Anexo I - Gestión del proyecto

En este anexo se detallarán los siguientes aspectos: la metodología utilizada para la realización del proyecto, las fases de las que consta el mismo, el esfuerzo realizada para la elaboración del proyecto, la supervisión llevada a cabo por el director y codirector del proyecto y las herramientas de gestión utilizadas.

I.1 - Metodología de desarrollo

Como metodología de trabajo se ha seguido un modelo de desarrollo basado en el ciclo de vida en cascada con retroalimentación. Como puede observarse en la figura 18, el ciclo de vida en cascada comienza en la fase de análisis seguida del diseño del sistema, la implementación y la verificación del mismo a través de una fase de pruebas. En cada una de estas fases se pueden detectar problemas o identificar mejoras que provocan una realimentación que afecta a las fases anteriores. Esta retroalimentación permite mejorar la calidad de las fases anteriores y así lograr una mejor calidad global en el proyecto desarrollado.

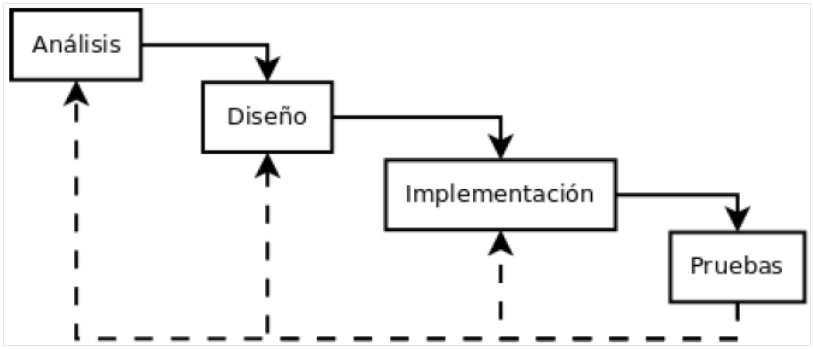

Figura 18: Ciclo de vida en cascado con retroalimentación.

I.2 - Fases del proyecto

En este apartado se indican las diferentes fases que engloban el desarrollo de este proyecto dando una breve descripción de las mismas y un listado de las tareas más relevantes que se llevaron a cabo durante las mismas.

1. *Puesta en Contexto*. En esta primera fase, se abordó el estudio del contexto en el que se ubica el proyecto. Para ello se buscó información y se leyeron una gran cantidad de artículos relacionados con los temas claves del mismo. Asimismo se estudiaron las tecnologías y herramientas que se iban a utilizar y se realizaron ejemplos para completar el aprendizaje de las mismas. El listado de conceptos que se estudiaron es el siguiente:

- *Workflows* científicos.
- Computación *Grid* y computación *Cloud*.
- Sistemas de gestión de *workflows* científicos en *Grid*.
- *Middlewares* de Grid: Condor y gLite.
- Redes de Petri, redes de referencia, paradigma redes-en-redes.
- Sistema de coordinación Linda. Implementación de WS-PTRLinda.

2. *Análisis*. En esta fase se llevó a cabo un análisis completo del estado del arte. Este análisis sirvió para identificar los requisitos que debía satisfacer el sistema, en base a las necesidades de los *workflows* científicos y los problemas identificados del análisis de otras soluciones. Las tareas fundamentales desarrolladas fueron:
   - Análisis de las necesidades de los *workflows* científicos.
   - Análisis de otras propuestas similares e identificación de los problemas y limitaciones de las mismas.
   - Definición de los requisitos del sistema a desarrollar.
3. *Diseño de la solución*. En esta fase se incluye la definición de la arquitectura global del sistema y de la arquitectura individual de cada uno de los componentes involucrados. Asimismo, también se incluye la decisión de las tecnologías a utilizar. El listado de tareas fundamentales realizado es:
   - Diseño de la arquitectura del sistema.
   - Diseño de la arquitectura de los componentes involucrados.
   - Decisión de las tecnologías a utilizar.
4. *Implementación*. Incluye todas las tareas relacionadas con la codificación de los diferentes componentes del sistema en el lenguaje de programación elegido. Los elementos desarrollados son los siguientes:
   - Modificación del *broke*r WS-PTRLinda. Esta modificación incluye la modificación de la API del Broker y la implementación de la política de emparejamiento competitivo.
   - Conjunto de mediadores: Condor y gLite.
   - Componente de movimiento de datos
   - Componente de manejo de fallos.
5. *Pruebas*. Esta fase corresponde a la elaboración, ejecución y análisis de las pruebas realizadas al sistema, tanto de forma individual a cada uno de sus componentes, como de forma global al sistema completo. Su objetivo es encontrar los errores en los que se ha incurrido en la implementación del sistema y dar solución a los mismos para garantizar un sistema libre de errores.
   - Pruebas unitarias de los diferentes componentes.
   - Pruebas de integración de todos los componentes del sistema.
   - Pruebas globales del sistema.
   - Definición, aplicación y testeo de los casos de prueba.
6. *Documentación*. Esta fase se ha llevado a cabo durante toda la extensión del proyecto. Consiste en la elaboración de todos los documentos, presentaciones y diagramas realizados durante el proyecto. Esta tarea se puede dividir en dos grandes grupos:
   - Documentación interna del trabajo realizado durante las diferentes fases del proyecto.
   - Elaboración de esta memoria.

I.3 - Gestión de tiempo y esfuerzo

En esta sección vamos a detallar el tiempo dedicado a cada una de las fases del proyecto y el esfuerzo dedicado a las mismas.

Gestión del tiempo

En la figura 19 se puede apreciar el diagrama de Gantt que representa la distribución de tiempo dedicada a cada una de las fases indicadas en el apartado anterior.

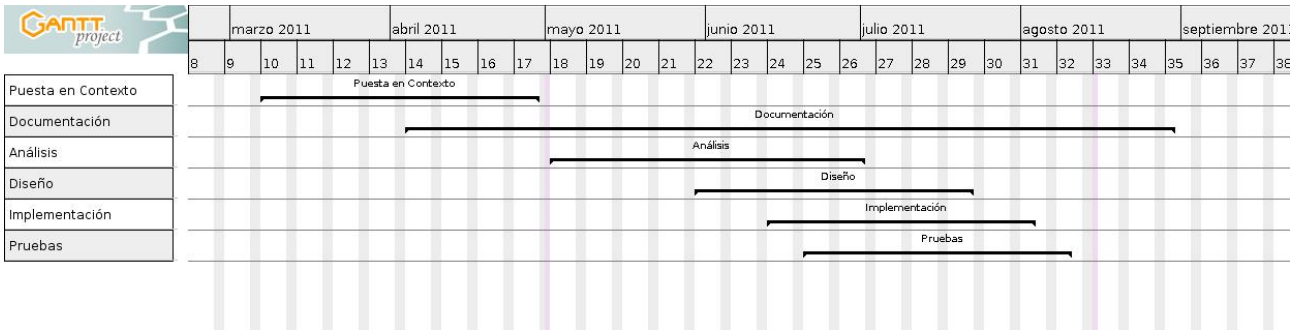

Figura 19: Diagrama de Gantt que muestra la distribución de todas las fases del proyecto.

El proyecto comenzó a principios de Marzo cuando, tras una reunión inicial, se comenzó la fase de puesta en contexto en la que el alumno comenzó las tareas de búsqueda de información y lectura de capítulos de libros y artículos con el objetivo de conocer el contexto en el que se enmarca el proyecto.

En la figura 20, se proporcionan más detalles de esta etapa junto con la etapa de documentación. Las mismas se presentan juntan porque son las únicas que no están sujetas al proceso de retroalimentación de la metodología utilizada.

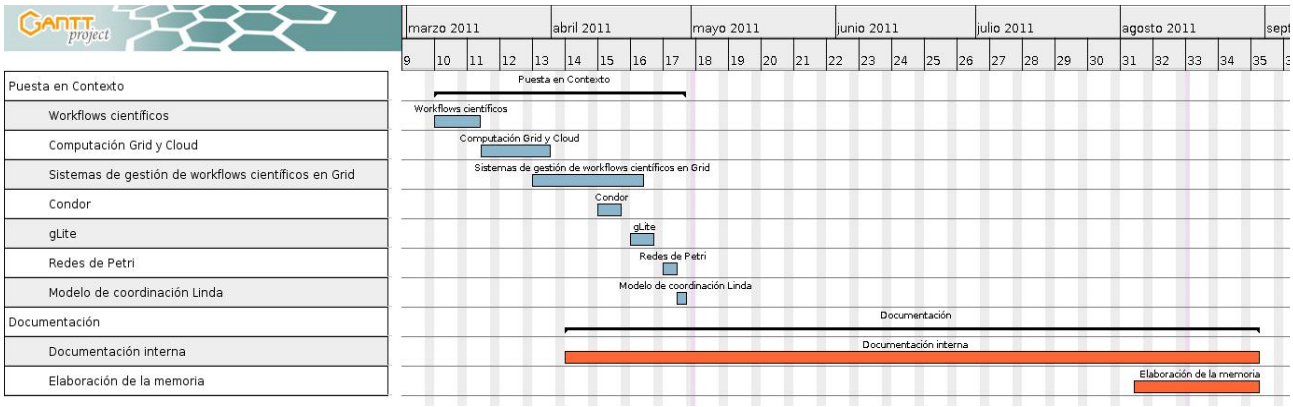

Figura 20: Diagrama de Gantt para las fases de puesta en contexto y documentación.

Estas dos etapas son imprescindibles en el desarrollo de cualquier proyecto. En primer lugar, es necesario conocer cuál es el contexto en el que se enmarca el proyecto para poder comprender el problema y abordar el mismo con garantías. En segundo lugar, la documentación ayuda a realizar un seguimiento del proyecto y a dejar constancia de la investigación realizada. En lo que respecta a la documentación hay que distinguir dos tareas fundamentales. En primer lugar, durante todo el desarrollo del proyecto se ha realizado documentación interna de las

tareas que se iban realizando y que ha sido aprovechada posteriormente para la elaboración de la memoria. La segunda tarea de esta fase es propiamente la elaboración de esta memoria, que consistió principalmente en adaptar la documentación interna que ya se había generado a lo largo del proyecto realizando pequeñas modificaciones.

Por su parte, en la figura 21 se muestra el diagrama de Gantt detallado para las fases de análisis, diseño, implementación y verificación. Estas fases si bien comienza en el orden nombrado anteriormente, están sujetas a modificaciones provocadas por otras fases, es decir, al proceso de retroalimentación de la metodología utilizada. De esta forma, como se puede observarse hay momentos del proyecto en los que coinciden todas estas fases.

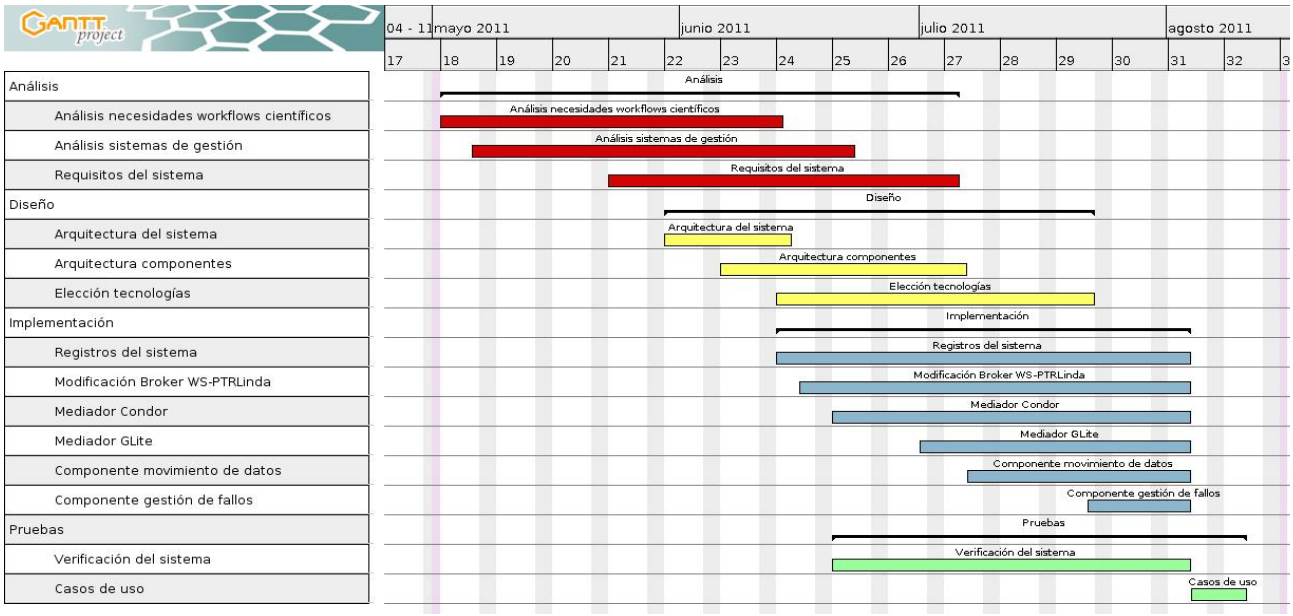

Figura 21: Diagrama de Gantt de las fases sujetas a retroalimentación.

La fase de análisis consistió en analizar más detalladamente los requisitos generales de un *workflow* científico, las características de los sistemas de gestión de *workflows* existentes y las limitaciones y problemas que presentaban los mismos. Una vez analizados estos aspectos, se definieron los requisitos del nuevo sistema en base a las necesidades y limitaciones detectadas.

Una vez que se definió cómo iba a ser el sistema a desarrollar, se pasó a la fase de diseño. Para ello se pasó a definir la arquitectura general del sistema, definiendo los aspectos fundamentales del mismo como los componentes involucrados, el mecanismo de interacción entre los mismos, etc. Posteriormente, se realizó el diseño de cada uno de los componentes del sistema, esta fase provocó la modificación de algunos aspectos iniciales de la arquitectura del sistema en base a los requisitos de los componentes. La última tarea de esta fase consistió en elegir las tecnologías a utilizar para los componentes del sistema, esta tarea se prolongó bastante tiempo ya que algunas tecnologías, como el motor de reglas, no se eligieron hasta unos días antes de comenzar la implementación del componente.

La fase de implementación comenzó a mitad del mes de Junio y finalizó la primera semana de Agosto. Comenzó con la creación de los registros del sistema para pasar posteriormente a la modificación del *broker* WS-PTRLinda. Posteriormente se abordó la creación de cada uno de los componentes que integran el sistema. La elaboración de diferentes

componentes provocó alguna modificación en otros por la aparición de errores. El componente más costoso de realizar fue el mediador de Condor al ser el primer mediador realizado. Además, éste se construyó en primer lugar sin utilizar URIs como datos de entrada y al incluir esta característica hubo que adaptar el mediador. Por contra, el mediador de gLite llevó poco tiempo ya que su implementación es muy similar al mediador de Condor.

Por su parte, la fase de pruebas se realiza de forma paralela a la fase de implementación, probando individualmente cada componente desarrollado y probando que la integración del mismo en el sistema se realizaba de forma correcta. Finalmente, cuando se determinó que no existían errores en el sistema y se finalizó la fase de implementación, se comenzó una fase en la que se diseñaron los casos de uso y se probaron los mismos, comprobando que el funcionamiento del sistema era correcto.

Finalmente, aunque no se incluye en el figura, durante todo el proyecto se ha realizado una gestión y un seguimiento del proyecto como se comentará de forma más detallada en la secciones 4 y 5 de este anexo.

Gestión del esfuerzo

En la figura 22 se puede apreciar el tiempo dedicado a cada una de las fases del proyecto en comparación con el resto de fases del mismo.

