya que funcionará como clave primaria en la base de datos. En WS-PTRLinda existen situaciones en modo de trabajo de persistencia débil en las que es necesario que una tupla posea un id mucho antes de que vaya a ser guardada en la base de datos. Por este motivo WS-PTRRLinda genera sus propios id mediante varias transiciones y lugares de su red.

Cuando nos refiramos a este id hablaremos siempre de *persistence id* o *pid* ya que dentro de WS-PTRLinda las operaciones de lectura poseen ya un identificador que se llama *id* que no tiene nada que ver. En resumen cada tupla tiene un *pid*, que es único y sirve para identificarla en la base de datos. El coordinador de WS-PTRLinda proporciona los siguientes métodos relacionados con el generador de PID.

- getPID() - devuelve un *pid*. Si no hay ningún *pid* en el lugar *recycled pool* aumenta en +1 el valor que haya en el lugar *newPID* y lo suministra.

- recyclePID() - RLinda permite reutilizar los *pid* usados. El sistema está pensado para soportar un gran número de operaciones, cada una de ellas trabajando con una tupla que poseerá su propio *pid*. Por ello se reutilizan los *pid* de las tuplas que se borran del sistema.

- inserPID() - introduce un valor inicial a partir del cual comenzar a generar *pids*, unicamente se emplea al restaurar una copia de seguridad. Cada tupla que se restaura posee un *pid* que se ha generado con anterioridad, pero como RLinda se ha reiniciado, sucesivas llamadas a *getPID()* devolverían /pids/ partiendo de cero y podría dar lugar a *pids* repetidos y en última instancia, errores en la base de datos por tratar de insertar diferentes objetos con la misma clave primaria.

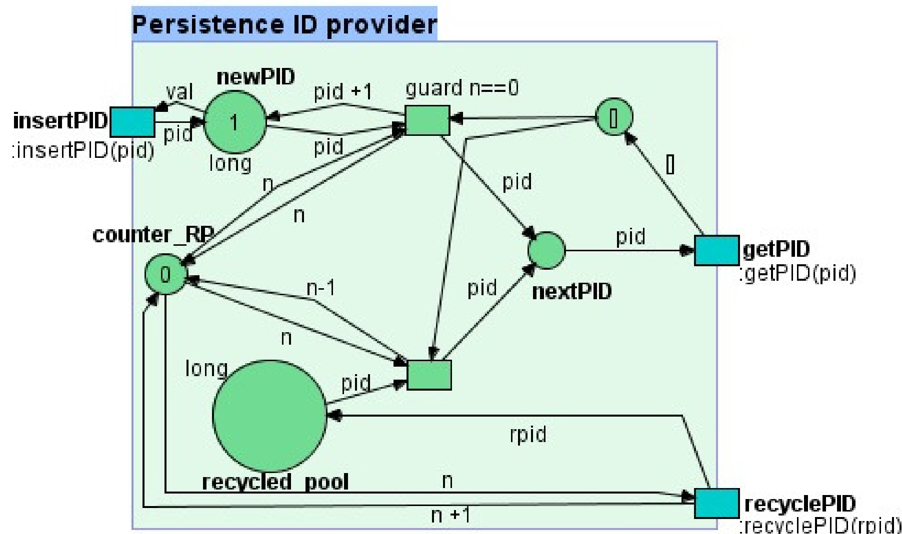


Fig. 30 Generador de pid

En la fig.12 pueden verse las transiciones que se invocan con las diferentes llamadas. Veamos la operación *getPID* que proporciona un nuevo *pid*. En el lugar *nextPID* está esperando el siguiente valor que debe ser devuelto. *GetPID* retira es valor y lo devuelve al usuario. A su vez ha depositado un token en el lugar de arriba a la derecha. Este token producirá el disparo de dos posibles transiciones. Si se cumple la guarda *guard n == 0* significará que no hay *pid* en el lugar *recycled* Pool así que se consume uno nuevo del lugar *newPID* que se deposite en *nextPID* para la próxima llamada de la función *getPID*. Si no se hubiera cumplido la condición de la guarda, el *pid* depositado en *nextPID* hubiera procedido de *recycledPool*.

MySQL

MySQL es un sistema de gestión de base de datos relacional, multihilo y multiusuario. Emplea un tipo de comunicación cliente-servidor y es una de las bases de datos Open Source más populares, principalmente gracias a su buen rendimiento y a que es sencilla de usar.

PRLinda almacena toda su información en memoria y únicamente emplea la base de datos para proporcionar persistencia. Únicamente necesitamos guardar en ella las tuplas almacenadas en el espacio de tuplas y unos pocos datos de configuración de RLinda. Para tratar con esta información es suficiente con una base de datos sencilla.

- Tabla tuple_space: almacena las tuplas que hay en el espacio de tuplas.
  long id - *clave primaria* que identifica cada tupla.
  String tupla - la tupla en formato *String*.
  long expirationTime - el tiempo de expiración de la tupla (ver capa de temporización).

- Tabla de configuración: almacena información de configuración de RLinda.
  int writeMode - tipo de persistencia que se esta empleando (1 - débil, 2 - fuerte).
  long time - tiempo del timer en persistencia débil (ver persistencia débil)
  int limit - límite de tuplas en la persistencia débil (ver persistencia débil)
  int matching - tipo de matching.

Mientras que la tabla tuple_space se emplea contantemente para escribir o borrar las tuplas que entran/salen del sistema, la segunda solo se llena a la hora de hacer la copia de seguridad salvando los parámetros con los que está trabajando PRLinda en ese instante.

5.1.2 Organización de Persistent RLinda como servicio Web

Persistent RLinda se implementa en dos servicios Web diferenciados, implementados y desplegados mediante Apache AXIS2 en el interior de un servidor Apache Tomcat embebido. Un servicio Web es el propio coordinador de PRLinda que ofrece las directivas básicas de RLinda y otro ofrece las funciones de creación y restauración de una copia de seguridad. Este primero permite acceder a él mediante protocolos SOAP y REST mientras que el responsable de las copias de seguridad emplea *SOAP with Attachment*. Esto quiere decir que además de crearse una copia de seguridad esta se envía al cliente y para restaurarla es el cliente quien debe enviársela a RLinda.

Desde el punto de vista de los servicios Web, así queda el sistema.

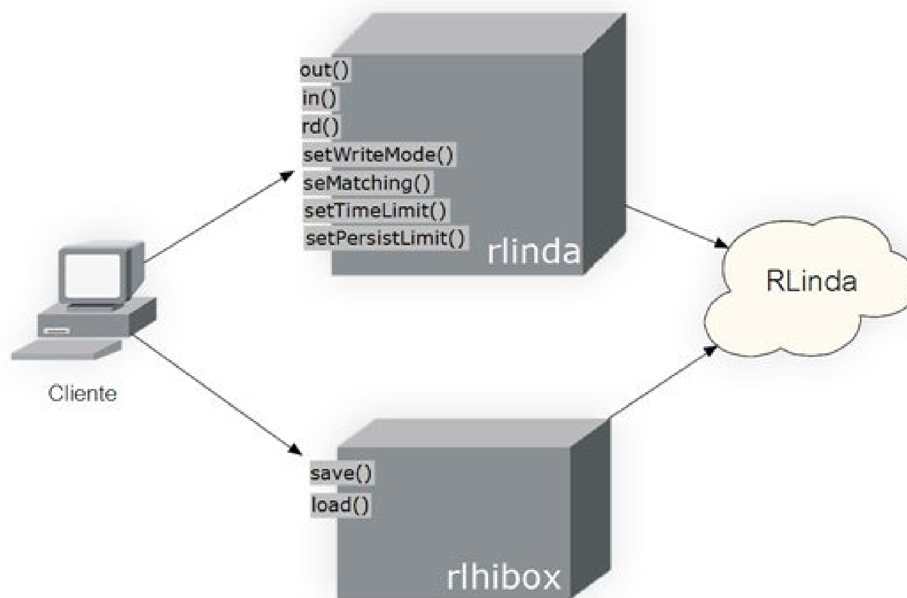


Fig. 31 Servicios Web en Persisten RLinda

Ambos servicios acceden al coordinador de PRLinda aunque lo hacen para realizar acciones diferentes. El servicio de ***rlinda*** ofrece las tres directivas de coordinación y los métodos de configuración relativos a la persistencia y tipo de matching.

El servicio Web encargado de las copias de seguridad, ***rlhibox***, accede al coordinador únicamente para hacer uso de las operaciones Load y Save. Estas operaciones suponen el bloqueo de toda operación de escritura *out* o de lectura destructiva *in* mientras se crea o se restaura la base de datos para evitar problemas de coherencia. Las operaciones de lectura no destructivas *rd* pueden seguir ejecutándose pues no modifican el espacio de tuplas.

5.1.3 Arquitectura del sistema

La clase Java que representa la capa de persistencia recibe el nombre de HibernateBox ya que su función es comunicarse con Hibernate para guardar/borrar los datos necesarios de la base de datos y, una vez hecho, interaccionar el coordinador. La figura 14 muestra la arquitectura del sistema.

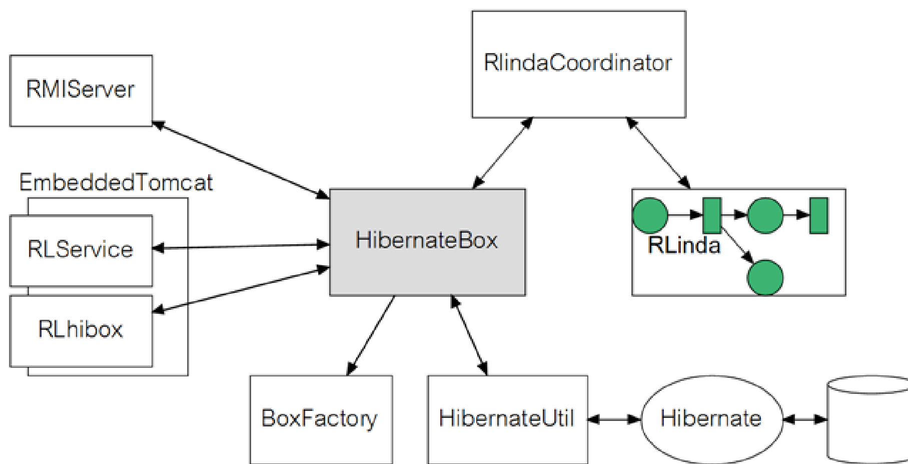


Fig. 32 Arquitectura de PRLinda

El Coordinador y la red de PRLinda

El coordinador de RLinda comunica la clase HibernateBox con la red RLinda, es la representación de la red de RLinda en forma de objeto Java. Algunas transiciones de la red de RLinda han sido modificadas al igual que el coordinador que debe reflejar estos cambios.

- write(v, pid) - Las transiciones write pasan a tener dos parámetros de entrada, el pid de la tupla y la propia tupla.
- writeF(v,pid) - Transición de escritura en el espacio de tuplas para la persistencia débil. Escribe además la tupla en el lugar writenStorage para que sea persistida cuando corresponda.
- insert(v,pid) - Escribe la tupla en el espacio de tuplas. Usada al cargar la información de la base de datos.
- control del lock - Transiciones para controlar el lock a la hora de bloquear escrituras y lecturas mientras se restaura la base de datos o se hace una copia.

Estas nuevas transiciones pueden observarse en la figura 35.

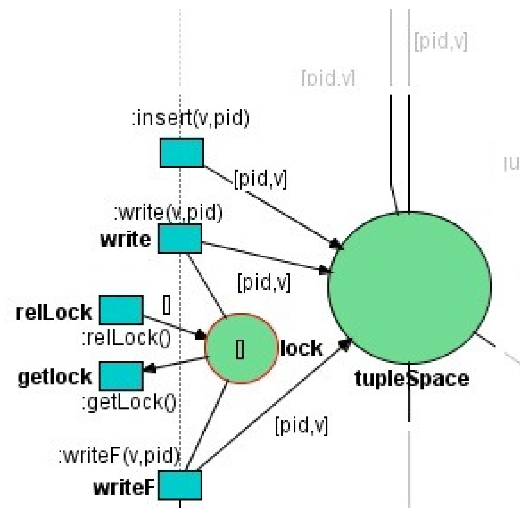


Fig. 33 nuevas transiciones write

Las transiciones *write* escriben una tupla en el espacio de tuplas al disparase, siempre y cuando el token del lugar *lock* no haya sido tomado. En ese caso no podrán dispararse y tendrán que esperar a que el *lock* vuelva a su sitio habitual.

Estos son los cambios que afectan a todo el sistema, se han introducido otros cambios relacionados exclusivamente con el modo de persistencia débil y se explican en la correspondiente sección de esta memoria

Hibernate Box

Hibernate box es la capa de presentación de la persistencia, por lo tanto encapsula toda la lógica de aplicación y ofrece las nuevas funcionalidades a través de sus métodos. Además de las operaciones básicas de RLinda, se ofrecen otros métodos que configuran el tipo de persistencia y otros parámetros. La totalidad de los métodos se incluyen en el anexo correspondiente.

- **out(String tuple)** – escribe una tupla en el espacio de tuplas.
- **in (String pattern) –** devuelve una tupla de espacio de tuplas que se ajusta al patrón dado. Si no se en encuentra ninguna se bloquea. Retira la tupla devuelta.
- **rd (String pattern) –** operación de lectura pero no destructiva con la tupla devuelta.
- **load(byte filedata, String filename) –** restaura la base de datos y el estado de PRLinda a través de una copia de seguridad *filedata* de nombre *filename*.
- **save(String fileName) -** crea una copia de seguridad de la base de datos. Está función se utiliza desde el lado del cliente RMI y transmite el archivo de copia de seguridad a su directorio.
- **setWriteMode(int mode)** - Establece el modo de persistencia (1 – débil , 2- fuerte).
- **setPersistLimit (int limit) -** en persistencia débil, límite de tuplas que pueden permanecer sin persistir.
- **setPersistTime(long time) –** segundos cada los cuales se realizan operaciones de sincronización.

- El resto de métodos de HibernateBox son:

- **iniHibernate(String DBName) -** LLamado desde el constructor, inicializa las librerías de Hibernate si no han sido inicializadas ya.

- **writeFast (Tuple tuple) -** Persistencia débil. Escribe *tuple* en RLinda, deja las labores de persistencia a la propia red según estén configurados los parámetros. Es llamado por write() en función del tipo de persistencia actual.
- **takeSlow(Tuple tuple) -** persistencia fuerte. Realiza una operación *in(pattern)* en RLinda. Si encuentra una tupla que coincida con el patrón *pattern* la extrae del espacio de tuplas, borra dicha tupla de la base de datos y se la devuelve al proceso que llamó la operación.
- **takeFast(Tuple pattern) -** persistencia débil. Realiza una operación *in(tuple)* en RLinda, deja las labores de persistencia a la propia red según estén configurados los parámetros. Es llamado por take() en función del tipo de persistencia actual.
- **setMatching(int type) -** cambia el tipo de *matching* en RLinda. (1 - weak_matching, 2 - Strong_matching, 3 - attribute_matching).
- **getWriteMode() -** devuelve el modo de escritura actual.
- **getPersistLimit() -** devuelve el límite de tuplas en persistencia débil.
- **closeHibernate() -** cierra Hibernate.
- **restartHibernate -** reinicia Hibernate.
- **load(byte filedata, String FileName) -** restaura la base de datos y el estado de RLinda a través de un fichero copia de seguridad *filedata* de nombre ***fileName***.
- **loadBackupFile(String file) -** carga la información de la copia de seguridad en la base de datos y llama a loadRLindaState para que escriba las tuplas y los datos de configuración en RLinda. Es invocado desde *load()*.
- **loadRlindaState() -** restaura el estado de RLinda escribiendo en el espacio de tuplas las tuplas contenidas en la base de datos. Es invocado desde *loadBackupFile()*.
- **save(String fileName) -** crea una copia de seguridad de la base de datos. Está función se utiliza desde el lado del cliente RMI y transmite el archivo de copia de seguridad a su directorio.
- **saveBackupFile(String file) -** conecta con la base de datos y crea un archivo de texto que contiene toda la información de tuplas de la base de datos. Es llamado por *save()*.
- **persistTuple(Tuple tuple) -** introduce una tupla en la base de datos. Se llama desde la red de RLinda en las operaciones de sincronizado de la persistencia débil.
- **detachTuple(Tuple tuple) -** elimina una tupla en la base de datos. Se llama desde la red de RLinda en las operaciones de sincronizado de la persistencia débil.

HibernateBox implementa la interfaz HibernateRemote de cara a ser referenciada como un objeto RMI y cuyo código se adjunta a continuación:

HibernateBoxRemote.java

```
package org.serviciosweb.rlinda.common.interfaces;
import java.rmi.RemoteException;
import de.renew.unify.Tuple;
import java.io.File;
public interface HibernateBoxRemote extends java.rmi.Remote {

   public String out(String xmlTuple) throws Exception;

   public String out(Tuple tuple) throws Exception;

   public String rd(String pattern) throws Exception;

   public Tuple rd(Tuple pattern) throws Exception;

   public String in(String XMLpattern) throws Exception;

   public Tuple in(Tuple pattern) throws Exception;
```

```
    public void setMatching(int type) throws Exception;

    public void setWriteMode(int mode) throws RemoteException;

    public void setPersistLimit(int newLim) throws RemoteException;

    public int getPersistLimit() throws RemoteException;

    public void setTimeLimit(long time) throws RemoteException;
    public byte[] save(String fileName) throws Exception;

    public void load(byte[] filedata,String fileName) throws Exception;
}
```

BoxFactory

Esta clase ofrece métodos de creación de archivos JAR, descompresión de los mismos y limpieza de archivos de copias de seguridad. Sus métodos son:

openBox(String name,String path) - Descomprime el archivo de nombre *name*.
prepareBoxToSend (String directory, Strinf name) - crea un archivo *JAR* de nombre *name*.jar y devuelve un String con la ruta de dicho archivo.
clean(String path) - limpia los archivos que haya en dado *path*

5.1.4 Tipos de persistencia ofrecidos

Existen muchos tipos de procesos Web cada uno con diferentes necesidades. Mientras uno puede necesitar que la comunicación sea lo más rápida posible y no tenga grandes problemas en perder información si ocurre un fallo, otro puede tener varios puntos críticos y requerir estar seguro de que sus datos están garantizados y que, en caso de fallo, podrá recuperar su estado y seguir en el punto en el que estaba.

Con un escenario tan amplio de posibilidades se han desarrollado dos formas distintas de proporcionar persistencia, una fuerte que garantiza que en todo momento en la base de datos hay una copia exacta del contenido de la memoria y otra que realiza las tareas de escritura o borrado de información en la base de datos periódicamente y de acuerdo a criterios configurables. De esta forma el sistema es flexible en función de a qué escenario se enfrenta y se deja la puerta abierta a un trabajo futuro de un balanceador que decida en función de las condiciones de tráfico si debe aplicar una u otra forma.

El tipo de persistencia que se está empleando es también llamado modo de escritura (writeMode), que puede ser lento o rápido en función de si la persistencia es fuerte o débil respectivamente.

Persistencia fuerte

Este modo de persistencia pretende garantizar que en la base de datos donde se duplica la información hay una copia exacta de la información que reside en memoria. La únicas operaciones que producen cambios en el espacio de tuplas de RLinda son *out* e *in*.

- **out(t) -** Antes de introducir la tupla en RLinda esta es guardada en la base de datos.

- **in(template)** - La plantilla template se introduce en RLinda, en el momento en se encuentra una tupla que corresponde con template, se busca y borra dicha tupla de la base de datos y para acabar se devuelve al proceso que invocó el método.

En las figuras 36 y 37 pueden verse los pasos que siguen las operaciones *out* e *in* ,respectivamente, en la persistencia fuerte.

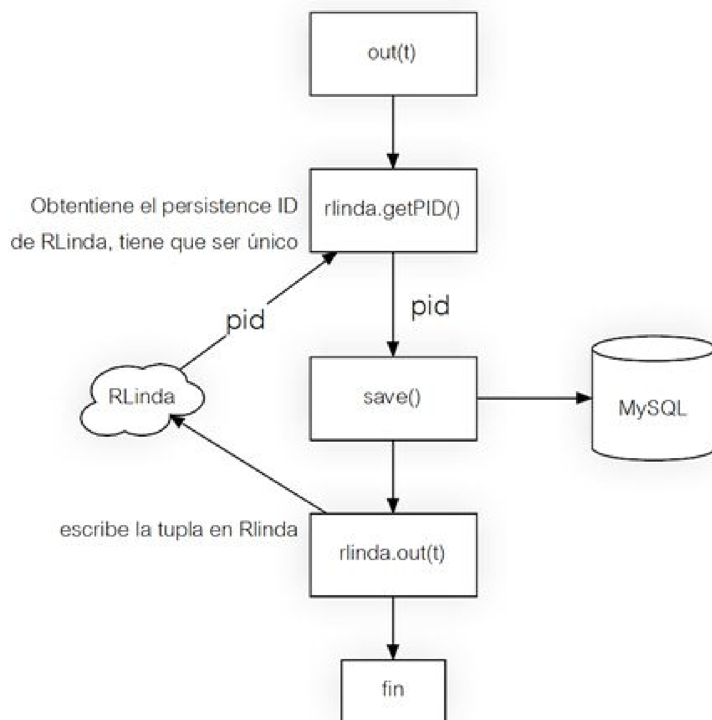


Fig. 34 Pasos de una operación out en persistencia fuerte

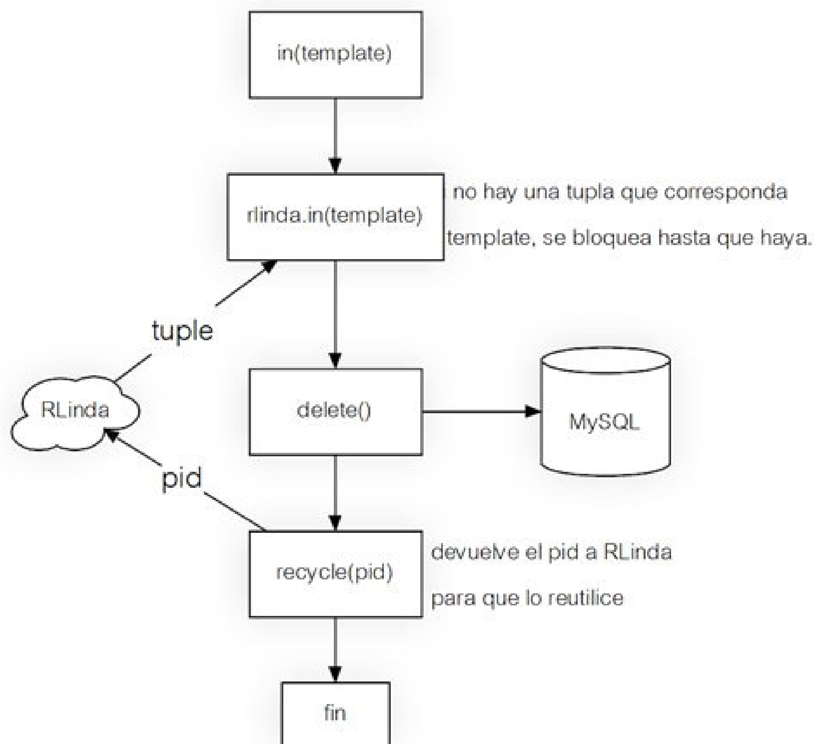


Fig. 35 Pasos de una operación in en persistencia fuerte

Persistencia débil

Este modo de persistencia proporciona una forma de acceso a PRLinda rápida pero sin renunciar a proteger la información en una base de datos. A diferencia del modo fuerte, en que en cada método invocado por el cliente produce un acceso a la base de datos para introducir o borrar información del sistema, las labores de persistencia no se realizan en la propia capa de persistencia sino que se delegan a la red deP RLinda.

La red de PRLinda realiza periódicamente, acorde a criterios configurables, el copiado o borrado de tuplas en la base de datos(labores de sincronizado). De esta forma el cliente no se ve ralentizado por accesos a la base de datos. La contrapartida es obvia, las tuplas insertadas o borradas entre una de estas labores de sincronizado están expuestas a fallos.

Existen dos valores a tener en cuenta a la hora de decidir cada cuanto se van a realizar las operaciones de sincronizado. Estos son:

- **Tiempo** - Número de segundos cada los cuales se van a volcar las tuplas en la base de datos (en escritura) o en lectura. Se accede y configura mediante los métodos getTimerVal() y setTimerVal(delay).

- **Número de tuplas** - Cuando el numero de tuplas sin volcar en la base de datos (o sin borrar) sobrepase este valor frontera, realizar las operaciones de sincronizado. Se accede y configura este valor mediante los métodos getPersistLimit() y setPersistLimit(new lim);

Se llevan a cabo varias modificaciones en la red de PRLinda. A continuación se muestran y se describe el mecanismo de borrado de tuplas en persistencia débil ligeramente mas sencillo que el de escritura y perfecto para explicar al lector como se realizan las labores de sincronizado con la base de datos. Pueden encontrarse la totalidad de modificaciones realizada en la red de PRLinda en el anexo correspondiente.