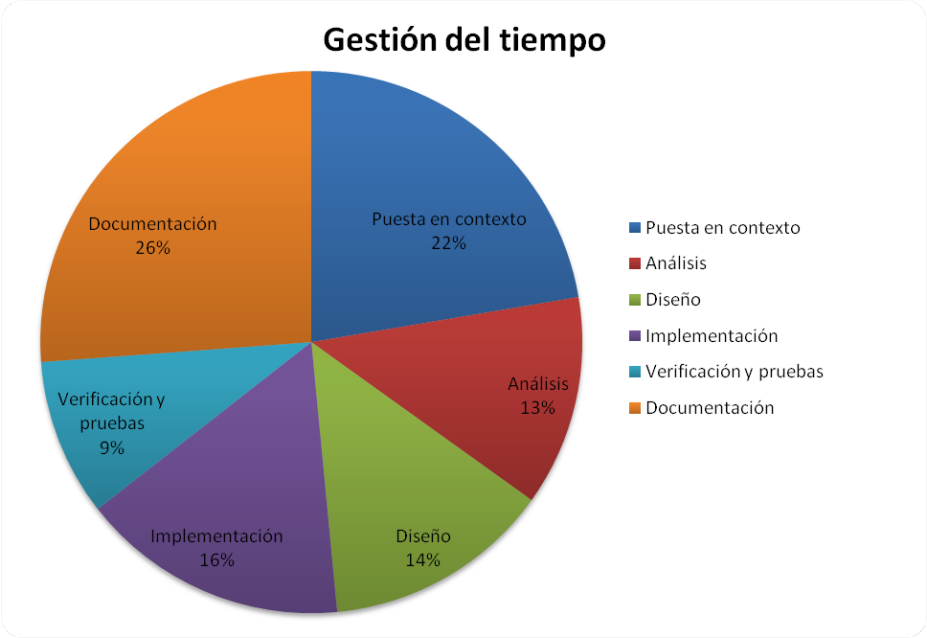

Figura 22: Diagrama de distribución del tiempo entre las diferentes fases del proyecto.

En total, la duración del proyecto asciende a 770 horas. El desglose de las mismas se puede observar a continuación:

- Puesta en contexto: 172 horas
- Análisis: 96,75 horas
- Diseño: 104,5 horas

- Implementación: 123 horas
- Verificación y pruebas: 72,25 horas
- Documentación: 201,5 horas

I.4 - Supervisión del proyecto

Para la realización de este proyecto se ha contado con la supervisión de un director y un codirector. En cualquier caso, el seguimiento del trabajo realizado ha sido constante por parte de ambos directores, así como por parte de otros miembros del grupo GIDHE [W1], grupo dentro del cual se ha desarrollado este proyecto.

Durante la duración del proyecto, se han realizado reuniones con los directores del mismo para tratar diversos aspectos. La periodicidad de las mismas ha sido variable, hay fases en las que ha sido suficiente con mantener una reunión por semana, y otras en las que se han realizado varias reuniones en una misma semana. En general, la estructura seguida durante las reuniones ha sido la siguiente:

1. Orden del día.
2. Evaluación del trabajo realizado desde la reunión anterior.
3. Discusión y debate de ideas, dudas e impresiones.
4. Planificación de nuevos objetivos.
5. Planificación de la siguiente reunión.

Además, se ha elaborado un acta en todas las reuniones mantenidas. Este acta incluye, además de los aspectos relevantes tratados durante la reunión, los asistentes a la misma, la duración de la reunión y el lugar en el que se ha llevado a cabo la misma. Estos actas han sido de gran ayuda tanto para realizar el seguimiento del proyecto como para consultar las decisiones tomadas a lo largo del mismo.

Asimismo, el seguimiento y supervisión del proyecto no se ha limitado a estas reuniones si no que se ha mantenido una comunicación constante con los directores del mismo utilizando diferentes mecanismos, como se detalla en la siguiente sección.

I.5 - Herramientas de gestión utilizadas

Para la gestión del proyecto se han utilizado varias herramientas de cara a facilitar la realización del mismo. Las herramientas utilizadas afectan a aspectos fundamentales del proyecto como la documentación, la gestión del tiempo, la comunicación o la seguridad, entre otros.

En primer lugar, se ha utilizado una wiki [W28] como base para elaborar, revisar, compartir y mantener la documentación del proyecto. Concretamente, se ha utilizado la herramienta DokuWiki [W29], un sistema de wiki sencillo orientado a crear documentación de cualquier tipo dentro de grupos de desarrollo, grupos de trabajo y pequeñas empresas. La utilización de este sistema ha aportado una gran serie de ventajas que han facilitado el desarrollo del proyecto y la comunicación con los directores del mismo. Las principales son:

- Facilidad y rapidez de aprendizaje.
- Sintaxis simple y potente.
- Elaboración y revisión de la documentación generada tanto por el alumno como por los directores del proyecto.
- Facilidad para compartir la documentación elaborada de una forma rápida y sencilla.
- Facilidad para elaborar documentación desde cualquier sitio con el único requisito de disponer de conexión a Internet.
- Seguridad y privacidad de la información.
- Posibilidad de volver a versiones anteriores.

Para la gestión de los esfuerzos realizados se necesitaba una herramienta que permitiese llevar una gestión detallada de las tareas que se iban realizando y el tiempo que se dedicaba a las mismas. Para este propósito, se consideró que la wiki no disponía de los mecanismos necesarios y se optó por emplear una hoja de cálculo compartida, utilizando para para ello Google Docs [W30]. La elección de este sistema se realizó porque mantenía las ventajas fundamentales de la wiki enumeradas anteriormente.

Otros dos sistemas fundamentales han sido el correo electrónico y Skype [W31]. El primero de ellos, ha permitido que la comunicación entre el alumno y los directores fuera rápida y fluida, facilitando y acelerando enormemente el desarrollo del proyecto. El segundo, ha permitido mantener reuniones a distancia cuando no era posible realizar las mismas de forma presencial evitando retrasos en el proyecto. Asimismo, la utilización de Skype ha permitido mantener una comunicación más fluida con los directores cuando se necesitaban concretar algunos aspectos y el correo electrónico no era suficiente.

Para poder llevar un control exhaustivo del estado del proyecto se ha utilizado un sistema de control de versiones, concretamente Subversion [W32]. Este sistema nos ha permitido gestionar de una forma flexible y segura los contenidos del proyecto independientemente de la máquina desde la que se accede a los mismos. Adicionalmente, se ha utilizado Dropbox [W33] como sistema de almacenamiento secundario. Dropbox es un sistema que permite almacenar y sincronizar archivos entre diferentes equipos a través de Internet y de forma automática. Su utilización se ha orientado tanto a la compartición de información entre los equipos de trabajo utilizados por el alumno como a la gestión de copias de seguridad. De esta forma, mediante la utilización de estos dos sistemas, no sólo ha sido muy sencillo compartir información entre los diferentes equipos utilizados para el desarrollo del proyecto si no que también se ha logrado un alto nivel de seguridad al mantener copias del trabajo realizado en diferentes lugares: servidor del sistema de control de versiones, servidor de Dropbox, equipo de trabajo del laboratorio 1.02 del I3A, portátil del alumno y equipo de sobremesa del alumno.

Anexo II - Tecnologías utilizadas

En este anexo se presentan las diferentes tecnologías utilizadas para la realización del proyecto junto con una breve descripción de las mismas y el motivo de su utilización. Las mismas se ordenan alfabéticamente

- **Apache Ant** [W34]. Herramienta utilizada para la automatizar tareas mecánicas y repetitivas, normalmente para la compilación y ejecución. Tiene la ventaja de que no depende de las órdenes del *shell* de cada sistema operativo, sino que se basa en archivos de configuración XML y clases Java para realizar las diferentes tareas, siendo idónea como solución multiplataforma. Se utiliza para facilitar la compilación y ejecución del sistema.
- **Condor** [W4]. *Middleware* de gestión de infraestructuras Grid que permite desplegar trabajos de forma sencilla en este tipo de infraestructuras. Se utiliza para ejecutar trabajos al cluster Hermes del I3A.
- **Eclipse** [W35]. Entorno de desarrollo multilenguaje, utilizado para la implementación de este proyecto. Incluye la posibilidad de ser extendido utilizando *plugins*. Se utiliza como entorno de desarrollo.
- **gLite** [W7]. *Middleware* de gestión de infraestructuras Grid que permite desplegar trabajos de forma sencilla en este tipo de infraestructuras. Se utiliza para ejecutar trabajos en las infraestructuras Grid de Aragrid y Piregrid.
- **IMAP** [W36]. Protocolo de red de acceso a mensajes electrónicos almacenados en un servidor. Se utiliza en el mediador de Condor para recuperar los correos electrónicos enviados.
- **Java** [W37]. Lenguaje de programación multiplataforma y de propósito general utilizado para implementar los diferentes componentes del proyecto.
- **Java Mail** [W38]. Biblioteca Java que permite utilizar el servicio de correo electrónico en aplicaciones Java. Concretamente se utiliza en el mediador de Condor para conectarse a través del protocolo IMAP a una dirección de correo y recuperar los correos electrónicos.
- **JDOM** [W39]. Biblioteca Java que permite parsear, manipular y generar código XML de forma rápida y sencilla. Se utliza para gestionar todos los ficheros XML manejados en el sistema.
- **JSch** [W40]. Biblioteca Java que permite conectarse a otra máquina de forma segura utilizando el protocolo SSH. Se utiliza para conectarse a las diferentes infraestructuras Grid.
- **OOjDREW** [W41]. Implementación de referencia de un motor de reglas en el formato definido por RuleML. Se utiliza como motor de reglas en el componente de gestión de fallos.

- **Oxygen** [W42]. Editor de ficheros XML que permite gestionar la creación y modificación de los mismos de forma sencilla. Ofrece una interfaz gráfica para visualizar el fichero de forma intuitiva. Se utiliza para construir los esquemas XML que definen el formato de los ficheros XML utilizados en el sistema.
- **Reference Nets** [B6]. Para. Se utiliza como lenguaje de modelado de *workflows científicos*.
- **Renew** [B7]. Simulador de redes de Petri de alto nivel basado en lenguaje de programación Java. Utiliza el paradigma de las redes de referencia (*reference nets*) para simular cualquier tipo de sistema o aplicación. Se utiliza como entorno de modelado de *workflows científicos*.
- **RuleML** [W21]. Lenguaje de reglas basado en XML, estándar para la descripción de reglas en entornos Web. Se utiliza para definir las reglas del componente de gestión de fallos.
- **SSH** [W43]. Protocolo que permite conectarse de forma segura a una máquina remota. Es el protocolo utilizado para conectarse a las infraestructuras Grid definidas en el sistema.
- **XML** [W44]. Metalenguaje extensible de etiquetas que permite definir la gramática de lenguajes específicos. Se utiliza en los ficheros de configuración del sistema.

Anexo III - Manual de usuario

III.1 - Instalación del sistema

Para instalar el sistema es necesario realizar una serie de pasos previos que nos permitirán obtener las herramientas necesarias para la instalación.

Requisitos previos

El primer paso consiste en asegurarse de que está instalado Java JDK versión 6 o superior. Si no estás seguro de si ya tienes Java instalado, o desconoces la versión instalada, existe una página web [W45] en la que puedes consultar la versión que tienes instalada de una forma rápida y sencilla.

Si no tienes Java JDK instalado puedes bajarte la última versión desde la página oficial de Java [W37]. Sólo tienes que seleccionar la plataforma adecuada y la versión adecuada dependiendo del Sistema Operativo que utilices.

Para poder utilizar WS-PTRLinda [B9] es necesario instalar el sistema gestor de bases de datos MySQL 5.1. Este sistema es utilizado por la capa de persistencia del *broker*. Él mismo se puede descargar gratuitamente de la página web oficial de MySQL [W46]. A partir de la versión 5.1, MySQL incorpora un asistente que facilita la instalación y configuración del sistema gestor de bases de datos. Además, se incluye una utilidad para crear una base de datos de forma sencilla. En cualquier caso puedes encontrar manuales de instalación y configuración en la propia Web de MySQL.

Para posibilitar la compilación del sistema y la ejecución del mismo, se utiliza la herramienta Ant de Apache [W34]. La herramienta se puede descargar desde la misma página. La instalación de la misma se realiza de forma muy sencillo a través del manual que se encuentra en la propia página web. Para facilitar la utilización del mismo, dentro del fichero comprimido que nos descargamos se proporciona un enlace a dicho manual.

Compilación del sistema

Para compilar el sistema, en la carpeta base del mismo se proporciona el fichero `build.xml` que automatiza la compilación y la ejecución del sistema. Este fichero proporciona opciones para compilar los elementos del sistema por separado como librerías o compilar todos los elementos del sistema juntos.

El listado de opciones que proporciona es el siguiente:

- *jarUserRegistry*. Compila y crea la biblioteca del registro de usuarios del sistema.
- *jarApplicationRegistry*. Compila y crea la biblioteca del registro de aplicaciones del sistema.
- *jarGridRegistry*. Compila y crea la biblioteca del registro de grids del sistema.
- *jarTrustedGridRegistry*. Compila y crea la biblioteca del registro de grids fiables del sistema.

- *jarBroker*. Compila y crea la biblioteca para utilizar la API de modelado para el Broker.
- *jarDataMovement*. Compila y crea la biblioteca del componente de movimiento de datos.
- *jarFaultManager*. Compila y crea la biblioteca del componente de gestión de fallos.
- *jarCondor*. Compila y crea la biblioteca del mediador de Condor.
- *jarGLite*. Compila y crea la biblioteca del mediador de gLite.
- *compRenew*. Compila y crea la biblioteca de la herramienta de modelado Renew.
- *compRLinda*. Compila y crea la biblioteca del broker *WS-PTRLinda*.
- *compile*. Compila y crea las librerías de todos los elementos del sistema.

Para compilar el sistema debe utilizarse la herramienta *ant* indicando la opción elegida como se muestra a continuación:

```
$ ant compile
```

III.2 - Configuración del sistema

Antes de utilizar el sistema es necesario configurar el mismo, indicando los usuarios que pueden utilizarlo, los grids que se pueden utilizar, tanto normales como fiables, y las aplicaciones que pueden ser ejecutadas en cada grid. Se recomienda la realización de las tareas de configuración antes de comenzar a ejecutar el programa, en cualquier caso, la configuración puede modificarse en cualquier momento sin necesidad de detener el sistema.

Las tareas de configuración que se permiten son la adición, modificación y eliminación de elementos, tanto para usuarios como para grids y aplicaciones. Para facilitar la tarea se proporcionan scripts que facilitan esta labor. Todos estos scripts se proporcionan dentro de la carpeta `scripts` que se encuentra en el directorio base del sistema.

Al añadir diferentes elementos, se deben indicar una serie de atributos, como se detalla a continuación. Algunos atributos son obligatorios mientras que otros son opcionales por lo que no es necesario especificar valor alguno. Para indicar que un atributo no tiene valor se utiliza la palabra `null` o el carácter `*`. En el caso de que algún atributo esté compuesta de varias palabras separadas se debe indicar el mismo entre comillas dobles.

Configuración de *grids*

La información que debe ser almacenada para cada Grid es la siguiente.

- El número de CPUs que forman el Grid.
- La cantidad memoria media por procesador (en Megabytes).
- El rendimiento del Grid expresado en FLOPs.
- El coste de utilización del Grid por hora de procesador (en Euros).
- Opciones personalizadas para la ejecución de trabajos en ese Grid.
- Una lista de los diferentes servicios o protocolos que pueden ser utilizados para transferir ficheros dentro del grid.

De los anteriores atributos, sólo las opciones personalizadas son optativas. Además, debe indicarse como mínimo un servicio de transferencia para el *grid* sin que exista límite máximo.

Para añadir un nuevo g*rid* o modificar la información de uno existente se proporciona el *script* `add_grid` que debe utilizarse de la siguiente manera:

```
$ add_grid host cpus avg_memory FLOPS cost custom_options service [service]
```

Para eliminar un *Grid* se debe utilizar el *script* `remove_grid` de la siguiente forma:

```
$ remove_grid host
```

Configuración de *grids* fiables

Para añadir infraestructuras *grid* fiables se utiliza el mismo mecanismo que en el caso anterior y se debe proporcionar la misma información. La única diferencia es que el *script* utilizado para añadir o modificar los grids se denomina `add_trusted_grid` y el *script* utilizado para eliminar grids fiables se llama `remove_trusted_grid`.

Configuración de usuarios

Para cada usuario se almacena la información de acceso del mismo a los diferentes grids. Concretamente, la información almacenada para que el usuario pueda acceder a un grid concreto es es la siguiente:

- Alias utilizado por el usuario para acceder al Grid.
- Nombre del servidor que da acceso al Grid.
- Contraseña utilizada para acceder al Grid.
- Ruta absoluta del directorio *home* del usuario.
- Permisos de acceso en el grid.
- Dirección de correo electrónico asociada.
- Contraseña para la dirección de correo electrónico.
- Opciones personalizadas para el usuario en el Grid.

En este caso, se permite que la información asociada al correo electrónico sea nula ya que la misma sólo es necesaria en algunos Grids, mientras que en otros la misma no es necesaria. Lo mismo ocurre con las opciones personalizadas, si se indican opciones personalizadas, las mismas deben realizarse en el formato utilizado por el *middleware* del Grid.

Para añadir el alias de un nuevo usuario o modificar la información del mismo en un determinado grid se proporciona el script `add_user` que debe utilizarse de la siguiente forma:

```
$ add_user global_user host local_user password home permissions email
email_password custom_options
```

De la misma forma, para eliminar el usuario de un grid se debe utilizar el script `remove_user` de la siguiente forma:

```
$ remove_user global_user host
```

Configuración de aplicaciones

Finalmente, para añadir una nueva aplicación a un *grid* se debe proporcionar la siguiente información:

- Nombre de la aplicación.
- Ruta de ejecución de la aplicación.
- Tipo de aplicación: C, java, shell, etc.
- Máxima memoria utilizada por la aplicación.
- Sistema Operativo que debe utilizarse por la aplicación
- Arquitectura que debe utilizarse para ejecutar la aplicación.
- Opciones de despliegue personalizadas por parte del usuario.

En este caso, los tres últimos atributos son opcionales y no es obligatorio proporcionar un valor para los mismos.

Para añadir una nueva aplicación a un *grid* o modificar la información de la misma se proporciona el *script* `add_application`. Su utilización es la siguiente:

```
$ add_application host app_name app_path app_type app_max_memory app_cpus
app_OS app_architecture custom_options
```

Para eliminar una aplicación se proporciona el *Script* `remove_application` que se utiliza de la siguiente forma:

```
$ remove_application host application_name
```

III.3 - Utilización del sistema

Para ejecutar el entorno de modelado de *workflows científicos* que proporciona la herramienta Renew, debemos entrar en el directorio base de ejecución y utilizar la aplicación ant. Para ello introducimos la siguiente orden:

```
$ ant run-gui
```

Nos aparecerá el entorno de ejecución de Renew con el menú que puede observarse en la figura 23. En [W47] puede consultarse el manual de la herramienta Renew. En el mismo se detallan todos los aspectos referentes a la utilización de la herramienta y se da una breve introducción al diseño de sistemas y aplicaciones utilizando la misma.

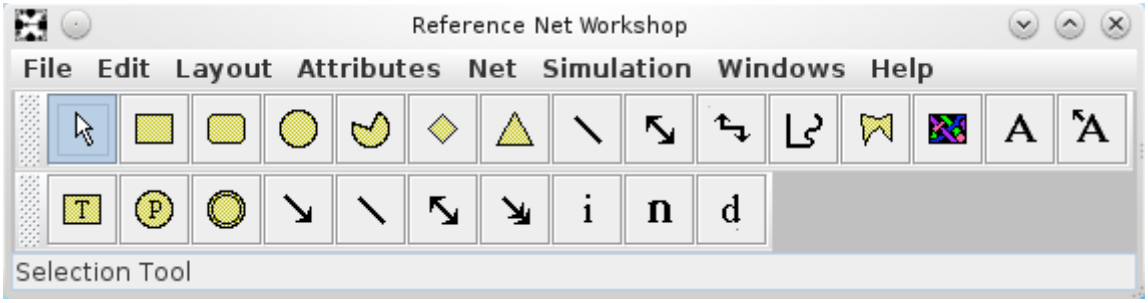

Figura 23: Barra de herramientas de Renew

Una vez creado el modelo correspondiente al *workflow* científico que queremos ejecutar debe incluirse en el modelo la red que aparece en la figura 24. Concretamente, en esa figura se indica la red que se debe utilizar para utilizar a la vez los Grids Hermes y Piregrid. Esta red de lanzamiento se proporciona en el fichero `launcher_net.rnw` dentro de la carpeta `nets`. La ejecución de la misma debe realizarse antes de que comience la ejecución del modelo. Para ello, se propone que desde la transición con nombre `Create_broker` se coloque un nuevo lugar de salida que, al recibir un token, indique que se puede comenzar la ejecución del modelo.

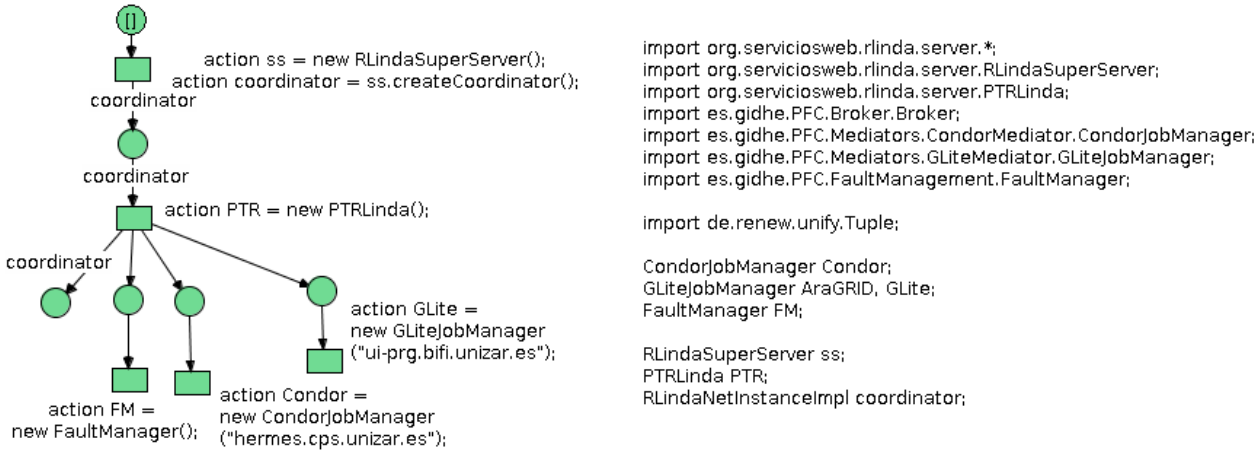

Figura 24: Red de lanzamiento del sistema

Como se puede observar en la figura, la creación de un nuevo componente se realiza creando un nuevo objeto Java. A continuación, se indica la clase de la que debe crearse el nuevo objeto para crear un nuevo componente:

- Componente de gestión de fallos. Debe crearse un nuevo objeto de la clase `FaultManagement`. No se debe indicar ningún parámetro.
- Mediador de Condor. Debe crearse un nuevo objeto de la clase `CondorJobManager`. Debe especificarse como parámetro el nombre del *host* de la infraestructura.
- Mediador de gLite. Debe crearse un nuevo objeto de la clase `GLiteJobManager`. Debe especificarse como parámetro el nombre del *host* de la infraestructura.

Además, de la red que aparece en la parte izquierda, deben incluirse las definiciones que aparecen la parte derecha de la figura 24 para que el sistema funcione correctamente.

Finalmente, un aspecto adicional es que para que se puedan ejecutar los problemas modelados, la red denominada *RLindaCoordinator* debe estar abierta en el momento de ejecutar el modelo. Esta red se encuentra en la carpeta `nets` dentro del directorio base del sistema y el fichero que la contiene se llama `RLindaCoordinator.rnw`.

Si se desea utilizar otro sistema de gestión de *workflows* científicos para realizar el modelado del sistema, deben seguirse las instrucciones del mismo para modelar el sistema. Una vez realizado el modelo, se puede realizar la ejecución del mismo utilizando la interfaz de servicios web del Broker. Más detalles sobre este aspecto pueden consultarse en el manual de usuario de WS-PTRLinda que aparece en [B9].

Un ejemplo de modelado de un *workflow* científico se proporciona en el *Anexo X*.

Anexo IV - Registros del sistema

Existen diferentes maneras de gestionar la información relativa al funcionamiento de un sistema de gestión de *workflows*. Algunos sistemas optan por almacenar determinada información y acceder a la misma cuando la necesitan, otros optan por no almacenar la información y realizar consultas al *middleware* para obtener la misma cuando la necesitan. Independientemente de la alternativa utilizada, existe información que es necesario almacenar ya que no puede ser obtenida bajo demanda, como los usuarios que tienen acceso a la infraestructura, por lo cual la utilización de registros de información se hace obligatoria.

En este anexo, vamos a detallar las características comunes y particulares de los cuatro tipos de registros de información utilizados: información sobre los *grids* utilizados pudiendo ser estos normales o fiables, información sobre los usuarios del sistema e información sobre las aplicaciones que es posible ejecutar en cada infraestructura.

IV.1 - Características comunes

El *framework* propuesto debe manejar información referente a los *grids* disponibles, tanto fiables como normales, usuarios que pueden ejecutar en los *grids* y aplicaciones que puede ejecutar cada *grid*. Por tanto, se hace necesaria la utilización de diferentes almacenes para los diferentes tipos de información tratada.

Las características que deben tener estos almacenes de información vienen dictadas por los objetivos del proyecto. En primer lugar, se necesita que el sistema sea flexible y adaptable frente a cambios en el entorno. Por tanto, debe ser posible modificar la información contenida en estos registros en tiempo de ejecución sin que afecte a la utilización del sistema. Asimismo, el sistema debe ser fácil de configurar, por lo que un usuario cualquiera debe ser capaz de configurar el mismo rellenando correctamente los registros del sistema. Para facilitar las cuestiones de configuración se proporcionan *scripts* que permiten configurar los diferentes almacenes de una forma sencilla. Estos *scripts* permiten insertar nueva información, modificar la existente y eliminar la información actual. Su utilización puede consultarse en el *Anexo III - Manual de Usuario*.

En lo que respecta a almacenar la información de usuarios, aplicaciones y *grids*, queda claro que ésta debe almacenarse de forma separada ya que se trata de información muy diferente. Sin embargo, separar la información relativa a los *grids* fiables y la información relativa a los *grids* normales no estaba tan claro. Finalmente, se decidió separar la información ya que se considera que es información de distinto tipo. Además, separar la misma va a permitir realizar una gestión de la información más sencilla, tener un acceso más eficiente a la misma y personalizar cada tipo de información por separado de acuerdo a sus necesidades.

Para almacenar la información se decidió utilizar ficheros XML por cuestiones de simplicidad y facilidad en cuanto a manejo de la información y compartición de la misma. Dichos ficheros se encuentran en la carpeta de configuración (directorio *config*) del *framework*. En lo que respecta al convenio de nombres para cada uno de estos ficheros, el mismo se comenta posteriormente, en las secciones dedicadas a cada registro dentro de este mismo anexo.

Una característica que se introduce en los ficheros para facilitar la comprensión de los mismos por parte del usuario, es que no se permiten elementos optativos, todos los elementos son obligatorios, es decir, no puede haber elementos que no aparezcan en el fichero de configuración. Sin embargo, sí que se permite que haya determinados atributos con un valor indefinido. Cuando se quiere expresar esta característica se utiliza el carácter '*' o la cadena 'null' para representar que el valor del atributo es indefinido.

A pesar de que cada tipo de registro trata diferente tipo de información, en todos los registros se incluye un campo para almacenar opciones personalizadas. Este campo se utiliza para indicar opciones personalizadas para la ejecución de una determinada aplicación, todas las aplicaciones de un usuario en un *grid* o todos los trabajos realizados al *grid*, dependiendo del registro que se trate. Esta característica proporciona al usuario una mayor flexibilidad, pudiendo adaptar la ejecución de cada trabajo de una forma más precisa si lo desea. Estas opciones deben proporcionarse en un formato que sea entendible directamente por el *middleware* del *grid* ya que no se realiza ningún tratamiento de las mismas, si no que se añaden directamente.

Como existe la posibilidad de que estas opciones personalizadas se solapen, se impone el siguiente mecanismo de prioridad: las opciones menos prioritarias son las indicadas en el registro del usuario, después las indicadas en el registro del *grid* y, finalmente, las más prioritarias son las específicas de la aplicación. Por tanto, si en algún momento dichas opciones se contradicen se tomará la alternativa fijada por el registro con mayor prioridad. En cualquier caso, la configuración propia del trabajo indicada por el mediador tiene mayor prioridad que las opciones personalizadas de los registros para impedir comportamientos erróneos o malévolos.

En lo que se refiere al diseño de los registros, se utiliza un componente diferente para gestionar cada uno de ellos si bien todos son análogos. La arquitectura de estos componentes se presenta en la figura 25. En lo que respecta al nombre de cada uno de los subcomponentes, la palabra *Element* se sustituye por la palabra adecuada dependiendo del componente del que se trate. El componente *ElementDiscovery* es el encargado de consultar la información almacenada en el registro permitiendo consultar por un elemento determinado o por toda la lista de elementos. Por su parte, los elementos *ElementRegister* y *ElementEraser* son los que permiten añadir y modificar y eliminar información del registro, respectivamente.

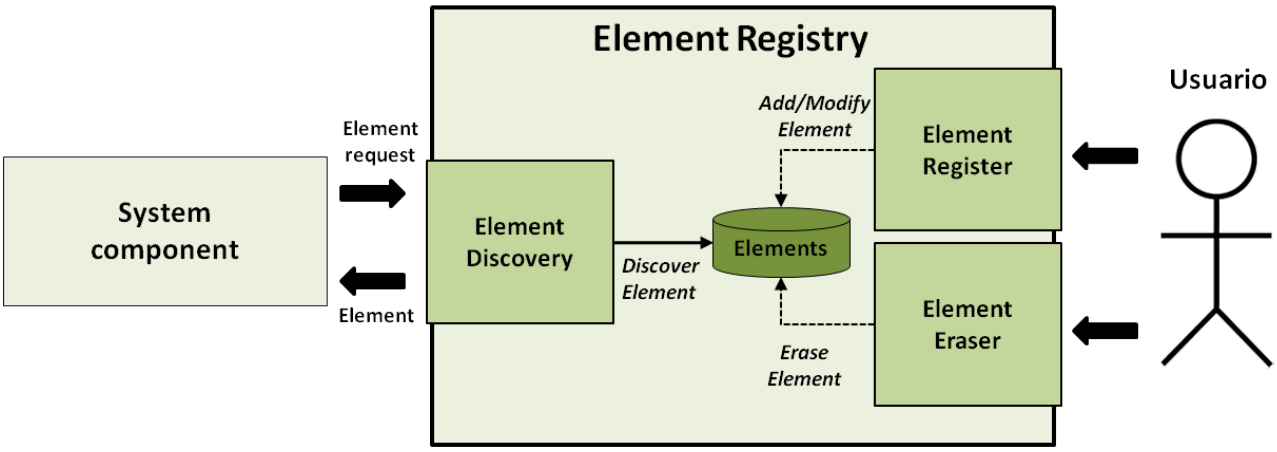

Figura 25: Arquitectura genérica de los componentes que implementan los registros

IV.2 - Registro de *Grids*

El registro de *grids* se implementa mediante un sólo fichero de configuración que contiene la lista de *grids* que pueden ser utilizados por el sistema. Dicho fichero tiene como nombre `grids.conf` y se encuentra en el directorio de configuración del sistema. Para cada *grid* se almacena la siguiente información:

- El número de CPUs o procesadores que forman el *Grid*.
- La cantidad memoria media por procesador (en Megabytes).
- El rendimiento del *grid* expresado en FLOPs.
- El coste de utilización del *grid* por hora de procesador (en Euros).
- Opciones personalizadas para la ejecución de trabajos en ese *grid*.
- Una lista de los diferentes servicios o protocolos que pueden ser utilizados para transferir ficheros dentro del *grid*.

Como en esta primera versión, todavía no se da soporte para calidad de servicio (QoS), el valor de los cuatro primeros atributos no se utiliza, pero en posteriores versiones de la aplicación se utilizarán para gestionar los requisitos de QoS definidos. Por su parte la lista de servicios o protocolos se utiliza para permitir la transferencia de ficheros al *grid*.

En este caso, el único valor optativo es el de las opciones personalizadas. A pesar de que no se da soporte para QoS actualmente, se ha considerado que los atributos relacionados con el mismo deben ser obligatorios con el objetivo de que futuras versiones del sistema no provoquen modificaciones en los ficheros de configuración.