Almacén intermedio de tuplas borradas y sistema de borrado

Las tuplas llegan procedentes de la función de matching y se almacenan en el lugar *TakenStorage*. Las tuplas van pasando al espacio de *prePersistence* hasta que el contador (que representa el número de tuplas que hay en *prePersistence*) llega al valor frontera de límite de tuplas. Desde esta zona son leídas por la transición deleteTakens que se encarga de borrarlas de la base de datos. Para que pueda dispararse esta transición el contador ha tenido que sobrepasar el límite impuesto, una vez sucedido, el contador se desplaza hacia la derecha donde se produce un bucle (hasta que el contador vale cero) que dispara la transición que persiste una tupla una vez por cada tupla en preErase.

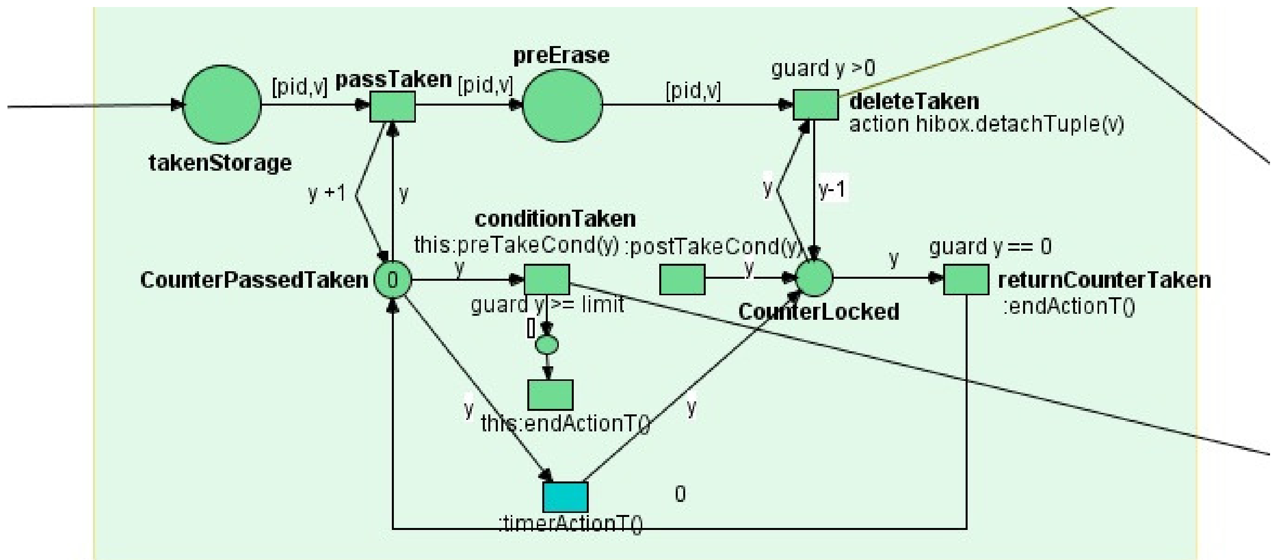


Fig. 36 Mecanismo de borrado de tuplas en persistencia débil

El mecanismo de escritura es similar pero llama a Hibernate para que guarde las tuplas en la base de datos. Además está dotado de una transición más cuya función es que antes de cada operación de borrado de tuplas, se realicen antes las operaciones de escritura (por razones de coherencia).

Valor configurable de límite de tuplas

Se añaden dos transiciones invocables desde el exterior para consultar o modificar el valor.

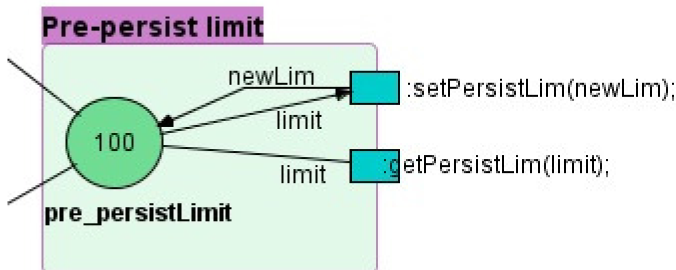


Fig. 37

Valor configurable de tiempo

Se añaden dos transiciones invocables desde el exterior para consultar o modificar el valor. Además se añade una referencia a la clase Timer de java. La clase TimerActioNManager que es llamada por Timer cada vez que pasa el tiempo preestablecido, posee una referencia a PRlinda y llama a las transiciónes timerActionW y timerActionT para que se copien y borren las tuplas a/de la base de datos respectivamente.

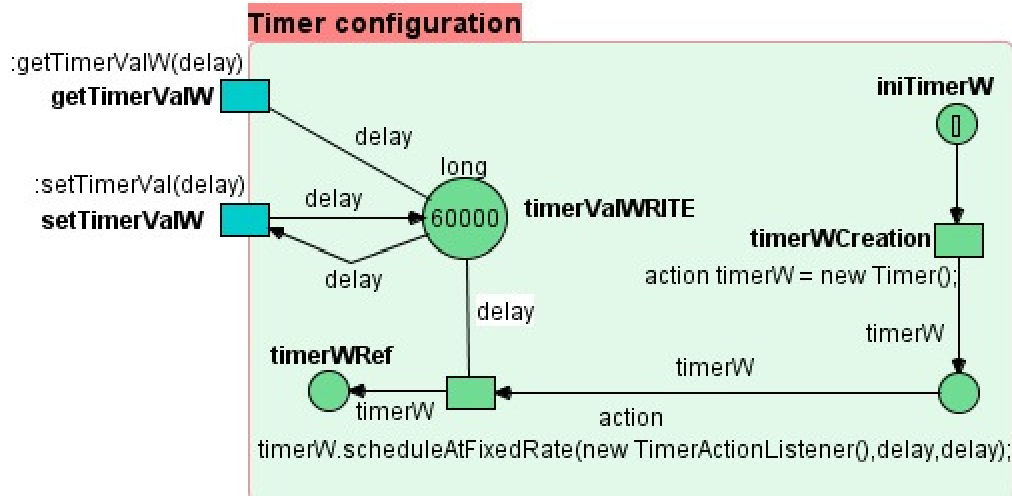


Fig. 38

Almacen intermedio de tuplas escritas y sistema de escritura

Esta sección de la red funciona exactamente igual que el punto anterior, con la excepción de que tiene añadida una transición mas cuyo disparo causa que se guarden las tuplas en la base de datos. la razón para ello es que cada vez que se cumple una condición por la que deben sincronizarse las tuplas del almacén intermedio de tuplas borradas (takenStorage), antes se salvan las escritas. La razón de esto es evitar problemas de intentar borrar tuplas que aun no han sido escritas en la base de datos. Esta situación se describe en detalle más adelante.

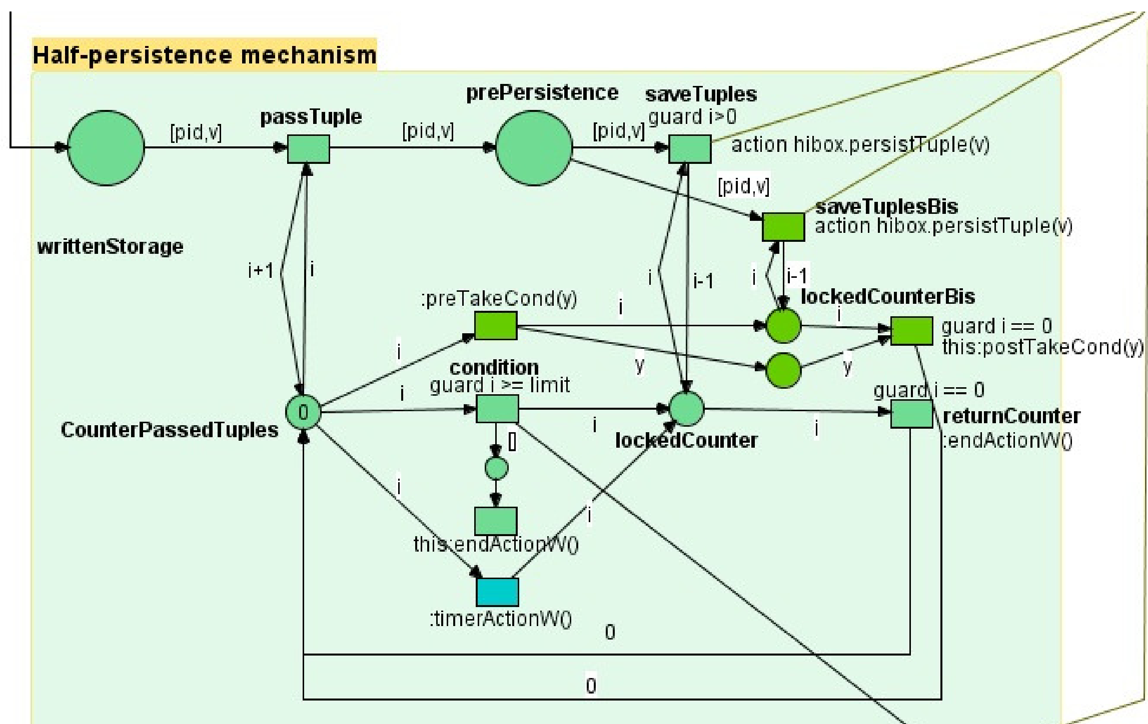


Fig. 39

Nuevas transiciones (writeF y matches)

Estas transiciones realizan lo mismo que sus equivalentes pero además envían una tupla al almacen temporal correspondiente. Si la tupla sale de una transición matches irá a pendientes de borrar (takenStorage), si sale de un writeF a pendientes de escribir (writtenStorage).

A continuación se muestra una figura con la visión general de toda la nueva zona de persistencia.

Una situación problemática

La existencia de varias condiciones que inicien las operaciones de sincronización de el espacio de tuplas con la base de datos puede generar una situación de falta de coherencia entre memoria y base de datos. A esto hay que añadirle que Renew, el programa sobre el que está diseñada la red de RLinda, dispara las transiciones de forma no determinística lo que añade aun más dificultad para preveer algunos comportamientos. Como se muestra en la figura inferior, podría dar lugar a que una tupla intente ser borrada de la base de datos antes de ser escrita, produciendo una excepción y luego tarde o temprano sería escrita (cuando debía haber sido borrada!).

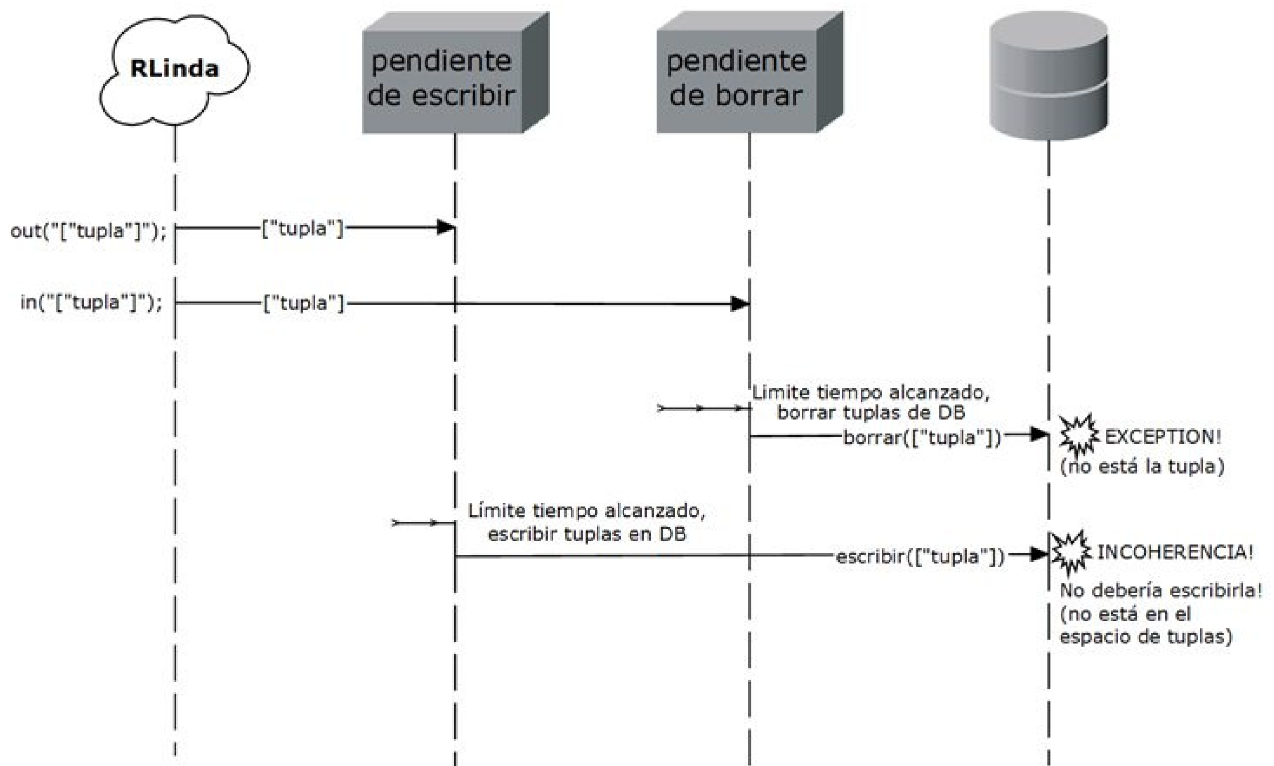


Fig. 40 Problema en la persistencia débil

Para solucionar este problema se emplean dos técnicas conjuntamente.