El esquema utilizado para el fichero de configuración de Grids puede observarse en la figura 26, mientras que un ejemplo de fichero de configuración de grids para los grids Hermes, Aragrid y Piregrid puede verse en la figura 27. En esta segunda figura puede observarse un ejemplo de uso de las opciones personalizadas, en la configuración del grid Hermes (hermes.cps.unizar.es). En este caso se indica que, por defecto, no debe utilizarse el entorno del usuario sino el entorno del propio sistema utilizando la opción `get_env = false`.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
  <!-- GRIDS ELEMENT (Root element) -->
  <xs:element name="grids">
    <xs:complexType>
      <xs:sequence>
        <xs:element minOccurs="1" maxOccurs="unbounded" ref="grid"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <!-- GRID ELEMENT -->
  <xs:element name="grid">
    <xs:complexType>
      <xs:sequence>
        <xs:element minOccurs="1" maxOccurs="1" ref="cpus"/>
        <xs:element minOccurs="1" maxOccurs="1" ref="avg_memory"/>
        <xs:element minOccurs="1" maxOccurs="1" ref="flops"/>
        <xs:element minOccurs="1" maxOccurs="1" ref="cost"/>
        <xs:element minOccurs="1" maxOccurs="1" ref="custom_options"/>
```

```
        <xs:element minOccurs="1" maxOccurs="unbounded" ref="service"/>
      </xs:sequence>
      <xs:attribute name="host" use="required" type="xs:NCName"/>
    </xs:complexType>
  </xs:element>
  <!-- Grid tags -->
  <xs:element name="cpus" type="xs:integer"/>
  <xs:element name="memory" type="xs:integer"/>
  <xs:element name="FLOPS" type="xs:integer"/>
  <xs:element name="cost" type="xs:float"/>
  <xs:element name="custom_options" type="xs:normalizedString"/>
  <xs:element name="service" type="xs:NCName"/>
</xs:schema>
```

Figura 26: Esquema para los ficheros de configuración de Grids. Fichero `grids.xsd`.

```
<?xml version="1.0" encoding="UTF-8"?>
<grids>
  <grid host="hermes.cps.unizar.es">
    <cpus>1400</cpus>
    <avg_memory>1024</avg_memory>
    <flops>1000000</flops>
    <cost>0.3</cost>
    <custom_options>getenv = False</custom_options>
    <service>sftp</service>
    <service>scp</service>
  </grid>
  <grid host="ui-prg.bifi.unizar.es">
    <cpus>400</cpus>
    <avg_memory>2048</avg_memory>
    <flops>500000</flops>
    <cost>0</cost>
    <custom_options>*</custom_options>
    <service>scp</service>
  </grid>
  <grid host="ui-ara.bifi.unizar.es">
    <cpus>2000</cpus>
    <avg_memory>5000</avg_memory>
    <flops>2000000</flops>
    <cost>0</cost>
    <custom_options>*</custom_options>
    <service>scp</service>
    <service>sftp</service>
 </grid>
</grids>
```

Figura 27: Ejemplo de fichero de configuración de grids.

IV.3 - Registro de *Grids* fiables

Este registro es análogo al anterior, la única diferencia es que se utilizan diferentes etiquetas para identificar los componentes del fichero de configuración.

La figura 28 muestra el esquema utilizado para el fichero de configuración `trusted_grids.conf` mientras que la figura 29 presenta un ejemplo de configuración de dicho fichero para el servicio de computación *Cloud* de Amazon, Amazon EC2 [W9].

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
  <!-- TRUSTED_GRIDS ELEMENT (Root element) -->
```

```
  <xs:element name="trusted_grids">
    <xs:complexType>
      <xs:sequence>
        <xs:element minOccurs="1" maxOccurs="unbounded" ref="grid"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <!-- TRUSTED_GRID ELEMENT -->
  <xs:element name="trusted_grid">
    <xs:complexType>
      <xs:sequence>
        <xs:element minOccurs="1" maxOccurs="1" ref="cpus"/>
        <xs:element minOccurs="1" maxOccurs="1" ref="avg_memory"/>
        <xs:element minOccurs="1" maxOccurs="1" ref="flops"/>
        <xs:element minOccurs="1" maxOccurs="1" ref="cost"/>
        <xs:element minOccurs="1" maxOccurs="1" ref="custom_options"/>
        <xs:element minOccurs="1" maxOccurs="unbounded" ref="service"/>
      </xs:sequence>
      <xs:attribute name="host" use="required" type="xs:NCName"/>
    </xs:complexType>
  </xs:element>
  <!-- Trusted Grid tags -->
  <xs:element name="cpus" type="xs:integer"/>
  <xs:element name="memory" type="xs:integer"/>
  <xs:element name="FLOPS" type="xs:integer"/>
  <xs:element name="cost" type="xs:float"/>
  <xs:element name="custom_options" type="xs:normalizedString"/>
  <xs:element name="service" type="xs:NCName"/>
</xs:schema>
```

Figura 28: Esquema para los ficheros de configuración de Grids fiables. Fichero `trusted_grids.xsd`.

```
<?xml version="1.0" encoding="UTF-8"?>
<trusted_grids>
  <trusted_grid host="AmazonEC2">
    <cpus>1</cpus>
    <avg_memory>1700</avg_memory>
    <flops>10000</flops>
    <cost>0.095</cost>
    <custom_options>*</custom_options>
    <service>scp</service>
  </trusted_grid>
</trusted_grids>
```

Figura 29: Ejemplo de fichero de configuración de grids fiables

IV.4 - Registro de usuarios

El registro de usuarios se compone de un número de ficheros variable, utilizándose un fichero para cada usuario del sistema. Esto se debe a que en este fichero se gestiona información sensible, como contraseñas, por lo que no es conveniente mezclar ficheros de diferentes usuarios. De esta forma, el usuario puede establecer sus propios permisos para el fichero haciendo que otros usuarios no puedan acceder al mismo, dotando de mayor seguridad a sus datos.

La información gestionada en estos ficheros de configuración hace referencia al acceso del usuario a los diferentes *grids* del sistema. Asimismo, se incluyen aspectos adicionales como los permisos del usuario o la información de acceso a la cuenta de correo electrónico del

usuario, ya que algunos sistemas utilizan este mecanismo para notificar el fin de un trabajo o la aparición de algún error al ejecutar el mismo.

En lo que respecta a los nombres de estos ficheros todos siguen la convención `username.users.conf`. Por ejemplo, para un usuario con nombre `sergio` el fichero de configuración asociado será `sergio.users.conf`.

Para cada usuario se almacena la información de acceso del mismo a los diferentes grids. Concretamente, la información almacenada para que el usuario pueda acceder a un grid concreto es es la siguiente:

- Alias utilizado por el usuario para acceder a dicho grid.
- Nombre del servidor que da acceso al Grid.
- Contraseña utilizada para acceder al Grid.
- Ruta aboluta del directorio *home* del usuario.
- Permisos de acceso en el grid.
- Dirección de correo electrónico asociada.
- Contraseña para la dirección de correo electrónico.
- Opciones personalizadas para el usuario en el Grid.

En este caso, se permite que la información asociada al correo electrónico sea nula ya que la misma sólo es necesaria en algunos Grids, mientras que en otros la misma no es necesaria. De la misma forma, la información de opciones personalizadas también es optativa.

En la figura 30 se muestra el esquema utilizado para los ficheros de configuración de usuarios, mientras que, en la figura 31, se muestra un ejemplo de fichero de configuración para el usuario *sergio*. En este caso, para el *grid* Hermes se indica que la ejecución de los trabajos debe realizarse como usuario *nice*, un usuario con una menor prioridad. Esta circunstancia se indica en las opciones personalizadas con la opción `nice_user = True`.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
  <!-- USERS ELEMENT (Root Element) -->
  <xs:element name="users">
    <xs:complexType>
      <xs:sequence>
        <xs:element minOccurs="1" maxOccurs="unbounded" ref="user"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <!-- USER ELEMENT -->
  <xs:element name="user">
    <xs:complexType>
      <xs:sequence>
        <xs:element minOccurs="1" maxOccurs="1" ref="password"/>
        <xs:element minOccurs="1" maxOccurs="1" ref="home"/>
        <xs:element minOccurs="1" maxOccurs="1" ref="permisssions"/>
        <xs:element minOccurs="1" maxOccurs="1" ref="email"/>
        <xs:element minOccurs="1" maxOccurs="1" ref="email_password"/>
        <xs:element minOccurs="1" maxOccurs="1" ref="custom_options"/>
      </xs:sequence>
```

```
      <xs:attribute name="host" use="required" type="xs:NCName"/>
      <xs:attribute name="name" use="required" type="xs:NCName"/>
    </xs:complexType>
  </xs:element>
  <!-- USER TAGS -->
  <xs:element name="password" type="xs:normalizedString"/>
  <xs:element name="home" type="xs:normalizedString"/>
  <xs:element name="permissions" type="xs:normalizedString"/>
  <xs:element name="email" type="xs:normalizedString"/>
  <xs:element name="email_password" type="xs:normalizedString"/>
  <xs:element name="custom_options" type="xs:normalizedString"/>
</xs:schema>
```

Figura 30: Esquema para los ficheros de configuración de Grids. Fichero *users.xsd*.

```
<?xml version="1.0" encoding="UTF-8"?>
<users>
  <user name="shernandez" host="ui-prg.bifi.unizar.es">
    <password>pass1</password>
    <home>/home/shernandez/</home>
    <permissions>-rwxr--r--</permissions>
    <email>*</email>
    <emailPassword>null</emailPassword>
  </user>
    <user name="shernandez" host="ui-ara.bifi.unizar.es">
    <password>pass2</password>
    <home>/home/shernandez/</home>
    <permissions>-rwxr--r--</permissions>
    <email>null</email>
    <emailPassword>*</emailPassword>
  </user>
  <user name="shernand" host="hermes.cps.unizar.es">
    <password>pass3</password>
    <home>/home/gidhe/shernand/</home>
    <permisssions>-rwxr--r--</permisssions>
    <email>srg46srg@gmail.com</email>
    <emailPassword>pass4</emailPassword>
  </user>
  <user name="554804@unizar.es" host="AmazonEC2">
    <password>pass5</password>
    <home>/home/sergio/</home>
    <permisssions>root</permisssions>
    <email>*</email>
    <emailPassword>*</emailPassword>
  </user>
</users>
```

Figura 31: Ejemplo de fichero de configuración para el usuario *sergio*.

IV.5 - Registro de aplicaciones

El registro de aplicaciones está formado por varios ficheros, un fichero por cada *grid* presente en el sistema. La decisión de separar los ficheros de configuración de aplicaciones de cada *grid* se debe a que el número de aplicaciones que pueden contener los mismos va a ser en general alto. Por lo tanto, es aconsejable separar la información por cuestiones de eficiencia.

En lo que respecta al criterio de nombres utilizado, se incluye el primer lugar el identificador del grid (su nombre de *host*) seguido de la cadena `.app.conf`. Para el caso del grid Hermes, con nombre *hermes.cps.unizar.es*, el fichero de configuración asociado al mismo es `hermes.cps.unizar.es.app.conf`.

En estos ficheros se incluyen las aplicaciones que cada *grid* es capaz de ejecutar. Por tanto, esta información es muy importante a la hora de realizar la decisión de en qué infraestructura ejecutar un trabajo y determinar si el *grid* es capaz de ejecutar dicha aplicación. Los aspectos almacenados para cada aplicación son los siguientes:

- Nombre de la aplicación.
- Ruta de ejecución de la aplicación.
- Tipo de aplicación: C, java, shell, etc.
- Máxima memoria utilizada por la aplicación.
- Sistema Operativo que debe utilizarse por la aplicación
- Arquitectura que debe utilizarse para ejecutar la aplicación.
- Opciones de despliegue personalizadas por parte del usuario.

En este caso, sólo es opcional la información relacionada con el sistema operativo y la arquitectura ya que hay aplicaciones que pueden ejecutarse en cualquier arquitectura y/o sistema operativo, como es el caso de las aplicaciones java. Asimismo, las opciones personalizadas también son opcionales.

En la figura 32 se muestra el esquema utilizado para los ficheros de configuración de aplicaciones, mientras que, en la figura 33, se muestra un ejemplo de fichero de configuración para el Grid Hermes. En este caso, el *grid* Hermes proporciona aspectos de configuración propios para mejorar el rendimiento de los trabajos ejecutados. Para ello, la infraestructura permite definir trabajos como *longJobs*, trabajos de larga duración, *shortJobs*, trabajos de corta duración, y *bigJobs*, trabajos que necesitan de mucha potencia de cálculo. Como ejemplo, se define la aplicación `align_warp` como *shortJob*.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
   <!-- APPLICATIONS ELEMENT (Root element) -->
  <xs:element name="applications">
    <xs:complexType>
      <xs:sequence>
        <xs:element minOccurs="1" maxOccurs="unbounded" ref="application"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <!-- APPLICATION ELEMENT -->
  <xs:element name="application">
    <xs:complexType>
      <xs:sequence>
        <xs:element minOccurs="1" maxOccurs="1" ref="path"/>
        <xs:element minOccurs="1" maxOccurs="1" ref="type"/>
        <xs:element minOccurs="1" maxOccurs="1" ref="max_memory"/>
        <xs:element minOccurs="1" maxOccurs="1" ref="cpus"/>
        <xs:element minOccurs="1" maxOccurs="1" ref="OS"/>
        <xs:element minOccurs="1" maxOccurs="1" ref="architecture"/>
        <xs:element minOccurs="1" maxOccurs="1" ref="custom_options"/>
      </xs:sequence>
      <!-- Application name -->
      <xs:attribute name="name" use="required" type="xs:NCName"/>
    </xs:complexType>
```

```
  </xs:element>
  <!-- APPLICATION TAGS -->
  <xs:element name="path" type="xs:normalizedString"/>
  <xs:element name="type" type="xs:normalizedString"/>
  <xs:element name="max_memory" type="xs:integer"/>
  <xs:element name="cpus" type="xs:integer"/>
  <xs:element name="OS" type="xs:normalizedString"/>
  <xs:element name="architecture" type="xs:normalizedString"/>
  <xs:element name="custom_options" type="xs:normalizedString"/>
</xs:schema>
```

Figura 32: Esquema para los ficheros de configuración de Grids. Fichero *applications.xsd*.

```
<?xml version="1.0" encoding="UTF-8"?>
<applications>
   <application name="align_warp">
      <path>/home/gidhe/shernand/align_warp</path>
      <type>C</type>
      <max_memory>1024</max_memory>
      <cpus>1</cpus>
      <OS>*</OS>
      <architecture>*</architecture>
      <custom_options>+shortjob = TRUE</custom_options>
  </application>
  <application name="reslice">
      <path>/home/gidhe/shernand/reslice</path>
      <type>C</type>
      <max_memory>1024</max_memory>
      <cpus>1</cpus>
      <OS>*</OS>
      <architecture>*</architecture>
      <custom_options>*</custom_options>
  </application>
     <application name="softmean">
      <path>/home/gidhe/shernand/softmean</path>
      <type>C</type>
      <max_memory>1024</max_memory>
      <cpus>1</cpus>
      <OS>*</OS>
      <architecture>*</architecture>
      <custom_options>*</custom_options>
  </application>
  <application name="slicer">
      <path>/home/gidhe/shernand/slicer</path>
      <type>C</type>
      <max_memory>1024</max_memory>
      <cpus>1</cpus>
      <OS>*</OS>
      <architecture>*</architecture>
      <custom_options>*</custom_options>
  </application>
  <application name="convert">
      <path>/home/gidhe/shernand/convert</path>
      <type>C</type>
      <max_memory>1024</max_memory>
      <cpus>1</cpus>
      <OS>*</OS>
      <architecture>*</architecture>
      <custom_options>*</custom_options>
  </application>
</applications>
```

Figura 33: Ejemplo de fichero de configuración de aplicaciones para el Grid Hermes.

Anexo V - *Broker* de coordinación

La coordinación y comunicación entre los diferentes componentes de un sistema de gestión de *workflows* científicos se ha solventado tradicionalmente utilizando componentes que se encuentran muy acoplados unos con otros. Además, este acoplamiento afecta tanto a los componentes de nivel de ejecución del sistema como a los componentes encargados del modelado de los problemas. De esta forma, la integración de nuevos componentes tiene un impacto muy grande en todo el sistema, dificultando enormemente esta tarea.

El uso de un *broker* de mensajes basado en el modelo de coordinación Linda aporta un nuevo enfoque en este aspecto. El modelo de coordinación Linda promueve la coordinación entre los diferentes componentes a través de tuplas que son depositadas y extraídas de un espacio compartido. Además, cada componente puede estar distribuido, es decir, puede encontrarse en una máquina diferente y utilizar Linda para comunicarse con el resto de componentes del sistema. Esta característica permite desacoplar los diferentes elementos que componen el sistema haciendo que el mismo sea más flexible y adaptable.

En nuestro caso hemos utilizado como *broker* de mensajes WS-PTRLinda [B9], una implementación del modelo de coordinación Linda del grupo de investigación GIDHE [W1]. Las principales mejoras que aporta esta implementación son la adición al modelo de Linda de una capa de temporización, permite que las tuplas sólo sean válidas durante un período de tiempo determinado, y una capa de persistencia, permite almacenar las tuplas en una base de datos. Asimismo, WS-PTRLinda incluye una capa que permite interaccionar con el sistema como un servicio web SOAP o REST, haciendo que el mismo sea accesible desde cualquier punto.

La utilización de Linda nos permite conectar al sistema un número variable de componentes que aporten diferentes funcionalidades. Algunos ejemplos de posibles componentes serían un componente gestión de fallos que permite tratar los fallos que se produzcan al ejecutar los trabajos, un componente de monitorización que permita obtener información detallada de los trabajos que se están ejecutando o un componente de *scheduling* avanzado que permite realizar el despliegue de los trabajos en base a diferentes parámetros y configuraciones.

V.1 - Comunicación entre los componentes del sistema

La utilización de Linda permite desacoplar la comunicación entre los diferentes componentes del sistema haciendo que la misma sea asíncrona. Cuando ocurre un cierto evento en un componente, éste deposita una tupla en el espacio de tuplas gestionado por el coordinar de Linda. Esa tupla será leída en cualquier momento por otro componente utilizando una tupla patrón que concuerde con la tupla anterior.

Como puede observarse en la figura 34, que muestra la arquitectura del sistema, existen varios componentes que utilizan el *broker* para coordinarse y comunicarse. En primer lugar, el entorno de ejecución de Renew, o de otro sistema de gestión de *workflows* científicos, deposita una tupla en el *broker*. Posteriormente, esta tupla es leída por alguno de los mediadores. Cuando el trabajo finaliza, el mediador correspondiente deposita otra tupla en el *broker* para que el entorno de ejecución del sistema de *workflows* científicos recoja el resultado del trabajo.

Adicionalmente, las tuplas pueden ser leídas por componentes adicionales como el componente de gestión de fallos.

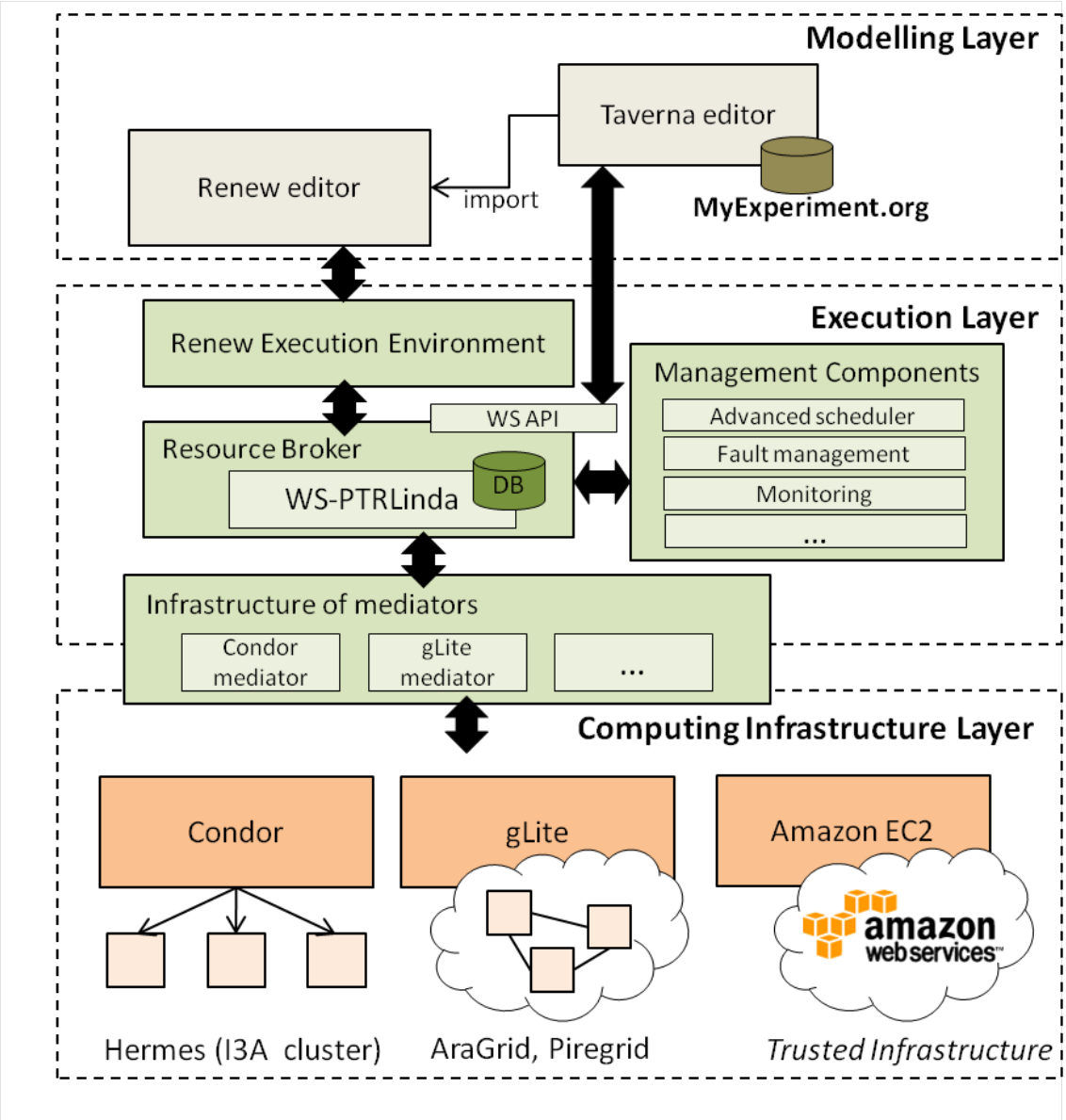

Figura 34: Arquitectura del sistema.

Para que cada componente sea capaz de distinguir y detectar qué tuplas debe gestionar se realizan diferentes acciones. En primer lugar, cada uno de los componentes añade al principio de la tupla una palabra clave que identifica el evento que se ha producido. El entorno de ejecución utiliza la palabra clave `DEPLOYED` para indicar que esa tupla solicita la ejecución de un trabajo. Este tipo de tuplas son detectadas y tratadas por los diferentes mediadores definidos. Estos, a su vez, depositan tuplas en el *broker* con palabra clave `EXECUTED`, para indicar que el trabajo se ha ejecutado y devolver sus resultados, o `ERROR`, para indicar que el trabajo ha sufrido algún fallo. Si el trabajo ha finalizado correctamente el entorno de ejecución del sistema de gestión de *workflows* científicos detecta la tupla y continúa con la ejecución del mismo. Si por contra, el trabajo ha fallado, es el componente de gestión de fallos el que recoge la tupla. Este componente gestiona el fallo y deposita en el *broker* una tupla indicando que se vuelva a ejecutar el trabajo (palabra clave `DEPLOYED`) o una tupla indicando que ha finalizado la ejecución porque el fallo no es corregible (palabra clave `EXECUTED` con estado de error).

De esta forma, la adición de nuevos componentes es muy sencilla. Tan sólo es necesario que el nuevo componente utilice otra palabra clave para identificar los eventos adecuados. De igual manera, el reemplazo de un componente por otro que realice la misma labor o la actualización del mismo es completamente transparente al resto del sistema haciendo que el mismo sea muy adaptable.

Otro aspecto fundamental para realizar la correspondencia entre las tuplas depositadas y los componentes que la gestionan es la denominada política de *matching* o emparejamiento. Cuando se deposita una tupla, se utiliza un patrón para recuperar la misma mediante una operación de lectura. Este patrón puede ser exactamente igual a la tupla, en ese caso el emparejamiento es trivial, o puede contener comodines que pueden reemplazar cualquier elemento de la tupla. Por defecto, existen dos tipos de políticas de emparejamiento:

- *Emparejamiento fuerte*. Los comodines del patrón sólo pueden corresponderse en la tupla emparejada con elementos atómicos (números enteros, cadenas, etc.) no con otras tuplas.
- *Emparejamiento débil*. Los comodines del patrón pueden emparejarse tanto con elementos atómicos como con otras tuplas.

En las operaciones de lectura realizadas por los componentes adicionales y por el sistema de gestión de *workflows* científicos utilizamos la política de emparejamiento débil ya que es más flexible que la política de emparejamiento fuerte. Por contra, no se puede utilizar este tipo de política en las operaciones de lectura realizadas por los mediadores.

Cuando se define un trabajo, puede indicarse el grid en el que se quiere ejecutar el mismo. Si no se indica este aspecto, el sistema decide en qué infraestructura se va a ejecutar el trabajo. Es en este segundo caso cuando la política de emparejamiento débil no funciona. En la figura 35 se muestra un ejemplo del problema. Los mediadores esperan tuplas que vayan dirigidas a los mismos, sin embargo, cuando no se especifica un grid, ni el emparejamiento débil ni el emparejamiento fuerte consiguen que exista una correspondencia entre las tuplas.

```
Tupla del trabajo:             ["DEPLOYED",[...],[...],[...],[usuario, "null"] ]
Patrón del mediador Hermes:    ["DEPLOYED",?,?,?, [?, "hermes.cps.unizar.es" ] ]
Patrón del mediador Aragrid:   ["DEPLOYED",?,?,?, [?, "ui-ara.bifi.unizar.es"] ]
Patrón del mediador Piregrid:  ["DEPLOYED",?,?,?, [?, "ui-prg.bifi.unizar.es"] ]
Patrón del mediador de Amazon:["DEPLOYED",?,?,?, [?, "AmazonEC2"] ]
```

Figura 35: Ejemplo de tuplas utilizadas para el despliegue de trabajos

Para solucionar este aspecto se define una nueva política de emparejamiento competitivo utilizada solamente con tuplas que describen trabajos sin especificar el *grid* de ejecución. En este caso, el sistema permite el emparejamiento con los mediadores que sean capaces de ejecutar la aplicación indicada. Si existen varios mediadores capaces de ejecutar dicha aplicación, el *broker* selecciona uno de ellos de forma no determinista. El algoritmo en pseudocódigo utilizado para la política de emparejamiento competitivo aparece en la figura 36.

```
/* Devuelve cierto si se pueden emparejar tupla y patrón */
boolean emparejamientoCompetitivo(tupla, patrón) {
   // Obtenemos la palabra clave
   id_tupla = obtenerPalabraClave(tupla);
   id_patrón = obtenerPalabraClave(patrón);

   // Comprobamos si es una tupla de trabajo y el patrón puede emparejarse
   Si ( id_tupla == "DEPLOYED" AND id_patrón == "DEPLOYED") {
```

```
      // Obtenemos el entorno de ejecución
      grid = obtenerGrid(tupla);

      // Comprobamos si la tupla es guiada hacia algún Grid
      Si (grid == null) {
         // Si no es guiada obtenemos el grid del patrón y la aplicación
         // a ejecutar y el usuario que lo solicita de la tupla
         grid = obtenerGrid(patrón);
         aplicación = obtenerAplicacion(tupla);
         usuario = obtenerUsuario(tupla);

         // Hay emparejamiento si existe el grid definido en el patrón,
         // el usuario tiene permiso para ejecutar en el grid y el grid es
         // es capaz de ejecutar la aplicación
         devuelve existeGrid(grid) AND
                  existeUsuario(usuario, grid) AND
                  existeAplicacion(aplicación, grid);
      }
      si no {
         // Si la tupla es guiada utilizamos el emparejamiento débil
         devuelve emparejamientoDebil(tupla, patrón);
      }
   }
   si no {
      // Para el resto de lecturas utilizamos el emparejamiento débil
      devuelve emparejamientoDebil(tupla, patrón);
   }
}
```

Figura 36: Algoritmo para la política de emparejamiento competitivo.

V.2 - Identificación de los trabajos

Otro aspecto fundamental del que se encarga el *broker* tiene que ver con la identificación de los trabajos entre los diferentes componentes del sistema. Los trabajos se identifican externamente utilizando los ficheros de salida del mismo. Sin embargo, si bien no tiene sentido que existan dos tuplas con el mismo identificador ya que se estaría sobreescribiendo la salida de los trabajos, este identificador no asegura la unicidad del trabajo y si sucediese dicha situación podrían aparecer problemas. Internamente, es necesario asegurar que los identificadores de los trabajos son únicos porque un trabajo puede pasar por diferentes componentes y su tratamiento no debe mezclarse con el de otros trabajos.

Para solucionar esta limitación, el *broker* asigna un identificador único a cada trabajo cuando este es desplegado. La razón de que el *broker* sea el encargado de asignar los identificadores se debe a que es el componente central del sistema, el que es compartido por todos los componentes del sistema. El identificador utilizado consiste en un número entero que se va incrementando sucesivamente junto con la fecha en la que se despliega la tupla. Con esto conseguimos que el identificador sea único, aunque el *broker* sea reiniciado.

El identificador de la tupla se añade al final de la misma y es recuperado por el componente que lee la tupla de forma completamente transparente al usuario. Una vez en el componente, se almacena dicho identificador y éste se utiliza para identificar el resto de tuplas relacionadas con el trabajo. Esta característica es importante en ciertos componentes, por ejemplo, en el de gestión de fallos para conocer cuántas veces ha fallado un mismo trabajo.

La gestión de estos identificadores se realiza de forma completamente transparente para el usuario. Para ello, el identificador se almacena internamente en el tipo de dato *Tupla* utilizado en el sistema. Sin embargo, debido a la implementación del *broker* de mensajes, no es posible almacenar el identificador en el mismo. Para poder almacenar dicho identificador ha sido necesario modificar la API que permite interactuar con el *broker*.

Los cambios realizados afectan tanto a las operaciones de escritura como a las operaciones de lectura. En las operaciones de escritura se añade al final de la tupla el identificador del trabajo. Este identificador se obtiene internamente de la tupla que se quiere añadir o es generado por el *grid* si la tupla no tiene identificador. En las operaciones de lectura se añade un comodín al final de la tupla patrón para poder recuperar ese identificador. Una vez recuperado se almacena en la tupla obtenida y se elimina el último campo de la misma, que contiene el identificador, para que la gestión del mismo sea completamente transparente.

V.3 - Interfaz de programación

Para permitir la comunicación entre los diferentes elementos del sistema, el *broker* WS-PTRLinda ofrece una API o interfaz de programación que implementa las operaciones definidas por el modelo de coordinación Linda. De esta forma, se proporcionan operaciones para lectura destructiva, lectura no destructiva y escritura de tuplas en el espacio compartido. La semántica de dichas operaciones es la siguiente:

- `out(tupla)`. La operación `out` permite escribir la tupla indicada como parámetro en el repositorio de mensajes. En este caso, se trata de una operación no bloqueante.
- `in(patrón) devuelve tupla`. La operación `in` se trata de una operación de lectura destructiva. La operación recibe como parámetro una tupla patrón y devuelve como resultado una tupla que se empareje con el patrón indicado de acuerdo a la política de emparejamiento definida. Si existe más de una tupla que puede ser emparejada con el patrón, se devuelve una de ellas de forma no determinista. Si no existe ninguna tupla que pueda ser emparejada con el patrón, la operación queda bloqueada hasta que exista alguna tupla que pueda ser emparejada con el patrón. Una vez que es leída la tupla, ésta es eliminada del espacio compartido.
- `rd(patrón) devuelve tupla`. La operación `rd` implementa la operación de lectura no destructiva. Su funcionamiento es el mismo que el de la operación `in` con la diferencia de que la tupla leída no es eliminada del espacio de tuplas y puede ser consultada en posteriores operaciones.

Estas tres operaciones se ofrecen a través de un servicio Web que puede ser tanto SOAP como REST para facilitar la utilización del *broker* desde cualquier máquina conectada a través de la red.

Anexo VI - Componente de gestión de fallos

La aparición de fallos es un elemento habitual debido a la naturaleza de las infraestructuras de computación y a la complejidad de los *workflows científicos*. Estos fallos pueden deberse tanto a errores en el modelado del *workflow científico*, fallos propios de la aplicación que se está ejecutando o fallos provocados por la propia infraestructura.

Los errores que pueden producirse son de naturaleza muy distinta, pueden aparecer por una mala definición del trabajo que se quiere ejecutar, un error propio del trabajo que se está ejecutando o un fallo de la infraestructura, por ejemplo, si un nodo del *grid* falla, se producirá un fallo en todos los trabajos que se estén ejecutando en dicho nodo. Por tanto, la gestión de estos fallos es un aspecto crítico que no puede ser obviado y que hay que gestionar de una manera adecuada.

VI.1 - Arquitectura general del componente

En la figura 37, puede observarse el diseño del componente de gestión de fallos y su interacción con el *broker* de mensajes para recuperar las tuplas de error y escribir las tuplas correspondientes a la decisión tomada.