En el caso de que se cumpla la condición de tiempo para las tuplas pendientes de ser borradas, se guardan todas las tuplas pendientes de escribir antes de eliminar las pendientes de borrar. De esta forma se minimizan los casos en los que se puede producir incoherencia.

Para prevenir los casos restantes, la función de borrado de la base de datos comprueba si hay alguna tupla escrita, si la encuentra se borra normalmente, si no es escrita una tupla comodín

[“FAKETUPLE”] que servirá para alertar al método de escritura. Cuando una tupla va a ser escrita en la base de datos, antes comprueba que no haya una tupla comodín en el lugar correspondiente, si la encuentra la borra y no escribe la tupla.

Como todas las tuplas poseen un id de persistencia único que actúa como clave primaria en la base de datos, podemos emplear este para saber si hay ya una tupla escrita en una posición determinada. De esta manera se solucionan los problemas de coherencia comentados.

5.1.5 Funciones Load / Save

Estos métodos son los responsables de realizar las copias de seguridad o restaurarlas y enviarle o recibir dichas copias del cliente. El acceso a estos dos métodos se produce por dos medios distintos.

Protocolo RMI - El envío de archivos por RMI es trivial y sencillo de implementar.
Protocolo SOAP (SOAP With attachments) - SOAP no soporta directamente el envío de archivos sino que hay que hacer uso de SOAP with attachments.

Este acceso desde dos protocolos distintos resulta problemático pues resulta diferente la forma de, por ejemplo, enviar un archivo por RMI o mediante SOAP with attachments, en donde es necesario crear el mensaje, adjuntar el archivo y pasárselo a AXIS2 para que se encargue de su envío. HibernateBox proporciona métodos para manejar esta copias de seguridad mientras están dentro de la capa de persistencia. Además ofrece métodos de nombre save() y load() pensados para ser llamados por el cliente RMI que provocan el envío o recepción de los archivos de backup.

Por la naturaleza de SOAP with attachments es el propio servicio Web de RLinda el que debe ofrecer los métodos finales correspondientes a la recepción y el envío de archivos. Independientemente de que los métodos finales responsable del envío y la recepción de los archivos de backup estén implementados en el servicio Web (SOAP with attachments) o en la propia clase HibernateBox, siguen el mismo esquema.

Método Save

Realiza una copia de la información contenida en la base de datos, crear un archivo *JAR* y enviárselo al cliente.

- saveBackupFile() - guarda la información y crea el paquete *JAR*.
- save(String filename) - Llamado desde el lado del cliente RMI provoca la transmisión del fichero de backup al cliente.

La figura 43 muestra en qué consiste la operación *save*.

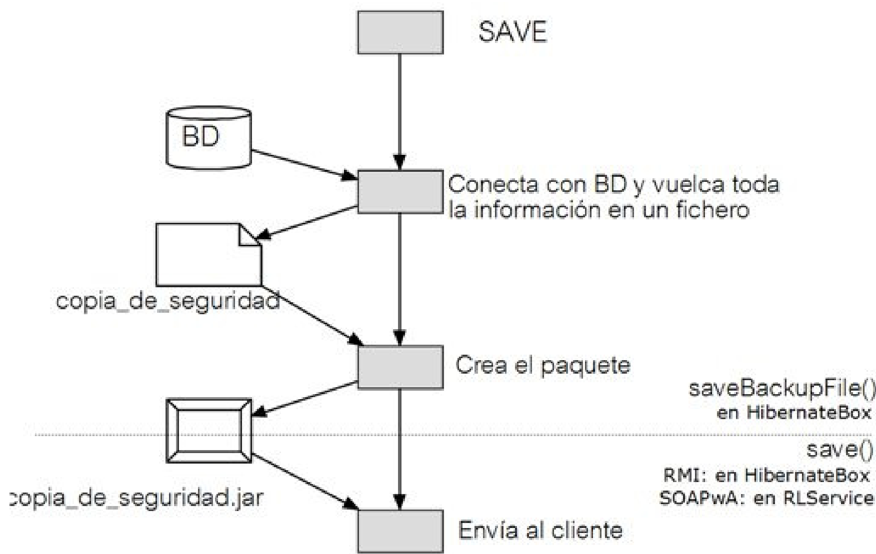


Fig. 41 Operación Save

Método Load

Restaura el estado de Rlinda a partir de una copia de seguridad suministrada. En el proceso intervienen los siguientes métodos de *HibernateBox*.

- load(byte[] filedata, String fileName) - Llamado desde el lado del cliente RMI provoca la transmisión del fichero de backup desde el cliente.
- loadBackupFile(String file) - descomprime el paquete *JAR* y restaura la base de datos.
- loadRlindaState() - inserta las tuplas que hay en la base de datos en el espacio de tuplas de la red RLinda.

La figura 44 muestra en qué consiste la operación *load*.

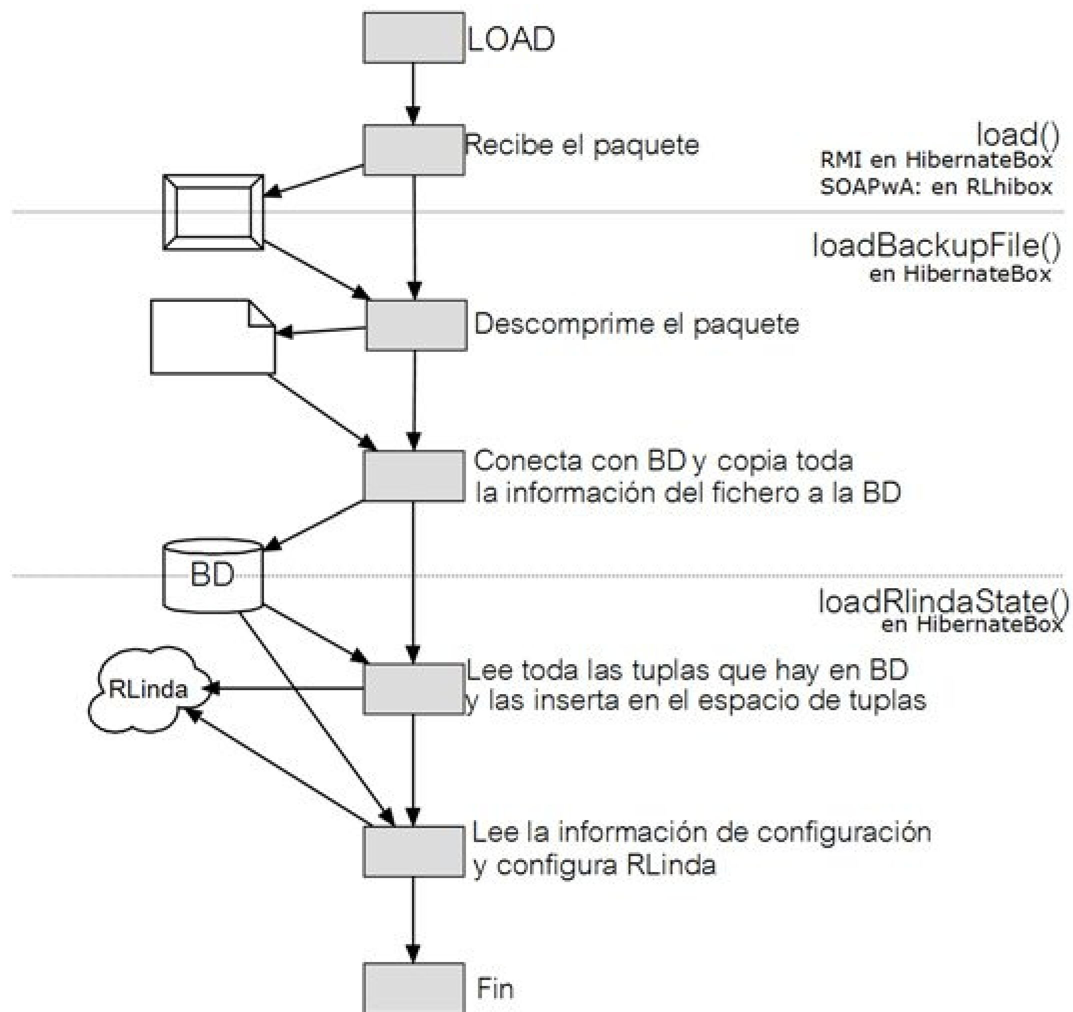


Fig. 42 Operación Load

5.2 Temporized RLinda

En el contexto de una comunicación entre procesos Web que se coordinan y se envían datos entre sí, es probable que cierta información solo sea válida durante un tiempo determinado o, en otro caso, que un proceso solo pueda permitirse esperarla durante un tiempo concreto. Pongamos como ejemplo un servicio diseñado para recibir información de un servicio de confianza pero que antes debe investigar durante un breve espacio de tiempo si existen otros servicios en el sistema más rápidos más convenientes. Con WS-PTRLinda como centro de la comunicación, sería estupendo poder lanzar una operación que mire si hay otros servicios disponibles pero que pasado un tiempo sin encontrar nada desbloquee al proceso para que pueda seguir su curso.

Otro caso práctico podemos establecerlo en un sistema de subastas tipo *Ebay*, donde un servicio Web pone a la venta un objeto durante un tiempo de dos días y medio. Sería perfecto que WS-PTRLinda desbloqueara al proceso pasado dicho tiempo si no hubiera llegado ninguna oferta.

Estos dos casos prácticos son formas diferentes de aplicar el concepto de temporización a RLinda. En esta sección se presenta una nueva capa de WS-PTRLinda que implementa este concepto de tiempo, ***Temporized RLinda***. Veremos cómo afecta a las operaciones ya existentes el parámetro de tiempo, como se implementa en el interior de RLinda (qué cambios han sido introducidos en la red) y el mecanismo encargado de desbloquear las operaciones de lectura que no han obtenido respuesta.

5.2.1 Funcionalidades añadidas

Estos son las nuevas funcionalidades del sistema.

- directivas básicas de comunicación de RLinda sobrecargadas para admitir un parámetro de tiempo*: out(tuple,timeout), in(pattern, timeout), rd(pattern,timeout).*

- Soporte para el parámetro de tiempo en las tuplas (clase Tuple de Renew).

Estas dos nuevas funcionalidades nos harán hablar en este capítulo de varios conceptos clave:

- **valor de timeout (o timeout) -** tiempo de validez de una operación o una tupla desde el momento que entra en el sistema.
- **tiempo/fecha de expiración -** punto en el tiempo a partir del cual la operación o la tupla ya no es válida. Por ejemplo, si en t=15 entra una tupla con valor de timeout = 10, la fecha de expiración es t=25.
- **Tuplas/datos temporizadas -** Las tuplas son los datos empleados en RLinda, añadirles tiempo supone que solo serán validas en el sistema durante dicho periodo. Una vez pasado son descartadas y nunca será devuelta.
- **Operaciones temporizadas -** La operación en sí es válida durante un tiempo determinado.
- **Cabecera de timeouts -** Una tupla está temporizada si tiene una cabecera con unas características concretas: [“TIMEOUT”,valor] dónde valor indica el valor del timeout en segundos.

Estas dos formas de temporización permiten combinaciones de operaciones temporizadas con datos temporizados, todo esto se explica en detalle más adelante.

5.2.2 Organización de TRLinda como servicio Web

Temporized RLinda se implementa como un servicio Web único que ofrece tanto operaciones temporizadas como sin temporizar. AXIS2 no soporta la sobrecarga de métodos de manera que las operaciones habituales se duplican para ofrecer la posibilidad de parámetros duplicados.

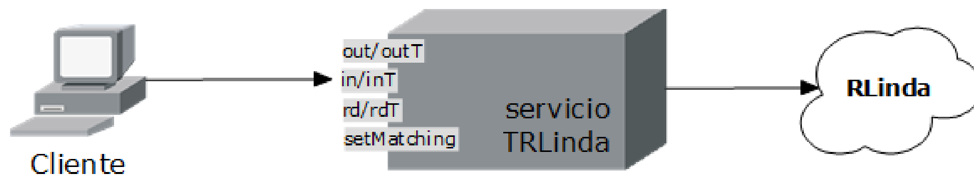


Fig. 43 Visión del servicio Web desplegado por Temporized RLinda

5.2.3 Arquitectura del sistema

Las nuevas funcionalidades de la capa de Temporización de WS-PTRLinda se basan en varios elementos:

- **Clase TemporizationBox -** sobrecarga las operaciones básicas de RLinda con sus versiones temporizadas.
- **Red de TRLinda -** nuevas transiciones para disparar las operaciones temporizadas.
- **Coordinador -** Modificado para ofrecer acceso desde código java a las nuevas transiciones.
- **Clases Sleeper y SleepWalker -** responsables de desbloquear las operaciones temporizadas una vez vencido el timeout.
- **Clase Tuple -** Se añaden los parámetros relativos a la temporización y se modifica la función de *matching* para tener en cuenta las tuplas temporizadas.

Todos estos elementos se explican a continuación seguidos para finalizar de unas trazas de ejecución que explican diferentes situaciones que pueden ocurrir.

TemporizationBox

La clase TemporizationBox es la clase que representa la capa de temporización de RLinda. A diferencia de la capa de persistencia, esta es una clase muy sencilla que simplemente sobrecarga las operaciones básicas de RLinda con sus versiones temporizadas.

- **out(Tuple tuple,long timeout) -** escribe una tupla en el espacio de tuplas, si *tuple* no tiene cabecera de temporización se le añade con valor *timeout*.
- **rd(Tuple pattern,long timeout) -** Si existe una tupla en el espacio de tuplas que se corresponde con el patrón *pattern* es devuelta. Si no se bloquea hasta que pasen *timeout* segundos.

- **in(Tuple pattern, long timeout) -** Si existe una tupla en el espacio de tuplas que se corresponde con el patrón *pattern* es devuelta. Si no se bloquea hasta que pasen *timeout* segundos. La tupla es borrada del espacio de tuplas.

- **setMatchingtype(int type) -** cambia al tipo de matching.

Todas las operaciones de TRlinda están sobrecargadas para aceptar tuplas en formato XML como parámetro de entrada.

Esta clase sirve como capa de presentación de todos los métodos, implementa la interfaz TemporizationBoxRemote que permite accederla por RMI. Las funcionalidades de esta capa se abasan principalmente en las dos clases que se describen a continuación y en las modificaciones en la red.

SleepWalker y Sleeper

Una operación de lectura se bloquea debido a que una transición no se puede disparar, de manera que no es trivial desbloquearla. Para ello se necesita una clase que funcione en *background* que lleve la cuenta del tiempo que quedad a cada operación antes de que expire. Toda operación de lectura se almacena en un lugar de peticiones pendientes, uno para *in* (*pendingTakeQueries*) y otro para *rd* (*pendingReadQueries*), y permanece ahí hasta que la función de *matching* encuentra una tupla que se ajuste al patrón dado. Cada vez que una operación expira esta clase debe llamar a una transición que retire la tupla del espacio de pendientes. Este trabajo es responsabilidad de las clases SleepWalker y Sleeper.

Ambas clases comparten una lista de datos pendientes y otra de operaciones pendientes. Todas las operaciones de lectura poseen un *id*, este *id* junto con el valor del timeout se almacenan en la lista de manera que el timeout mas bajos ocupan las primeras posiciones. Sleeper lee siempre la primera posición de la lista y se bloquea durante el número de segundos correspondientes. Al despertar emplea este id para desbloquear la operación. La figura 46 muestra la relación entre ambos.

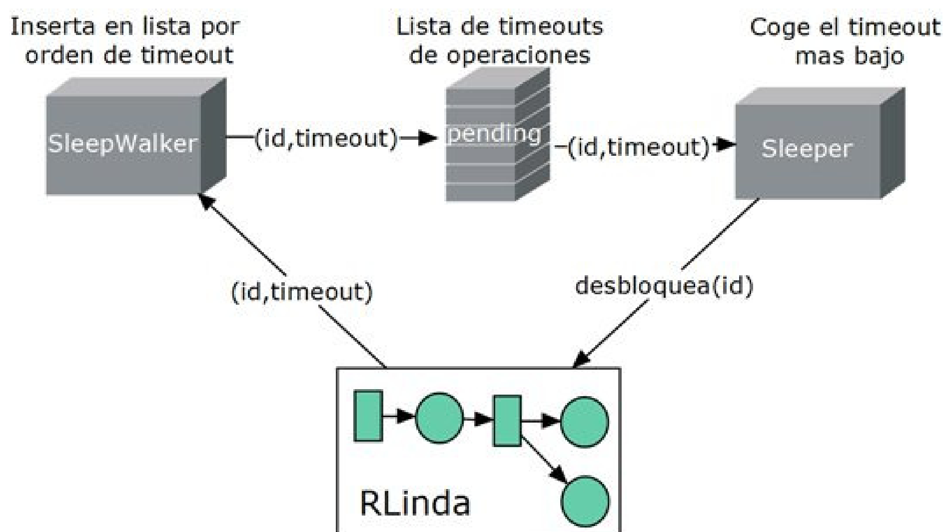


Fig. 44 Interacción de Sleeper y SleepWalker

Diagrama de clases de la capa de Temporización

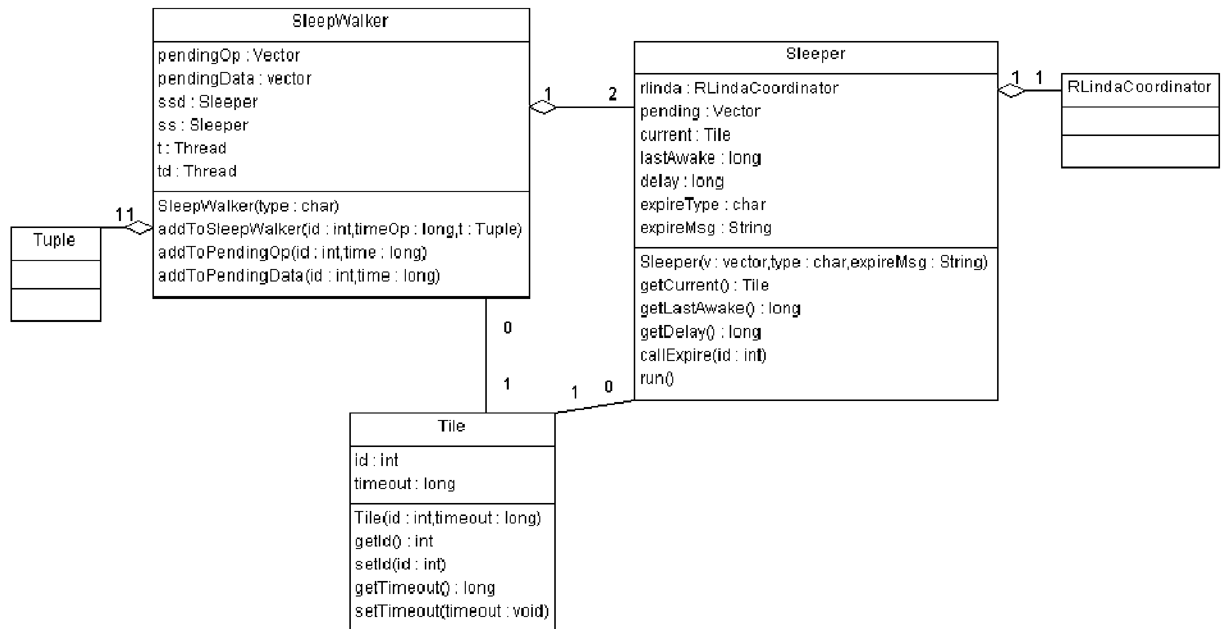


Fig. 45

SleepWalker

SleepWalker es creado y llamado desde la red de TRLinda cada vez que se produce una operación de lectura. Su misión es insertar la celda *id+timeout* en la lista de operaciones o datos pendientes. Esta lista está ordenada por orden de timeout, siendo el primer elemento de la lista el de timeout más bajo. Esta lista de pendientes es compartida con objeto Sleeper que se encarga de gestionar los timeouts. SleepWalker consta de los siguientes elementos.

- **Lista de operaciones pendientes** - Se almacena en cada celda el *id* de la operación y su valor de *timeout*.
- **Lista de datos pendientes** - Se almacena en cada celda el *id* de la operación a la que pertenece el dato y el valor de *timeout* del dato.
- **Objeto Sleeper de operaciones** - Objeto Sleeper pendiente de la lista de operaciones pendientes.
- **Objeto Sleeper de datos** - Objeto Sleeper pendiente de la lista de datos pendientes.

Además posee los siguientes métodos.

- **addToSleepWalker(int id,long timeOp,Tuple t)** - Inserta la operación en la lista de operaciones pendientes o inserta el dato en la lista de datos pendientes en función de cual tiene el valor de *timeout* más bajo.

- **addToPendingOp(int id,long time) -** Inserta *id* y *time* en la lista de operaciones pendientes.
- **addToPendingData(int id, long time) -** Inserta *id* y *time* en la lista de datos pendientes.

Clase Tile

Es una clase muy sencilla con dos campos, *id* de operación y *timeout* . Se emplea por comodidad para tener en un mismo objeto los datos de la operación relevantes en las listas de pendientes. A lo largo de los siguientes puntos aparecerá el concepto de *Tile* para referirnos al conjunto'*id* + *timeout*'.

addToSleepWalker

Una operación temporizada puede además tener un dato temporizado. En estos casos hay que comprobar cuál de los dos valores de *timeout* s menor para decidir si insertar un *Tile* (*id* + *timeout*) en la lista de operaciones pendientes o de datos pendientes. El método *addToSleepWalker* introduce una operación de lectura en la clase y es esta la que se encarga de decidir si tiene prioridad el timeout del dato o de la operación. Un diagrama de decisión muestra en la figura 50.

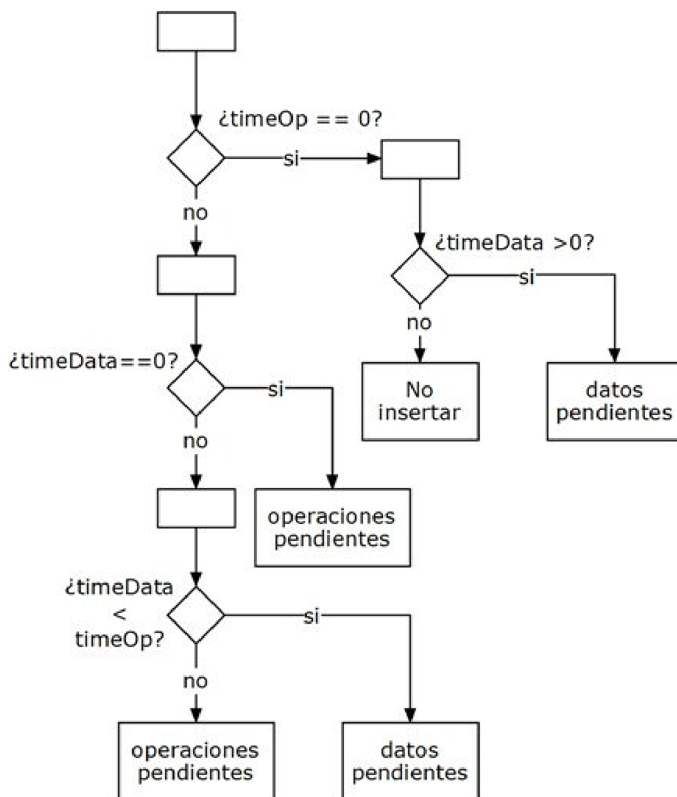


Fig. 46

addToPendingData y AddToPendingOp

Insertan el *Tile* en la posición de la lista de pendientes que corresponda acorde a varios criterios:
El valor de timeout (el *timeout* más pequeño el primero)
El tiempo que lleva consumido el primer elemento de la lista

Cómo se insertan operaciones en la lista de operaciones pendientes

Cuando un nuevo Tile debe ser incluido en la lista, el Tile que está en primera posición ya lleva un tiempo siendo gestionado por el Sleeper. Este tiempo debe de ser tenido en cuenta a la hora de decidir donde insertar el nuevo Tile. La figura muestra una simple traza de ejecución.