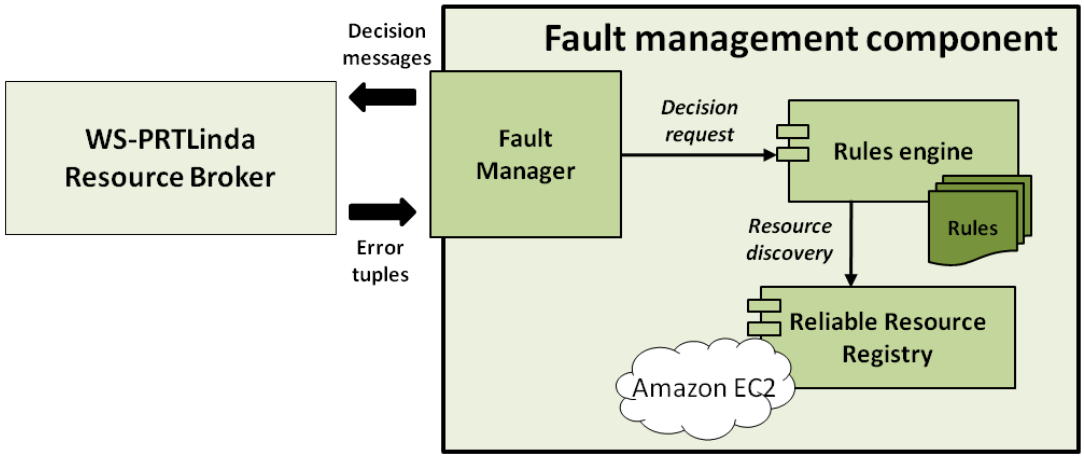

Figura 37: Arquitectura del componente de gestión de fallos

En primer lugar, de forma totalmente transparente a este componente, cuando la ejecución de un trabajo falla, el mediador correspondiente captura el fallo e introduce en el *broker* una tupla de error. Esta tupla se identifica por la palabra clave `ERROR` y contiene tanto la información relativa al trabajo que falló como la causa del fallo. El gestor de fallos (*Fault Manager*) es el encargado de recuperar dicha tupla, procesarla, creando para ello un nuevo hilo de ejecución, y solicitar al motor de reglas que tome una decisión sobre el trabajo.

Por su parte, el motor de reglas (*Rules engine*) toma una decisión teniendo en cuenta una serie de reglas básicas (base de conocimiento) y si el trabajo había fallado ya antes. Para definir estas reglas, hemos utilizado RuleML [W21] como lenguaje de reglas ya que se trata del lenguaje estándar para reglas en entornos Web. Las decisiones que puede tomar pueden ser: abortar la ejecución del trabajo, reiniciar el trabajo en el mismo *grid* o en otro diferente, reiniciar el mismo utilizando un *grid* confiable o continuar la ejecución de forma normal.

Este último caso contempla la ejecución del trabajo en un tipo especial de recurso que es catalogado como fiable. No hay que olvidar, que nuestro objetivo es ejecutar de forma satisfactoria los diferentes trabajos solicitados, por lo que la utilización de estos recursos ayudan a lograr dicho objetivo. Para ello se incluye un Registro de Recursos Fiable (*Reliable Resource Registry*) en el que se guarda una lista de *grids* fiables. Es aquí donde proponemos la utilización de alguna infraestructura de computación externa fiable como el servicio de computación *Cloud* de Amazon, *Amazon Elastic Compute Clod* (Amazon EC2) [W9]. En cualquier caso, el motor de reglas es el encargado de decidir en cuál de los recursos fiables debe ser ejecutado el trabajo, devolviendo esta información al gestor de fallos.

Independientemente, de si se reinicia el trabajo en un *grid* fiable o se toma otra decisión, el gestor de fallos recupera esa decisión e introduce una nueva tupla en el *broker* de mensajes indicando la misma. Esta tupla será capturada por el mediador correspondiente si la decisión es reiniciar el trabajo o por el motor encargado de la ejecución del *workflow* si la acción tomado es la de abortar el trabajo o continuar con la ejecución.

Para ilustrar el funcionamiento del componente, se incluye la figura 38. En dicha figura, se muestra la ejecución del componente en el caso de que sea necesario reiniciar la ejecución del trabajo en un *grid* confiable. En caso de que la decisión tomada por el motor de reglas sea otra diferente, el comportamiento es análogo eliminando la interacción con el servicio de descubrimiento de *grids* fiables.

En primer lugar, el *Fault Manager* obtiene una tupla del broker WS-PTRLinda y crea un nuevo hilo para gestionar la tupla (denominado *Fault Handler*). A continuación se obtiene el identificador del trabajo y los fallos que se han producido, solicitándose al motor de reglas que tome una decisión para cada fallo. Para ello, el motor añade el fallo al sistema y razona utilizando las reglas definidas para tomar una decisión. En caso de que la decisión sea reiniciar el trabajo en un grid fiable, se solicita al servicio de descubrimiento todos los grids fiables. Una vez obtenida la lista, el motor indica la decisión al gestor de fallos seleccionando uno de los grids de forma no determinista. Cuando se han tratado todos los fallos se construye la tupla asociada a la decisión generada y se coloca la misma en el *broker*.

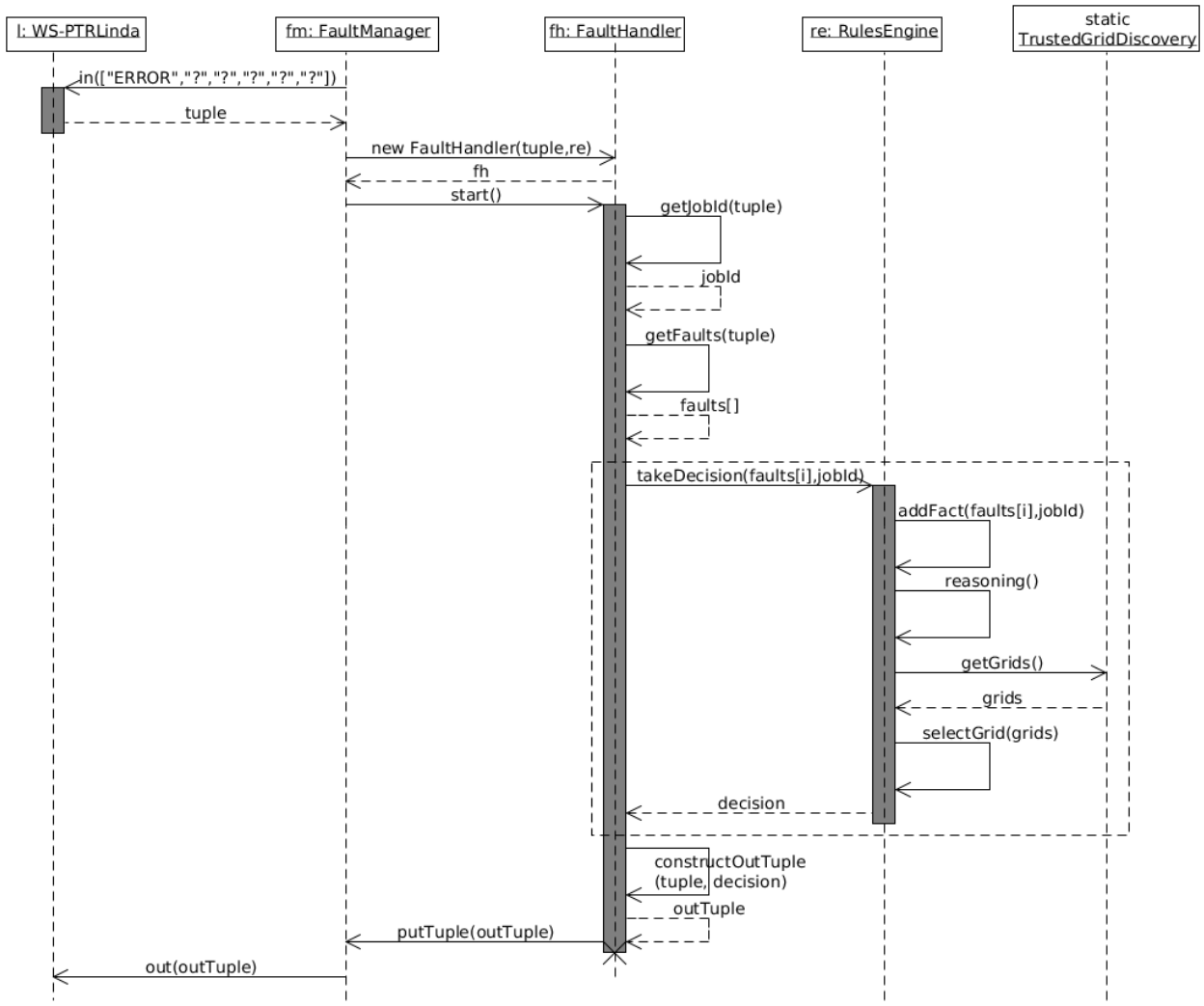

Figura 38: Traza de eventos del componente de gestión de fallos

VI.2 - Tipos de errores y acciones correctoras

La aparición de fallos puede deberse a diferentes causas, la aparición de un fallo durante la ejecución de la tarea, un fallo durante el movimiento de información necesario (a la entrada o a la salida) de la tarea o un fallo en la definición del trabajo. En la figura 39, se puede observar una relación de todos los fallos que pueden aparecer al ejecutar las diferentes tareas que componen un *workflow* científico.

```
ERROR(1) : Malformed tuple.
ERROR(2) : Malformed scp URI $(uri). Impossible to move the file
ERROR(3) : The string $(file) can not be converted to an URI
ERROR(4) : Application can not be null.
ERROR(5) : Impossible to replace wildcards in $(path)
ERROR(6) : Execution failed. $(Execution_error)
ERROR(7) : Download/Upload from $(uri) failed. Connection error. $(error)
ERROR(8) : Download/Upload from $(uri) failed. Data movement error. $(error)
ERROR(9) : Job deployment failed. $(error)
ERROR(10): Undefined email for user $(user)
ERROR(11): User for $(global_user) undefinied on grid $(host)
ERROR(12): Grid $(host) undefinied
ERROR(13): Application $(appName) undefined on $(host)
```

```
ERROR(14): Output data retrieval failed. $(error)
ERROR(15): Movement from $(source) to $(destination) failed. $(Error)
ERROR(16): Job execution error. $(status)
ERROR(17): Invalid URI $(uri)
ERROR(18): Impossible to retrieve log. $(error)
```

Figura 39: Tipos de errores identificados en el sistema

Las acciones correctoras propuestas para los errores anteriores aparecen en la tabla 4. Se han definido cuatro tipos de acciones correctoras, dependiendo del tipo de fallo:

- La acción más restrictiva consiste en abortar la ejecución del trabajo. Este tipo de acción se toma cuando se está seguro de que el reinicio del trabajo provocará un nuevo fallo, por ejemplo, si la tarea ha sido mal definida. En este caso, se notifica al usuario del fallo para que corrija el error cometido.
- Otra posibilidad es reiniciar el trabajo. Este tipo de acción se realiza cuando existe la posibilidad de que reiniciar el trabajo puede solucionar el problema. Esta política es frecuentemente utilizada en sistemas distribuidos y permite solventar problemas de disponibilidad de los recursos, principalmente. Dentro de esta acción correctora se establecen tres posibilidades:
  1. Si es la primera vez que falla esa tarea, se reinicia la misma en los recursos habituales.
  2. Si se produce por segunda vez el mismo fallo, se reinicia el trabajo en un *grid* fiable para aumentar las posibilidades de que el mismo se ejecute con éxito.
  3. Si el trabajo falla por tercera vez, se considera que existe un error grave y se aborta el trabajo en lugar de reiniciarlo.error grave que impide la ejecución del mismo.
- En caso de que el fallo se haya producido a la hora de recuperar los ficheros de salida, se permite continuar normalmente la ejecución indicando en el *log* que se ha producido dicho fallo. A pesar de que es posible que este fallo provoque un fallo en sucesivas tareas, un error al mover la salida no justifica volver a ejecutar el trabajo de forma completa, ya que éste puede ser muy costoso, por lo que es más sencillo y útil que el usuario realice el movimiento de dicha salida manualmente.
- La última acción correctora propuesta es similar a la anterior y se utiliza en el caso de que no se pueda recuperar el *log* de la tarea ejecutada. En este caso, se decide continuar con la ejecución normal del *workflow* indicando en el *log* que no se ha podido recuperar el mismo.

Número de fallo	Acción correctora
1, 2, 3, 4, 10, 11, 12, 13, 17	Abort
5, 6, 7, 8, 9, 16	Restart
14, 15	Continue without output
18	Continue without log

Tabla 4: Acciones correctoras para los diferentes tipos de errores.

VI.3 - Gestor de fallos

El gestor de fallos es el componente encargado de la interacción con el *broker* de mensajes Linda utilizando la interfaz que este proporciona. De esta forma, una vez inicializado el gestor de fallos realiza una petición de tupla de error al *broker* de recursos, quedándose bloqueado hasta que exista alguna en el mismo. Para identificar las tuplas de error se utiliza la palabra clave `ERROR`. Asimismo, este tipo de tuplas esta formado, además de por el identificador de tupla `ERROR`, por la información del trabajo que generó el fallo seguida de una cadena que indica el motivo del fallo. En la figura 40, puede observarse el formato de una tupla de estas características mientras que en la figura 41 puede observarse la tupla patrón utilizada para capturar la misma.

```
[
  ERROR,
  ["aplicación", "argumentos", "fichero_entrada1 fichero_entrada2",
"fichero_salida1 fichero_salida2"],
  ["entrada_estándar", "salida_estándar", "salida_error"],
  ["fecha_entrega","presupuesto"],
  ["usuario", "grid"],
  "mensajes_de_error",
]
```

Figura 40: Formato de una tupla de error

```
["ERROR", "?", "?", "?", "?", "?" ]
```

Figura 41: patrón utilizado por el Gestor de Fallos.

Una vez obtenida una tupla de error, se obtienen los mensajes de error que contiene la misma (pueden ser varios) y se separan los mismos descartando los mensajes de error repetidos. Una vez realizado este proceso, se tratan cada uno de los mensajes por separado, enviando los mismos al motor de reglas y seleccionando la acción más restrictiva. Para lograr una mayor eficiencia, se dejan de tratar los mensajes cuando, como respuesta a alguno de los fallos, el motor de reglas indica que hay que abortar la ejecución de la tarea. El pseudoalgoritmo utilizado puede observarse en la figura 42.

```
// Leer tupla del Broker
tupla_fallo = Broker.in();

//Conseguir identificar de la tarea y obtener los fallos asociados
id_tarea = obtenerIdentificador(tupla_fallo);
fallos[] = obtenerFallos(tupla_fallo);

// Gestionamos los fallos individualmente
abortar = falso;          // Chequeamos si la acción más restrictiva es abortar
reiniciar = falso;          // Chequeamos si la acción más restrictiva es
reiniciar
reiniciar_fiable = falso; // Chequeamos si la acción más restrictiva es
reiniciar en un grid confiable
contador = 0;

Repetir {
   // ¿Ha sido ya tratado un fallo con el mismo número?
   Si(NOT haSidoTratado(fallos[contador])    {
      // Solicitar decisión al motor de reglas
      acciones[contador] = MotorReglas.decidir(fallos[contador], id_tarea);
```

```
    //Obtenemos el numero de veces que ha fallado esa tarea en total
    numero_veces = obtenerNumeroFallos(acciones[contador];

    // Creamos la tupla adecuada dependiendo de la acción indicada
    Si(acciones[contador] == "Restart") {
       Si(NOT reiniciar_fiable) {
          tupla_salida = crearTuplaReiniciar(tupla_fallo);
          reiniciar = verdad;
       }
    }
    sino si (acciones[contador] == "Abort") {
       tupla_salida = crearTuplaAbortar(tupla_fallo);
       abortar = verdad;
    }
    sino si (acciones[contador] == "Continue without output") {
       Si(NOT reiniciar AND NOT reiniciar_fiable) {
          tupla_salida = crearTuplaContinuarSinSalida(tupla_fallo);
       }
    }
    sino si (acciones[contador] == "Continue without log") {
       Si(NOT reiniciar AND NOT reiniciar_fiable) {
          tupla_salida = crearTuplaContinuarSinLog(tupla_fallo);
       }
    }
    sino si (acciones[contador].empieza por("Trusted Restart")) {
       grid = obtenerGridFiable(acciones[contador]);
       tupla_salida = crearTuplaReinicarFiable(grid);
       reiniciar_fiable = verdad;
    }
    sino { // ACCION DESCONOCIDA
       tupla_salida = crearTuplaAbortar(tupla_original);
       abortar = verdad;
    }
    contador++; //Incrementar contador
 }mientras_que(NOT abortar AND contador < obtenerNumeroFallos(fallos));

 // Escribimos la tupla de salida en el broker
 Broker.out(tupla_salida);
```

Figura 42: Algoritmo utilizado por el Gestor de Fallos.

VI.4 - Motor de reglas

El motor de reglas es el componente encargado de decidir que acción se debe realizar cuando se produce un fallo en la ejecución de una tarea. El lenguaje utilizado para la definición de reglas es RuleML (*Rule Markup Language*). La decisión de utilizar este lenguaje se debe a que se trata del lenguaje de reglas estándar en entornos web y a que proporciona una sencilla pero potente sintaxis para describir las reglas.

En lo que corresponde al motor de reglas, en un primer momento, se pensó en realizar una sencilla implementación del mismo para ilustrar el funcionamiento de dicho componente. Sin embargo, finalmente se descartó esta idea y se ha utilizado como motor de reglas OOjDREW [W41], la implementación de referencia para gestionar reglas en el formato definido por RuleML. Esta decisión se debe a que la utilización de un motor de reglas real nos permite dotar al componente de gestión de fallos de una mayor potencia y expresividad. Asimismo, conseguimos una mayor flexibilidad al poder cambiar las reglas establecidas en cualquier momento, facilitando la ampliación y mejora del componente.

En lo que respecta al funcionamiento y puesta en marcha del motor de reglas, en primer lugar hay que definir una serie de reglas que se conocen como base de conocimiento y que son cargadas al crear el motor. Estas reglas son las reglas que definen las acciones correctoras que se deben tomar para los diferentes fallos. Las mismas se definen en un fichero de configuración denominado `rules.ruleml` en el directorio de configuración del sistema (directorio `config`). Dicho fichero puede observarse en la figura 43.

```
<Assert>
   <Rulebase mapClosure="universal">
      <!-- FAULT NUMBER 1 -->
      <Implies>
         <Atom>
            <Rel>FAULT</Rel>
            <Var>job</Var>
            <Ind>1</Ind>
         </Atom>
         <Atom>
            <Rel>ACTION</Rel>
            <Var>job</Var>
            <Ind>Restart</Ind>
         </Atom>
      </Implies>
      <!-- FAULT NUMBER 2 -->
      <Implies>
         <Atom>
            <Rel>FAULT</Rel>
            <Var>job</Var>
            <Ind>2</Ind>
         </Atom>
         <Atom>
            <Rel>ACTION</Rel>
            <Var>job</Var>
            <Ind>Restart</Ind>
         </Atom>
      </Implies>
      <!-- FAULT NUMBER 3 -->
      <Implies>
         <Atom>
            <Rel>FAULT</Rel>
            <Var>job</Var>
            <Ind>3</Ind>
         </Atom>
         <Atom>
            <Rel>ACTION</Rel>
            <Var>job</Var>
            <Ind>Abort</Ind>
         </Atom>
      </Implies>
      <!-- FAULT NUMBER 4 -->
      <Implies>
         <Atom>
            <Rel>FAULT</Rel>
            <Var>job</Var>
            <Ind>4</Ind>
         </Atom>
         <Atom>
            <Rel>ACTION</Rel>
            <Var>job</Var>
            <Ind>Abort</Ind>
         </Atom>
      </Implies>
```

```
<!-- FAULT NUMBER 5 -->
<Implies>
   <Atom>
      <Rel>FAULT</Rel>
      <Var>job</Var>
      <Ind>5</Ind>
   </Atom>
   <Atom>
      <Rel>ACTION</Rel>
      <Var>job</Var>
      <Ind>Restart</Ind>
   </Atom>
</Implies>
<!-- FAULT NUMBER 6 -->
<Implies>
   <Atom>
      <Rel>FAULT</Rel>
      <Var>job</Var>
      <Ind>6</Ind>
   </Atom>
   <Atom>
      <Rel>ACTION</Rel>
      <Var>job</Var>
      <Ind>Restart</Ind>
   </Atom>
</Implies>
<!-- FAULT NUMBER 7 -->
<Implies>
   <Atom>
      <Rel>FAULT</Rel>
      <Var>job</Var>
      <Ind>7</Ind>
   </Atom>
   <Atom>
      <Rel>ACTION</Rel>
      <Var>job</Var>
      <Ind>Restart</Ind>
   </Atom>
</Implies>
<!-- FAULT NUMBER 8 -->
<Implies>
   <Atom>
      <Rel>FAULT</Rel>
      <Var>job</Var>
      <Ind>8</Ind>
   </Atom>
   <Atom>
      <Rel>ACTION</Rel>
      <Var>job</Var>
      <Ind>Restart</Ind>
   </Atom>
</Implies>
<!-- FAULT NUMBER 9 -->
<Implies>
   <Atom>
      <Rel>FAULT</Rel>
      <Var>job</Var>
      <Ind>9</Ind>
   </Atom>
   <Atom>
      <Rel>ACTION</Rel>
      <Var>job</Var>
      <Ind>Restart</Ind>
```

```
      </Atom>
   </Implies>
   <!-- FAULT NUMBER 10 -->
   <Implies>
      <Atom>
         <Rel>FAULT</Rel>
         <Var>job</Var>
         <Ind>10</Ind>
      </Atom>
      <Atom>
         <Rel>ACTION</Rel>
         <Var>job</Var>
         <Ind>Abort</Ind>
      </Atom>
   </Implies>
   <!-- FAULT NUMBER 11 -->
   <Implies>
      <Atom>
         <Rel>FAULT</Rel>
         <Var>job</Var>
         <Ind>11</Ind>
      </Atom>
      <Atom>
         <Rel>ACTION</Rel>
         <Var>job</Var>
         <Ind>Abort</Ind>
      </Atom>
   </Implies>
   <!-- FAULT NUMBER 12 -->
   <Implies>
      <Atom>
         <Rel>FAULT</Rel>
         <Var>job</Var>
         <Ind>12</Ind>
      </Atom>
      <Atom>
         <Rel>ACTION</Rel>
         <Var>job</Var>
         <Ind>Abort</Ind>
      </Atom>
   </Implies>
   <!-- FAULT NUMBER 13 -->
   <Implies>
      <Atom>
         <Rel>FAULT</Rel>
         <Var>job</Var>
         <Ind>13</Ind>
      </Atom>
      <Atom>
         <Rel>ACTION</Rel>
         <Var>job</Var>
         <Ind>Abort</Ind>
      </Atom>
   </Implies>
   <!-- FAULT NUMBER 14 -->
   <Implies>
      <Atom>
         <Rel>FAULT</Rel>
         <Var>job</Var>
         <Ind>14</Ind>
      </Atom>
      <Atom>
         <Rel>ACTION</Rel>
```

```
            <Var>job</Var>
            <Ind>Continue without output</Ind>
         </Atom>
      </Implies>
      <!-- FAULT NUMBER 15 -->
      <Implies>
         <Atom>
            <Rel>FAULT</Rel>
            <Var>job</Var>
            <Ind>Continue without output</Ind>
         </Atom>
         <Atom>
            <Rel>ACTION</Rel>
            <Var>job</Var>
            <Ind>Restart</Ind>
         </Atom>
      </Implies>
      <!-- FAULT NUMBER 16 -->
      <Implies>
         <Atom>
            <Rel>FAULT</Rel>
            <Var>job</Var>
            <Ind>16</Ind>
         </Atom>
         <Atom>
            <Rel>ACTION</Rel>
            <Var>job</Var>
            <Ind>Restart</Ind>
         </Atom>
      </Implies>
      <!-- FAULT NUMBER 17 -->
      <Implies>
         <Atom>
            <Rel>FAULT</Rel>
            <Var>job</Var>
            <Ind>17</Ind>
         </Atom>
         <Atom>
            <Rel>ACTION</Rel>
            <Var>job</Var>
            <Ind>Abort</Ind>
         </Atom>
      </Implies>
      <!-- FAULT NUMBER 18 -->
      <Implies>
         <Atom>
            <Rel>FAULT</Rel>
            <Var>job</Var>
            <Ind>18</Ind>
         </Atom>
         <Atom>
            <Rel>ACTION</Rel>
            <Var>job</Var>
            <Ind>Continue without log</Ind>
         </Atom>
      </Implies>
   </Rulebase>
</Assert>
```

Figura 43: Base de conocimiento para el motor de reglas.

Cuando el gestor de fallos indica al motor de reglas que se ha producido un fallo, se introduce una nueva regla en el motor y se realiza una pregunta al mismo para determinar la acción correctora que debe realizarse que dependerá del número de veces que haya fallado ese mismo trabajo como se ha comentado anteriormente. De la misma manera, en el caso de que la decisión sea reiniciar el trabajo en un *grid* fiable, el motor de reglas contacta con el servicio de descubrimiento de *grids* fiables para obtener un recurso en el que ejecutar el trabajo. En el caso de que no exista ningún recurso fiable, se reinicia la tarea de forma normal. En la figura 44, se proporciona el pseudoalgoritmo para la toma de decisiones realizada por el motor de reglas.

```
tomarDecision(mensaje_fallo, identificar_tarea) devuelve accion_correctora
{
   // Obtener número de fallo
   número = obtenerNumeroFallo(mensaje_fallo)

   // Construir nuevo hecho y añadirlo al motor de reglas
   hecho = construirHecho(número);
   motorOOjDREW.AñadirHecho(hecho);

   // Construir consulta para determinar accion correctora
   consulta = construirConsulta(identificador_tarea);
   solucion = motorOOjDREW.ejecutar(consulta);

   // Obtener la accion correcta y el numero de fallos anteriores
   acción_correctora = obtenerAcciónCorrectora(solucion);
   numero_fallos = obtenerNumeroFallosPrevios(solucion);

   //Comprobamos si hay que modificar la accion correctora
   Si (acción_correctora == "Restart")
   {
      elegir(numero_fallos)
      {
         1 --> acción_correctora = "Restart";
         2 --> grids[] = TrustedGridDiscovery.getGrids();
               Si(HayAlgunGrid(grids))
               {
                  grid = seleccionarGrid(grids);
                  acción_correctora = "Trusted Restart (" + grid + ")";
               }
               si no
               {
                  acción_correctora = "Restart";
               }
         3 --> acción_correctora = "Abort";
   }

   devuelve acción_correctora;
```

Figura 44: Pseudoalgoritmo para la toma de decisiones por parte del motor de reglas

VI.5 - Registro de recursos fiables

El registro de recursos fiables se utiliza para almacenar los recursos fiables en los que pueden ejecutarse los diferentes trabajos en caso de fallo. La implementación del mismo se ha realizado de la misma manera que el resto de registros del sistema.

Más detalles acera de su implementación pueden ser consultados en el *Anexo IV* –

Registros del sistema.

VI.6 - Diagrama de clases

En esta última sección, se incluye la figura 45 en la que se muestra el diagrama de clases del componente.

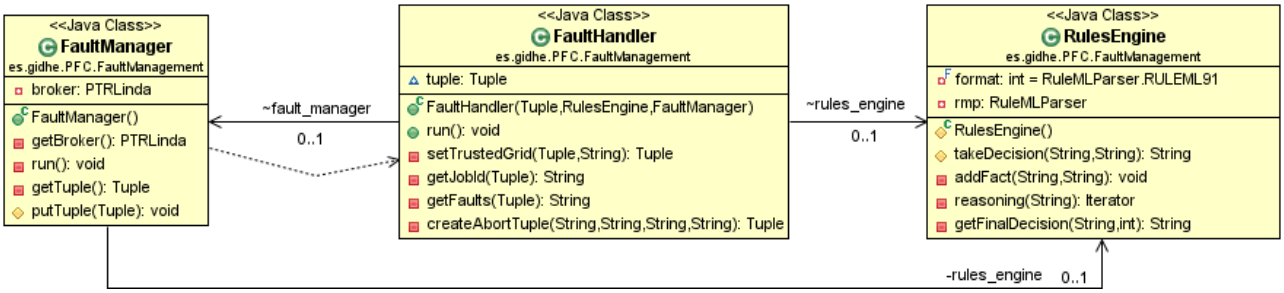

Figura 45: Diagrama de clases del componente de gestión de fallos.

Anexo VII - Componente de movimiento de datos

Uno de los aspectos más importantes de los que se encarga un *middleware* es de realizar el movimiento de los datos entre los diferentes recursos de un *grid*. La necesidad de este movimiento viene determinada por la abstracción de que todo el *grid* es un único supercomputador. Para lograr la misma, el movimiento de los datos necesarios para ejecutar un trabajo desde su lugar de origen hasta el nodo en el que se ejecuta el mismo debe ser completamente automático y transparente para el usuario.

El auge de las infraestructuras de computación *grid* ha propiciado la aparición de protocolos de transferencia de ficheros especializados. Este es el caso de GridFTP [B17] que rápidamente se ha establecido como estándar para el movimiento de datos dentro del *grid*.

En nuestro caso, el problema cambia significativamente. El sistema desarrollado no pretende sustituir al *middleware* utilizado en el *grid*, si no englobar diferentes *middlewares* para integrar varios *grids*. Esto nos permite aprovechar las características de los *middlewares* como es el caso de la gestión de los datos dentro de los recursos del *grid*. Por tanto, nuestro problema se refiere al movimiento de datos entre diferentes *grids* de forma automática. En este anexo vamos a explicar la problemática y a analizar la solución alcanzada.

VII.1 - Descripción del problema y solución utilizada

La inclusión de diferentes *grids* bajo un mismo sistema plantea el problema del movimiento de datos entre los mismos, el cual se dificulta porque el usuario no conoce a priori en qué *grid* va a ser ejecutado su trabajo. Además, no se puede conocer la infraestructura de ejecución hasta el momento exacto en el que se va a ejecutar el trabajo, complicando enormemente el movimiento de los datos.

La primera cuestión que era necesaria resolver antes de definir cómo se iba a realizar el movimiento de datos era la identificación de los mismos. Se valoraron tres alternativas:

- Identificar los datos como si fueran datos locales. Esta primera alternativa tenía como objetivo identificar los datos sin tener en cuenta los recursos, permitiendo que los mismos se ubicaran en cualquiera de los *grids* definidos. El principal beneficio que presenta esta alternativa es que abstrae al usuario de definir el *grid* en el que se encuentra un dato. Por contra, esta alternativa implica que el sistema tenga que buscar el dato en todos los *grids* usados, con el aumento de complejidad y la pérdida de eficiencia consecuente, y presenta el problema de que no es posible saber qué fichero es el correcto si existen ficheros con el mismo nombre en diferentes *grids*. Este último aspecto hizo que se descartara la idea.
- La segunda alternativa consistía en especificar, con algún tipo de lenguaje propio, tanto el propio dato como el recurso en el que se encuentra el mismo. Sin embargo, esta alternativa fue descartada porque uno de los objetivos del proyecto es facilitar la compartición de *workflows* científicos por lo que la utilización de un lenguaje propio no era una alternativa viable.
- La tercera alternativa planteada, y que fue elegida finalmente, fue la utilización de URIs [W23] para identificar de forma unívoca los datos. La utilización de URIs, es una forma

estándar de identificar datos en entornos distribuidos y heterogéneos por lo que representa la mejor alternativa para nuestro propósito. Además, esta alternativa nos da la posibilidad de utilizar datos que se encuentren en servidores externos de cualquier tipo con lo que no limitamos los datos que pueden ser utilizados a datos que se encuentren en los Grids definidos.

La utilización de URIs planteaba el problema de que el usuario debe ser consciente del protocolo de transferencia que debe ser usado para transferir el fichero. Para facilitar este aspecto se introducen dos características. En primer lugar, para el caso de que el usuario esté seguro de que los ficheros se encuentran en el *grid* en el que se va a ejecutar una tarea, se permite utilizar simplemente la ruta del fichero. En segundo lugar, se permite identificar un fichero con el esquema o protocolo `file`. La utilización de este formato permite al usuario no indicar el usuario del *grid* y el protocolo de transferencia, siendo estos recuperados del registro de usuarios y del registro de *grids* del sistema (para más información sobre los registros del sistema consultar el *Anexo IV - Registros del sistema*). Un ejemplo de cómo identificar un fichero con estos dos mecanismos puede observarse en la figura 46.

```
a) file://hermes.cps.unizar.es/~/ejemplo.txt
b) sftp://shernandez@hermes.cps.unizar.es/~/ejemplo.txt
```

Figura 46: Identificación de un fichero. **a)** Identificación con URI 'file'. **b)** Identificación con URI 'sftp'.