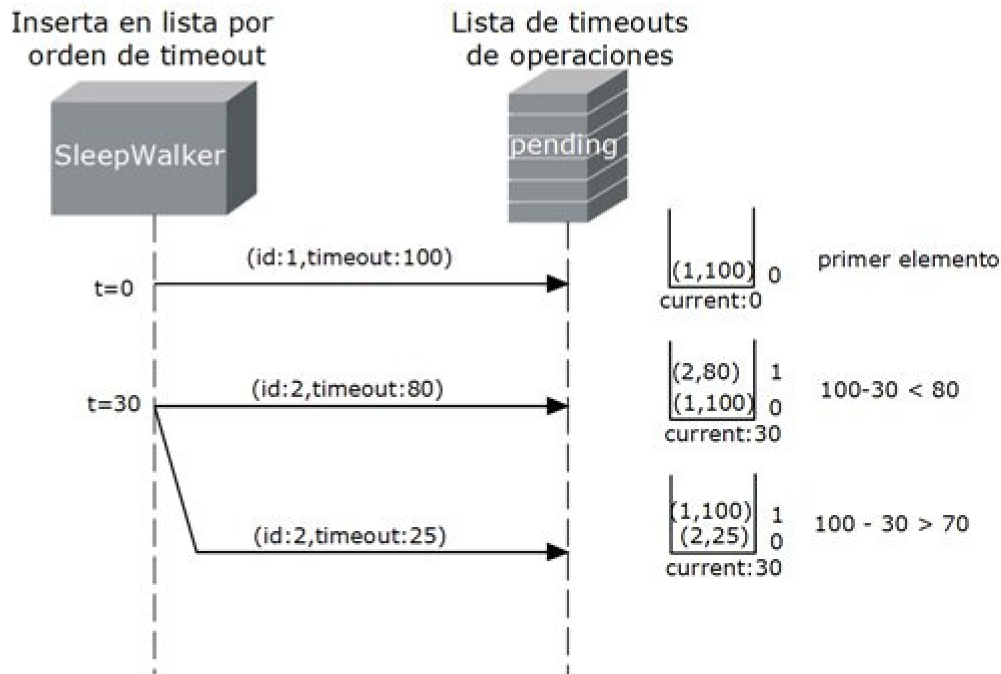


Fig. 47 Como se insertan elementos en la lista

La traza de ejecución muestra la llegada de dos posibles timeouts en segundo lugar. En el primer caso (*timeout:80*) el *timeout* es mayor así que se añade en la posición dos. El otro caso (*timeout:25*) tiene mayor prioridad (expira antes que el que está actualmente en primer lugar) de manera que debe insertarse el primero de la lista.

Sleeper

Un objeto Sleeper se ejecuta en un *thread* distinto y comparte una lista de pendientes son una instancia del objeto SleepWalker. Tiene como función gestionar el *timeout* del primer elemento de su lista de pendientes. Posee una instancia del coordinador de TRLinda para, una vez transcurra el tiempo de *timeout* del primer elemento de la lista, desbloquear la operación de lectura correspondiente. Los elementos de la clase Sleeper son:

- **rlinda -** instancia del coordinador de TRLinda.
- **pending -** Lista de elementos pendientes.
- **current -** elemento que está siendo gestionado actualmente.
- **lastAwake** - último instante de tiempo (tiempo del sistema en milisegundos) en que Sleeper no estaba bloqueado.
- **delay -** tiempo en milisegundos desde que la lista de pendientes no está vacía.
- **expireType -** indica si el Sleeper está trabajando con operaciones de *read* o *take*
- **expireMsg -** el mensaje que debe mostrar al desbloquear una operación.

Implementa además los siguientes métodos:

- **callExpire(int id) -** desbloquea la operación de lectura con identificador *id* a través de el coordinador de RLinda.
- **run() -** gestión de los timeouts.
- métodos get de current,lastAwake y delay.

Sleeper implementa la interfaz *Runnable* le permite ser ejecutado en un thread aparte. Toda la gestión del *timeout* se produce en él método *run()*, que se describe la figura 25. El mecanismo se basa en que *Sleeper* lee el primer elemento de la lista que es el de menor *timeout*, realiza una llamada a *wait(timeout)* que lo dejará bloqueado durante *timeout* segundos. Al despertar, accede a TRLinda para desbloquear la operación correspondiente.

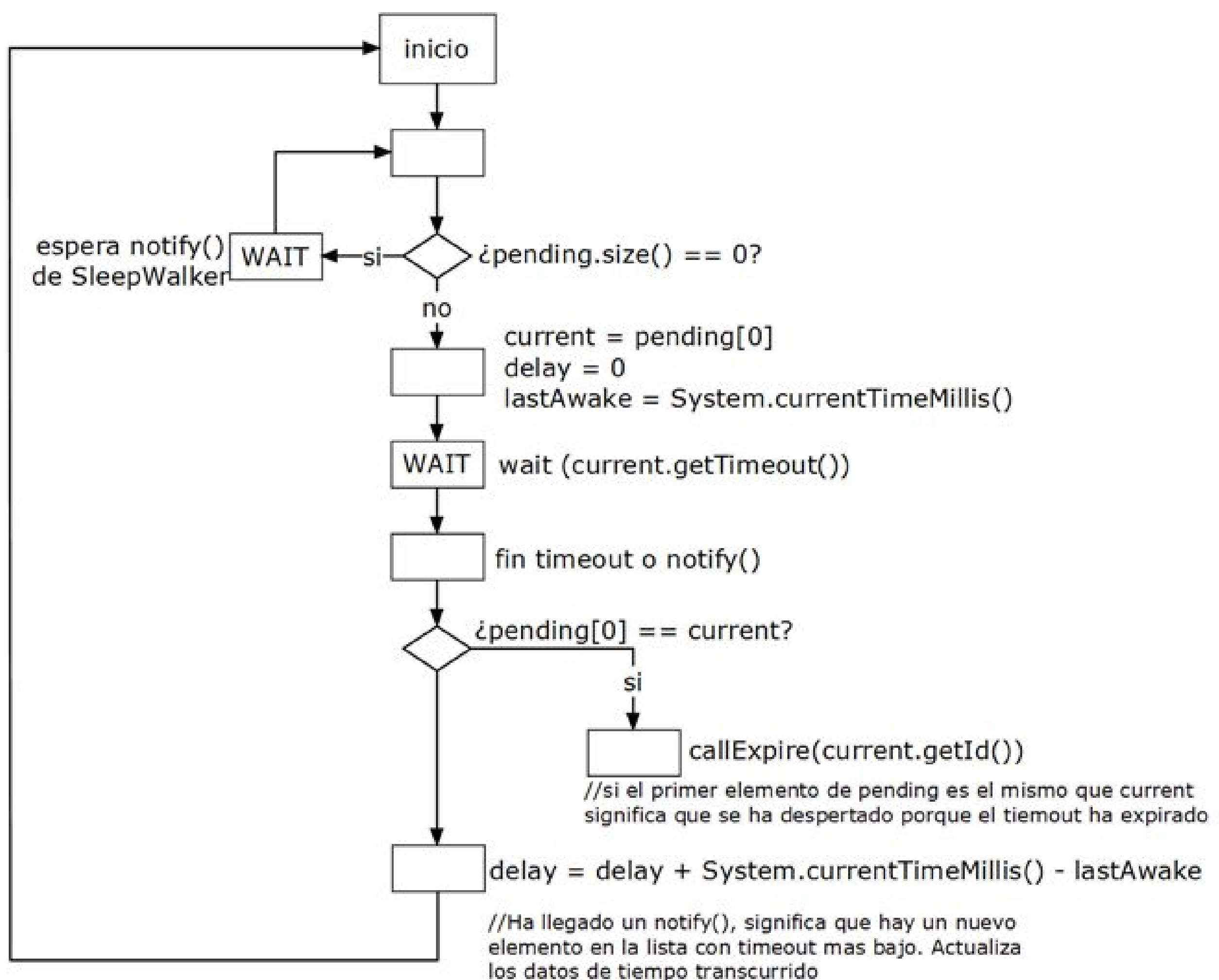


Fig. 48 Gestión de timeouts por parte de Sleeper

SleepWalker y Sleeper se comunican entre ellos por medio de la lista de pendientes y se sincronizan para evitar bloqueos y problemas de acceso simultáneo a la misma mediante llamadas *wait()* y *notify()* por parte de Sleeper y SleepWalker respectivamente.

El coordinador y la red de TRLinda

La red de RLinda debe de ser modificada para soportar los objetos anteriormente definidos y se deben añadir nuevas transiciones para poder desbloquear las operaciones cuyo timeout expira. Por su parte el coordinador debe ofrecer una forma de acceso a dichos cambios desde el exterior.

Instancia del objeto SleepWalker - Se crean dos instancias diferentes de SleepWalker, una que gestiona las listas de operaciones y datos pendientes para directivas read y otra para directivas take.

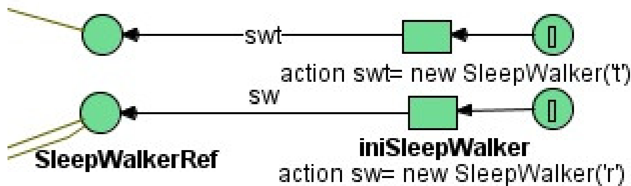


Fig. 49

Transiciones de inicio de operación - Toda operación de lectura se inicia en la red con una transición que escribe el patrón en el lugar pendingRead. Son necesarias nuevas transiciones iniciales que además se comuniquen con SleepWalker en caso de que la operación o los datos están temporizados. La nueva transición tiene el nombre de startReadTimed o startTakeTimed.

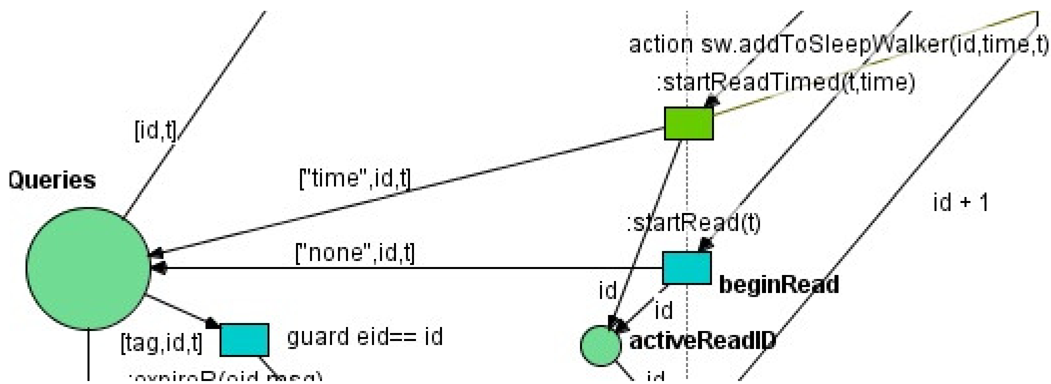


Fig. 50

Sistema de desbloqueo de operaciones - una transición de nombre expireR() (y su equivalente expireT()) se encarga de consumir el patrón del lugar pendingReads y depositar en el lugarmatched una tupla con el mensaje [“data timeout”] o [“operation Timeout”].

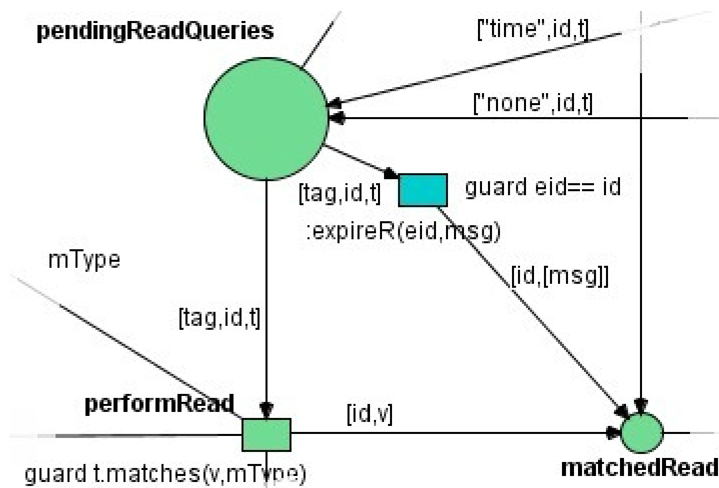


Fig. 51 Mecanismo de desbloqueo de operaciones

Una operación de lectura bloqueada permanece en el lugar *pendingReadQueries* (para las operaciones *rd*). Cuando su timeout expira (Sleeper está gestionándolo), Sleeper dispara, desde el exterior, la transición *expireR(eid,msg)* donde eid es el id de la operación a desbloquear (que no deja de ser una tupla en el lugar *pendingReadQueries*). Solo se extrae la tupla con dicho id debido a la condición de guarda *guard id == eid*.
Al dispararse, la transición *expireR* deposita una tupla en el lugar *matchedRea*d cómo si fuera un resultado normal de la operación de lectura.
La operación se desbloqueará (pues ha encontrado una tupla que se ajuste a su patrón, o eso piensa ella).

La tupla devuelta está contenida en *msg* y será [“operation timeout”] o [“data timeout”] según hayan expirado los datos o la operación.

Ejemplo de operación de lectura

Supondremos que se ha ejecutado una operación **rd(template,200)** y veremos como se van activando las transicones en la red de RLinda. Puede observarse la red en la figura 54.

Se dispara *startReadTimed(t,time)* pruciendo la escritura de *template* en *pendingReadQueries* y añadiendo en SleepWalker la operación temporizada.
Suponemos que no existe ninguna tupla en el espacio de tuplas que concuerde con *template* de manera que este permanecerá en *pendingReadQueries* hasta que finalice el *timeout* o entre en el sistema una tupla que corresponda.
El *timeout* expira y la transición *expireR(eid,msg)* es disparada retirando de *pendingReadQueries* la tupla con *id* igual a *eid*, que es la nuestra.
Al dispararse la transición se provoca la escritura de una tupla con mensaje *msg* (que es “operation timeout”) en el lugar *matchedRead.*
De aquí en adelante todo continúa como una operación *read* normal.

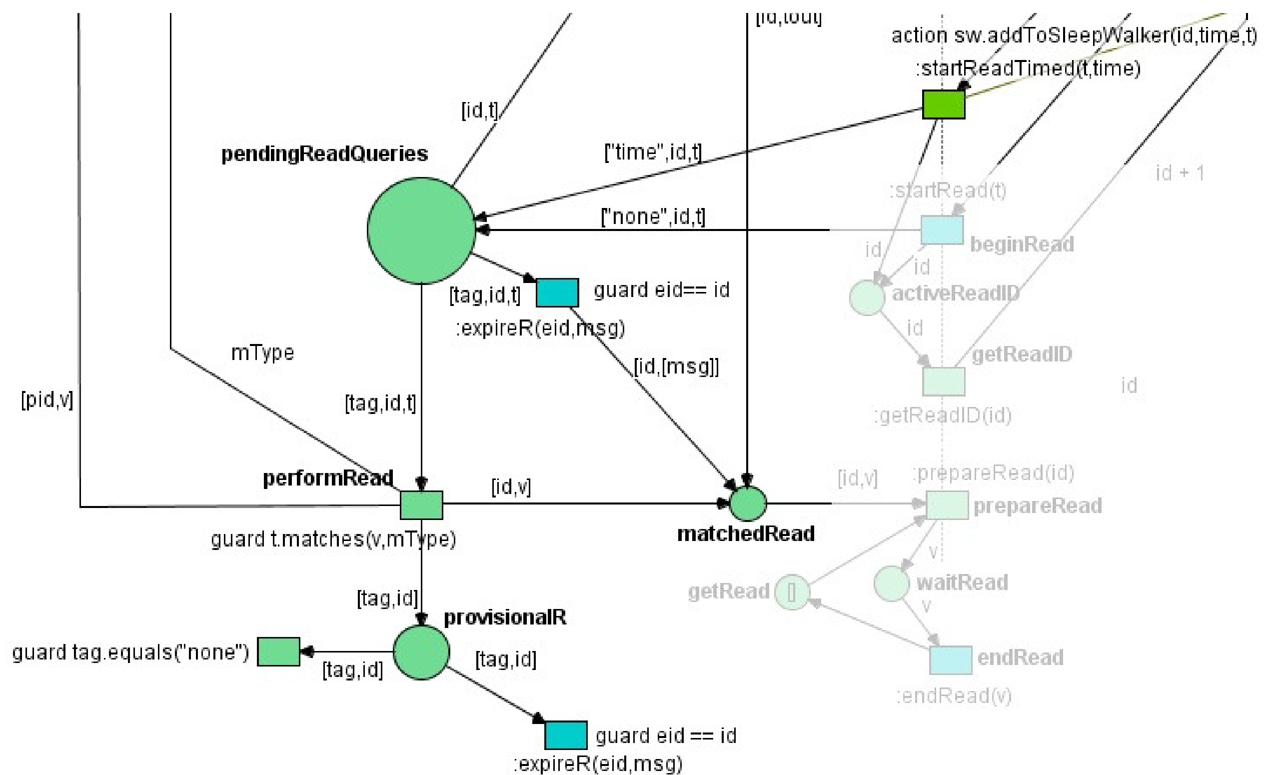


Fig. 52

5.2.4 Aplicando la temporización a: las tuplas

Una tupla está temporizada si posee la cabecera ***[“TIMEOUT”,valor]*** donde *“TIMEOUT”* es una palabra clave que emplea la red de RLinda para destectar que una tupla tiene timeout y *valor* es el tiempo en segundos que esa tupla permanecerá en el sistema antes de que expire.

Temporizar una tupla es muy sencillo, solo hay que añadirle la cabecera al inicio:

- [“tupla original”] pasa a estar temporizada así: [[“TIMEOUT”,15],[“tupla original”]] .
- [[“juan”,10],[“pedro”,11]] pasa a estar temporizada así:
  [[“TIMEOUT”,6000], [[“juan”,10],[“pedro”,11]]]

Una tupla temporizada con valor de timeout de cero se comporta como si no estuviera temporizada.

Las tuplas temporizadas se diferencian de las tuplas normales en que en la función de *matching* se comprueba si han expirado ya. Si esto sucede, se marcan como que han expirado para que la red de TRLinda las descarte como se muestra en la figura 27.

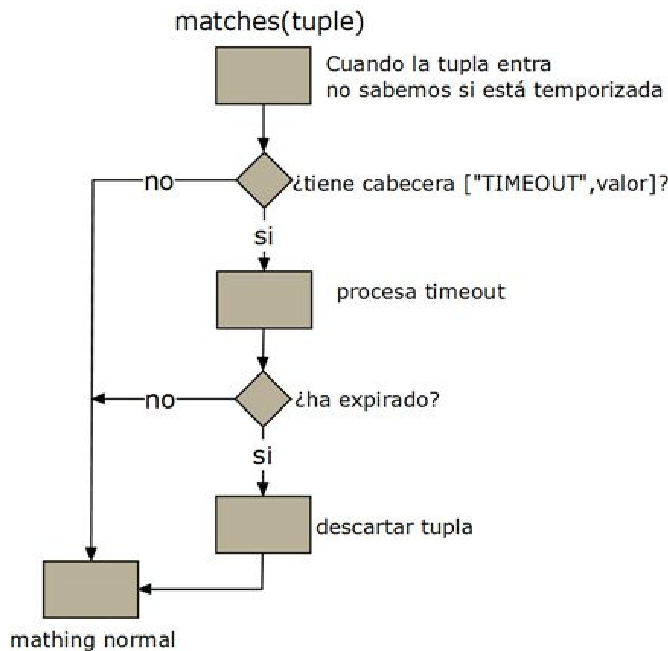


Fig. 53 Matching temporizado

Patrones temporizados

Los patrones no dejan de ser tuplas (clase Tuple) pero empleadas en las operaciones de lectura para buscar tuplas del espacio de tuplas que se les ajusten. Un patrón se temporiza de la misma forma explicada unas líneas atrás, añadiéndole una cabecera de timeout.

Los patrones temporizados sin embargo se tratan de una forma totalmente diferente en RLinda ya que una vez expiran deben desbloquear al proceso que lanzó la operación de lectura. Como sucede esto se explica un poco más adelante, por ahora nos quedaremos con que si se lanza una operación de lectura y el patrón expira y el proceso se desbloquea recibiendo como resultado una tupla indicadora de lo que ha sucedido: *["data timeout"]*.

5.2.5 Aplicando temporización a: las operaciones

Aplicar el concepto de tiempo a una operación supone que dicha operación solo será valida durante un tiempo limitado. Por definición, esto afecta de forma diferente a escritura y a lectura.

En RLinda el método *out(tuple)* provoca que la tupla *tuple* se escriba en el espacio de tuplas. No es bloqueante, por lo que añadir tiempo a la operación carece de sentido. Luego la operación de escritura temporizada es equivalente a una operación *out(tuple)* normal cuya tupla está temporizada.

out(tuple,timeout) - temporiza la tupla *tuple* y la escribe en el espacio de tuplas de TRLinda.

Las operaciones de lectura de TRLinda son por definición bloqueantes en caso de que no se encuentre una tupla en el espacio de tuplas que se corresponda con el patrón dado. Al aplicar temporización a estas operaciones estamos diciendo que si no se encuentra ninguna tupla que corresponda con el patrón, pasado un tiempo *t* la llamada se desbloquee.
in(pattern, timeout) y *rd(pattern, timeout)* son operaciones temporizadas que se desbloquean pasados *timeout* segundos.

Una operación expirada devuelve como resultado al proceso que la ejecutó (y que permanecía bloqueado esperando un resultado) una tupla indicativa de lo sucedido: ["operation timeout"].

Operaciones temporizadas con datos temporizados
En el caso de operaciones de lectura existe la posibilidad de temporizar tanto operaciones como datos. Un dato representa información y tiene más importancia en el sistema que una simple operación, de forma que se considera preferente su timeout. El resultado obtenido al desbloquearse una operación se rige según lo siguiente:

- **T_datos > T_operación** - devuelve [“Operation timeout”].

- **T_datos < T_operación** - No tiene sentido una operación que caduca antes que los datos, en este caso se toma T_datos = T_operación y devuelve [“Data timeout”].

- **T_datos == T_operación** - devuelve [“Data timeout”].

5.2.6 Traza de interacción SleepWalker – Sleeper

En la figura 28 se observan las interacciones, a través de la lista de pendientes, entre SleepWalker y Sleeper. Este primero inserta en orden los timeout en pendientes y notifica a Sleeper si ha insertado un *timeout* menor del tiempo que queda del que está gestionando actualmente.

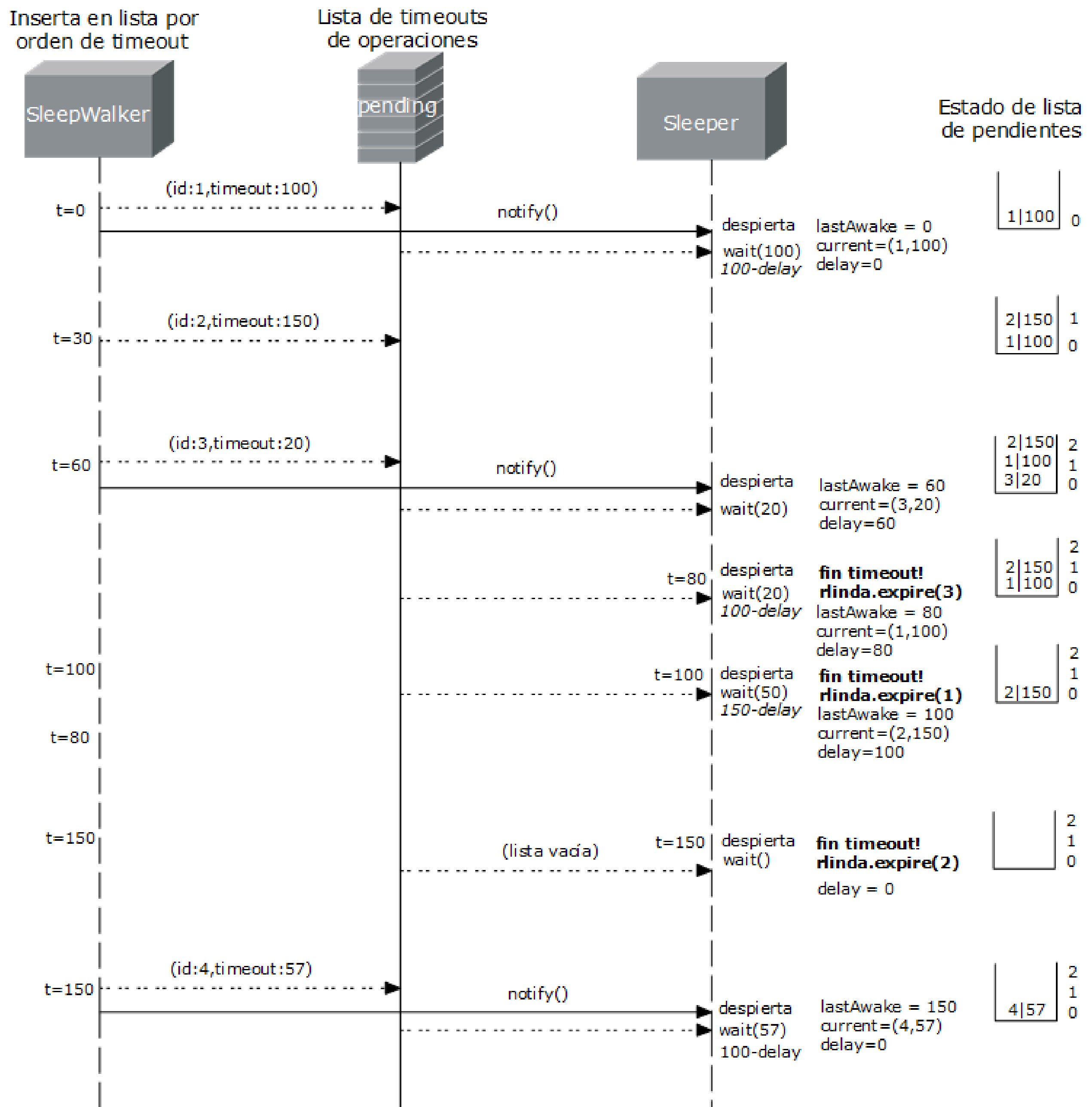


Fig. 54 Gestion de tiemouts de Sleeper y SleepWalker

La clase Sleeper gestiona el *timeout* del primer elemento de la lista de pendientes, que tiene el timeout más pequeño, mediante de *wait()* de timeout segundos. Cuando se despierta comprueba si lo ha hecho porque el timeout ha pasado o si ha sido despertado por SleepWalker. En caso primero Sleeper retirará el primer elemento de la lista, accederá a TRLinda para desbloquear la operación y empezará de nuevo con el siguiente elemento de la lista.

En el segundo caso significa que ha llegado un *timeout* con mayor prioridad (más pequeño y que ya ha sido puesto en primer lugar de la lista) . Sleeper se bloqueará otra vez con *wait()* pero con el valor de este nuevo *timeout*.

Un caso problemático

Mientras Sleeper se despierta, accede a la lista de pendientes y lanza una nueva llamada a *wait()* con el valor de *timeout* que corresponda pasan unos pocos milisegundos. No hay manera de tener estos pocos milisegundos y puede producirse la situación en que los tiempos de los diferentes *timeouts* estén tan ajustados que esa pérdida de segundos suponga que un *timeout* ya ha expirado cuando entra en el Sleeper y lanza *wait()*.

En este caso lo que sucede es que *wait(tiempo)* se está llamando con un valor de *tiempo* negativo y genera una excepción. La solución es simple, si ocurre dicha excepción se captura y se emplea la referencia a RLinda para desbloquear la transición consecuente. La ejecución podrá continuar normalmente.

5.3 WS- Persistent Temporized RLinda

Una vez presentadas las dos capas que componen este PFC, se agrupan todas bajo una capa de presentación que ofrece todas las operaciones disponibles desde el exterior. Esta clase tiene por nombre PTRLinda e implementa la interfaz PTRLindaRemote que permite instanciarla en RMI.

La capa de persistencia y la de temporización se han desarrollado de forma modular para que puedan usarse de forma aislada, pero la herramienta final se presenta como una combinación de ambas, una que aporta persistencia y otra los conceptos de temporización. Visto desde la perspectiva del cliente Web, así se ofrece el sistema WS-PTRLinda.

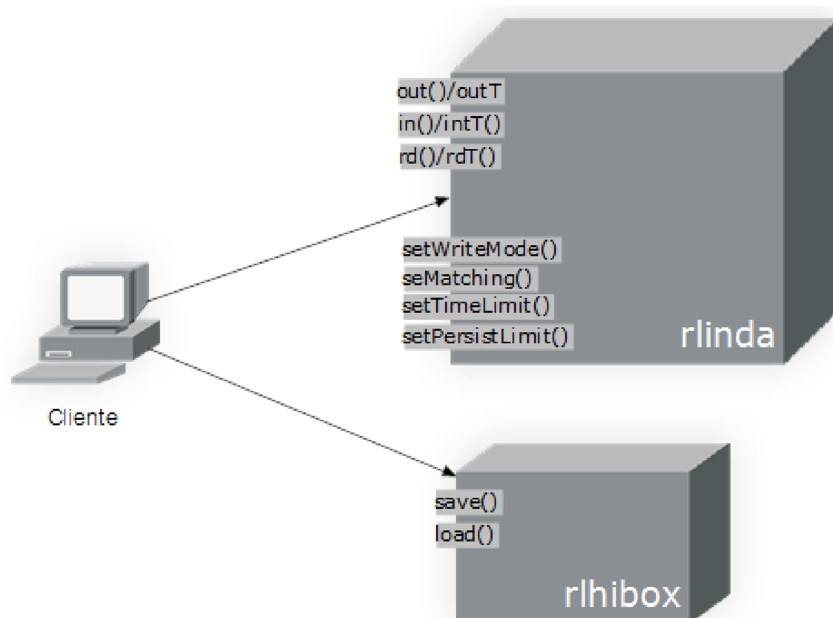


Fig. 55 Visión de los servicios Web desplegados por WS-PTRLinda

El servicio principal tiene ahora una referencia a la capa de temporización y ofrece las nuevas operaciones temporizadas. AXIS2 no permite sobrecarga de manera que se ofrecen con nombres distintos (por ejemplo *out()* y *outT()*). El servicio encargado de proporcionar las operaciones de copia de seguridad permanece intacto y sigue poseyendo una referencia a la capa de persistencia.

ANEXO 7 | **Glosario de términos**

agente (programación) - Objeto desplegado en un entorno que se comunica con este y otros agentes para lograr los objetivos que le han sido asignados. Referente a la programación orientada a Agentes y programación distribuida.

Apache Tomcat - servidor de aplicaciones sobre el cual se despliegan los servicios Web de PTRLinda.

arco (redes de petri) - línea que comunica un lugar con una transición o al revés.

AXIS2 - Framework para la creación, ejecución y despliegue de servicios Web

clave primaria - identificador único de un registro en una base de datos.

Criterio de matching - características de la función de matching a la ahora comprobar si una una tupla se ajusta o no a un patrón.

C3pO - pool de conexiones empleado por Hibernate.

C++ - Lenguaje de programación que extiende C con mecanismos para la manipulación de objetos.

C# - lenguaje de programación orientado a objetos desarrollado y estandarizado por Microsoft como parte de su plataforma .NET.

Hibernate - Herramienta de Mapeo objeto-relacional para la plataforma Java que facilita el mapeo de atributos entre una base de datos relacional y el modelo de objetos.

IBM - International Business Machines o IBM es una empresa multinacional que fabrica y comercializa herramientas, programas y servicios relacionados con la informática.

interfaz (java) - Colección de métodos abstractos y propiedades. En ellas se especifica qué se debe hacer pero no su implementación siendo las clases que implementen estas interfaces las que describan la lógica del comportamiento de los métodos.

JDBC - Java Database Connectivity. Es una API que permite la ejecución de operaciones sobre bases de datos desde el lenguaje de programación Java, independientemente del sistema operativo donde se ejecute o de la base de datos a la cual se accede, utilizando el dialecto SQL del modelo de base de datos que se utilice.

Lisp - lenguaje de programación basado en listas de elementos.

lugar (redes de Petri) - nodo circular donde un arco puede depositar o retirar tuplas.
matching - función que compara una tupla con un patrón y devuelve verdadero si dicha tupla se ajusta a dicho patrón o falso en caso contrario.

MySQL - Sistema de gestión de bases de datos relacional, multihilo y multiusuario.

patrón - una tupla que puede contener el elemento ”?” que actúa como comodín a la hora de comprobar si una tupla se ajusta a un patrón en la función de matching.

persistencia - la acción de preservar la información de un objeto de forma permanente (guardar), pero a su vez también se refiere a poder recuperar la información del mismo (leer) para que pueda ser nuevamente utilizada.

pid - identificador de persistencia, empleado en la capa de persistencia de WS-PTRLinda para identificar cada tupla, tanto dentro del espacio de tuplas como en la base de datos.

plug & play - tecnología que permite a un dispositivo informático ser conectado a un ordenador sin tener que configurar ni proporcionar parámetros a sus controladores.

P2P - Peer to peer. Sistema de intercambio de archivos donde no existe una entidad central que controle dichos archivos.

stub (Java) - Componente en la comunicación RMi. Objeto que encapsula a un Objeto remoto, tiene una identificación del dicho objeto remoto a utilizar y su interfaz de manera que posibilita instanciar un objeto mediante RMI.

SQLite - Sistema de gestión de bases de datos relacional, contenida en una relativamente pequeña (~275 kiB)1 biblioteca en C.

stub (Renew) - Objeto que posibilita el acceso a las transiciones de las redes de Petri modeladas en Renew desde código Java.

REST - (Representational State Transfer) o REST es una técnica de arquitectura software para sistemas hipermedia distribuidos como la World Wide Web.

RMI - (Java Remote Method Invocation) es un mecanismo ofrecido por Java para invocar un método de manera remota. Forma parte del entorno estándar de ejecución de Java y provee de un mecanismo simple para la comunicación de servidores en aplicaciones distribuidas basadas exclusivamente en Java.

Shell - interfaz usada para interactuar con el núcleo de un sistema operativo.

SOA - Arquitectura Orientada a Servicios (en inglés Service Oriented Architecture), es un concepto de arquitectura de software que define la utilización de servicios para dar soporte a los requisitos del negocio. Permite la creación de sistemas altamente escalables que reflejan el negocio de la organización, a su vez brinda una forma bien definida de exposición e invocación de servicios (comúnmente pero no exclusivamente servicios Web), lo cual facilita la interacción entre diferentes sistemas propios o de terceros.

SOAP - (siglas de Simple Object Access Protocol) es un protocolo estándar que define cómo dos objetos en diferentes procesos pueden comunicarse por medio de intercambio de datos XML.

SQLite - Sistema gestor de bases de datos muy ligero y basado en ficheros.

thread - hilo de ejecución de un programa.

Token – en redes de Petri, entidad que se deposita en un lugar. Es tomado y depositado en otros lugares por una transición.

transición (redes de petri) - nodo rectangular. Comunica un lugar con otro lugar. Para dispararse debe existir una tupla en el lugar de origen, que será retirada y depositada en el lugar de llegada.

tupla - dato en RLinda. Puede estar formado por varios elementos, incluidas otras tuplas. Una tupla comienza por el caracter ”[” y acaba con el caracter ”]”.

Unix - sistema operativo portable, multitarea y multiusuario.

URL - Un localizador uniforme de recursos, más comúnmente denominado URL (sigla en inglés de uniform resource locator), es una secuencia de caracteres, de acuerdo a un formato modélico y estándar, que se usa para nombrar recursos en Internet para su localización o identificación, como por ejemplo documentos textuales, imágenes, videos, presentaciones digitales, etc.

workflow - El flujo de trabajo (workflow en inglés) es el estudio de los aspectos operacionales de una actividad de trabajo: cómo se estructuran las tareas, cómo se realizan, cuál es su orden correlativo, cómo se sincronizan, cómo fluye la información que soporta las tareas y cómo se le hace seguimiento al cumplimiento de las tareas.

W3C - consorcio internacional que produce recomendaciones para la World Wide Web.

WSDL - Web Services Description Language, un formato XML que se utiliza para describir servicios Web. Describe la interfaz pública a los servicios Web. Está basado en XML y describe la forma de comunicación, es decir, los requisitos del protocolo y los formatos de los mensajes necesarios para interactuar con los servicios listados en su catálogo. Las operaciones y mensajes que soporta se describen en abstracto y se ligan después al protocolo concreto de red y al formato del mensaje.

XML - siglas en inglés de eXtensible Markup Language (lenguaje de marcas extensible), es un metalenguaje extensible de etiquetas desarrollado por el World Wide Web Consortium (W3C).