Asimismo, se permite la utilización de comodines para definir la ruta de un fichero o grupo de ficheros dentro de un *grid*. Por ejemplo, en la figura 1 se ha utilizado el carácter '~' para definir que el fichero 'ejemplo.txt' se encuentra en el *home* del usuario. Esta característica permite definir de una forma sencilla y potente un grupo de ficheros utilizando solamente una URI. Finalmente, en la figura 47 se incluye el árbol de decisión utilizado para definir el comportamiento que debe ser seguido ante las diferentes descripciones posibles.

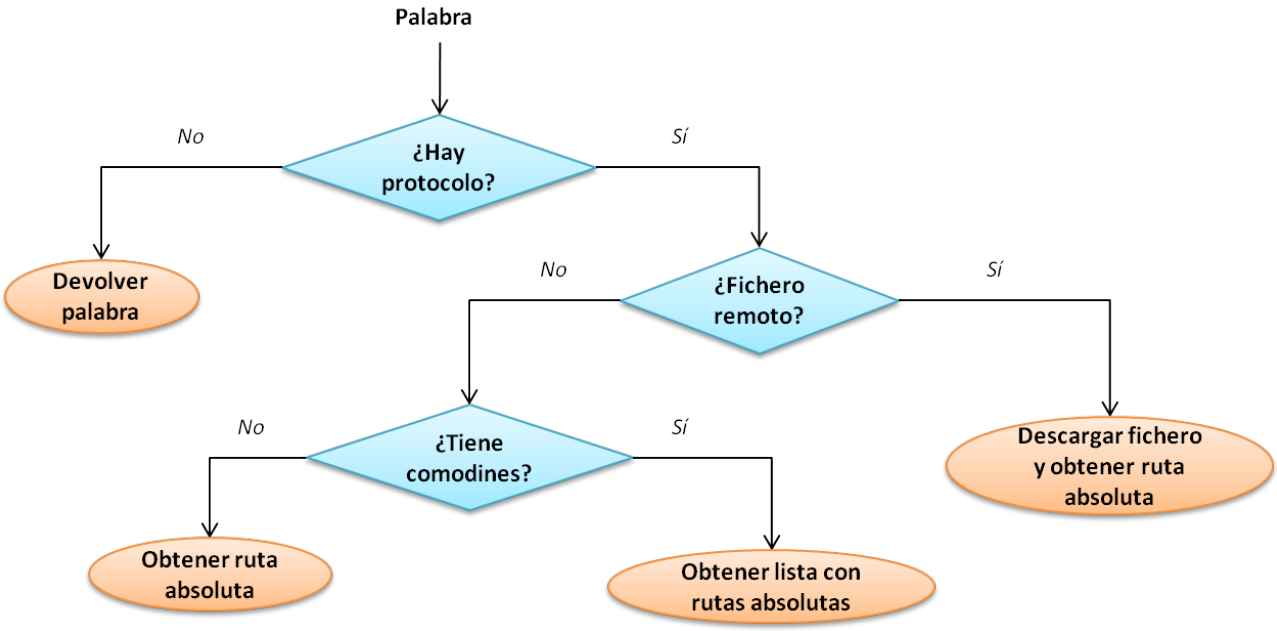

Figura 47: Árbol de decisión para las URIs

Cuando se parsea una palabra, se comprueba en primer lugar que la palabra corresponde efectivamente a una URI. Si no es así, la palabra puede hacer referencia a un argumento o parámetro de la aplicación o a una ruta relativa y no se realiza ninguna modificación sobre la misma al no tener certeza de dicha circunstancia. Si la palabra se trata de una URI, se detecta si la misma corresponde a un fichero local o a un fichero remoto. En el caso de que se trate de un fichero remoto, se descarga el mismo y se obtiene la ruta absoluta del fichero dentro del *grid* en el que se ha descargado el fichero. Si por contra se trata de un fichero local, se obtiene simplemente la ruta absoluta del mismo. Sin embargo, en este último caso, la palabra podría tener algún comodín y referirse a una lista de ficheros por lo que hay que detectar esta situación y, en el caso de que exista algún comodín, reemplazar el mismo por la lista de ficheros correspondiente.

VII.2 - Descripción del componente

En lo que corresponde a la implementación del componente de movimiento de datos, para facilitar su utilización por parte de los mediadores, se implementa el mismo como una única clase con métodos estáticos. Una de las características comunes que presentan dichos métodos es la utilización de conexiones SSH para garantizar la seguridad en las transferencias realizadas.

El componente ofrece una interfaz de programación de aplicaciones(*Application Programming Interface*, API) que permite realizar transferencias entre Grids, descargar ficheros de servicios externos, mover datos simples dentro del mismo *grid* y reemplazar comodines de las URIs. Para realizar estas operaciones se ofrecen los siguientes métodos

1. `String[] parseWord(word, global_user, host, ssh_session, download)`. Esta operación parsea la palabra `word` que puede ser una URI o una ruta local para determinar si es necesario realizar la descarga del fichero y obtener la ruta o rutas locales asociadas a la misma. El parámetro `global_user` indica el nombre del usuario y el parámetro `host` indica el *host* en el que se debe descargar el fichero. En ese caso, se utiliza la sesión SSH `ssh_session` para realizar la descarga. Finalmente, el parámetro booleano `download` indica si se debe descargar el fichero o ficheros asociados. En general, el fichero debe ser descargado y el parámetro sólo debe ser falso cuando se está seguro de que el fichero ya se ha descargado previamente. Como resultado, la operación devuelve un vector de dos componentes. En la primera componente se indica el resultado de parsear la palabra (la ruta local asociada) que puede ser una única palabra o una lista de palabras separadas por espacio, tabulador o salto de línea. En la segunda componente se indica un mensaje de error si existe o una cadena vacía si no hay error.
2. `String[] download(uri, global_user, destination_path, ssh_session)`. Esta operación permite descargar un fichero desde una URI externa a la ruta destino indicada por el parámetro `destination_path` utilizando para ello una sesión SSH. Además, para realizar la operación debe indicarse el usuario que realiza la misma a través del parámetro `global_user`. Como resultado, la operación devuelve un vector en el que indica en su primera componente el nombre del fichero descargado y en

la segunda componente un mensaje de error en el caso de que se produzca el mismo o una cadena vacía si no hay error.

3. `String upload(uri, global_user, source_path, ssh_session)`. Esta operación permite mover un fichero desde la ruta de origen `source_path` a la ubicación indicada en la URI. Para realizar esta transferencia se utiliza una conexión SSH. Asimismo, debe indicarse el usuario que realiza la misma en el parámetro `global_user`. Como resultado, el método devuelve una cadena con un mensaje de error si este se produce o una cadena vacía si la operación tiene éxito.
4. `String move(source, destination, ssh_session)`. Este método mueve un fichero desde la ruta indicada por el parámetro `source` hasta la ruta indicada por el parámetro `destination` utilizando para ello una sesión SSH. Como resultado, el método devuelve una cadena con un mensaje de error si este se produce o una cadena vacía si la operación tiene éxito.
5. `boolean hasWildcards(word)`. La función devuelve cierto si el parámetro `word` contiene algún comodín y falso en otro caso.
6. `String[] replaceWildcards(path, ssh_session)`. La operación permite reemplazar los comodines existentes en el parámetro `path` utilizando la sesión SSH definida en el segundo parámetro. Como resultado, la operación devuelve un vector en el que indica en su primera componente el resultado de reemplazar los comodines y en la segunda componente un mensaje de error en el caso de que se produzca el mismo o una cadena vacía si no hay error.
7. `String[] logRetrieval(command, ssh_session)`. Este método utiliza una conexión SSH para ejecutar el comando definido por el parámetro `command` que permite recuperar el *log* de ejecución de un trabajo. Como resultado, la operación devuelve un vector en el que indica en su primera componente el resultado del comando (la ejecución del *log*) y en la segunda componente un mensaje de error en el caso de que se produzca el mismo o una cadena vacía si no hay error.

Las operaciones `download` y `upload` que permiten el movimiento de datos entre diferentes infraestructuras soportan los siguientes protocolos de transferencia:

- Para la operación `download` se soportan: HTTP, HTTPS, FTP, SFTP, SCP, FILE
- Para la operación `upload` se soportan:FTP, SFTP, SCP, FILE

Asimismo, estas operaciones presentan la limitación de que para permitir la transferencia entre las diferentes infraestructuras, el usuario debe haber definido previamente la posibilidad de realizar conexiones SSH entre las diferentes infraestructuras sin la necesidad de utilizar contraseña, utilizando un mecanismo de autenticación con clave pública.

El método `parseWord` implementa el árbol de decisión mostrado en la figura 44. La implementación en pseudocódigo de dicho árbol de decisión puede observarse a continuación en la figura 48.

```
String[] parseWord(word, global_user, host, ssh_session, download) {
   //Inicializar resultado
   resultado[0] = ""; resultado[1] = "";

   // Obtener usuario
   user = userDiscovery.getUser(global_user);

   // Construimos una URI y tenemos el protocolo y el host remoto de la palabra
   uri = construirURI(word);
   protocolo  = obtenerEsquema(uri);
   host_remoto = obtenerHostRemoto(uri);

   Si (protocolo == null) { // Es un argumento, parámetro o ruta local
      resultado[0] = word;
   }
   sino si(sonDistintos(host, host_remoto)) { // FICHERO REMOTO
      // Comprobamos si hay que descargar el fichero
      Si(download == true) {
         // Descargar el fichero al home del usuario
         resultado[] = download(uri,global_user,obtenerHome(user),
ssh_session);
      }
      si no {
         // Devolver la ruta absoluta para el fichero
         fichero = obtenerNombreFichero(word);
         resultado[0] = rutaAbsoluta(user,fichero);
      }
   }
   si no { // FICHERO LOCAL
      //Obtener ruta del fichero sustituyendo el path
      ruta = obtenerRuta(user, uri)

      //Chequear si la palabra tiene comodines
      Si(hasWildcards(ruta)) { // Hay comodines
         //Reemplazar los comodines
         aux[] = replaceWildcards(path, ssh_session);
         lista[] = obtenerLista(aux[0]); // Usando separadores ' ', '\t' y '\n'
         Para j desde 0 hasta longitud(lista)-1 hacer {
            resultado[0] = resultado[0] + " " + rutaAbsoluta(user, lista[j]);
         }
      }
      si no { // No hay comodines
         resultado[0] = rutaAbsoluta(user, ruta);
      }
   }

   devolver resultado;
}
```

Figura 48: Algoritmo para la operación `parseWord`

Finalmente, en la figura 49, se muestra el pseudoalgoritmo de la operación `download` que es la que más complejidad reviste. En el mismo no se entran en los detalles referentes a todas las condiciones de error que son detectadas. El pseudoalgoritmo para la operación `upload` es análogo.

```
String[] download(uri, global_user, destination_path, ssh_session) {
   //Inicializar resultado
   resultado[0] = ""; resultado[1] = "";

   // Obtenemos el protocolo
   protocolo = obtenerProtocolo(uri);

   // Gestionamos cada uno de los protocolos soportados
   Si(protocolo == "file") { // FICHERO DE GRID
      //Obtenemos los datos del Grid y el usuario
      uri_host = obtenerHost(uri);
      user = UserDiscovery.getUser(global_user, uri_host);

      // Obtenemos los protocolos de transferencia para el Grid
      // (1º Intentamos ver si es grid normal, si no, intentamos ver si es grid
fiable)
      grid = GridDiscovery.getGrid(uri_host);
      if(grid == null) TrustedGridDiscovery.getGrid(uri_host);
      protocolos[] = grid.getServices();

      // Obtenemos la ruta del fichero y comprobamos si tiene compodines
      ruta = obtenerRuta(uri);
      Si(hasWildcards(ruta)) {   // TIENE COMODINES
         // Reemplazamos los comodines
         aux[] = replaceWildcards(ruta, ssh_session);
         // Separamos los ficheros
         ficheros[] = separarFicheros(aux[0]);

         // Tratamos los ficheros por separado
         Para j desde 0 hasta longitud(ficheros[]) - 1 hacer {
          numero_protocolo = 0;

            // Intentamos descargar el fichero. Si hay error probamos con el
siguiente protocolo.
            Repetir {
               //Construimos la nueva URI
               nueva_uri =
construirURI(ficheros[j],protocolos[numero_protocolo],uri);
               numero_protocolo++;

               //Descargamos el fichero
               resultado[] = download(nueva_uri, global_user, destination_path,
ssh_session);
               resultado[1] = salida[1]; // Mensaje de error
            }mientras_que(noHayError(resultado[1]) AND numero_protocolo <
longitud(protocolos[]))

            //Obtenemos los nombre de los ficheros
            resultado[0] = resultado[0] + " " + obtenerNombreFichero(uri);
         }
      }
      si no { // NO TIENE COMODINES
         // Intentamos descargar el fichero. Si hay error probamos con el
siguiente protocolo.
         Repetir {
            //Construimos la nueva URI
            nueva_uri = construirURI(ruta, protocolos[numero_protocolo],uri);
            numero_protocolo++;

            //Descargamos el fichero
            salida[] = download(nueva_uri, global_user, destination_path,
ssh_session);
            resultado[1] = salida[1]; // Mensaje de error
```

```
        }mientras_que(noHayError(salida[1]) AND numero_protocolo <
longitud(protocolos[]))

        //Obtenemos el nombre del fichero
        resultado[0] = obtenerNombreFichero(uri);
      }
   }
   sino si (protocolo == "http" OR protocolo == "https") { // HTTP/HTTPS
       //Construir comando wget
       comando = construirWget(uri, destination_path);

       salida[] = ejecutar(ssh_sesion, comando);

       //Obtenemos el nombre del fichero
       resultado[0] = obtenerNombreFichero(uri);

       // Comprobamos si hay errores
       resultado[1] = comprobarSiHayErrorWget(salida[1]);
   }
   sino si (protocolo == "ftp") { // FTP
       //Construir comando ftp
       comando = construirFTP(uri, destination_path);

       salida = ejecutar(ssh_sesion, comando);

       //Obtenemos el nombre del fichero
       resultado[0] = obtenerNombreFichero(uri);

       // Comprobamos si hay errores
       resultado[1] = comprobarSiHayErrorFTP(salida[1]);
   }
   sino si (protocolo == "sftp") { // SFTP
       //Construir comando sftp
       comando = construirSFTP(uri, destination_path);

       salida = ejecutar(ssh_sesion, comando);

       //Obtenemos el nombre del fichero
       resultado[0] = obtenerNombreFichero(uri);

       // Comprobamos si hay errores
       resultado[1] = comprobarSiHayErrorSFTP(salida[1]);
   }
   sino si (protocolo == "scp") { // SCP
       //Construir comando scp
       comando = construirSCP(uri, destination_path);

       salida = ejecutar(ssh_sesion, comando);

       //Obtenemos el nombre del fichero
       resultado[0] = obtenerNombreFichero(uri);

       // Comprobamos si hay errores
       resultado[1] = comprobarSiHayErrorSCP(salida[1]);
   }
   sino {
      resultado[1] = "ERROR(10): Invalid URI " + uri;
   }

   devolver resultado;
}
```

Figura 49: Algoritmo para la operación `download`

Anexo VIII - Mediadores

Para permitir la interacción automática y transparente entre la capa de ejecución del sistema y las infraestructuras de computación dónde se ejecutan los trabajos que componen un *workflow científico* se utilizan un conjunto de mediadores. Los mediadores interactúan con el *middleware* que gestiona el *grid* para permitir la ejecución de las tareas indicadas por parte del coordinador. La necesidad de definir diferentes mediadores viene determinada por los diferentes *middlewares* utilizados para gestionar los diferentes entornos de ejecución posibles. Dichos *middlewares* son muy diferentes entre ellos provocando que no exista compatibilidad ente ellos y que la migración de uno a otro sea compleja.

La utilización de un conjunto de mediadores permite que el sistema pueda integrar cualquier tipo de *grid*, independientemente del *middleware* que utilice. Para ello, sólo es necesario implementar un mediador que gestione y controle la ejecución de los trabajos en ese entorno de ejecución concreto, gracias a que la descripción de los trabajos y la coordinación dentro del sistema es independiente de la infraestructura de computación. En nuestro caso, hemos desarrollado un mediador para el *middleware* Condor [W4] y otro mediador para el *middleware* gLite [W7], ya que son los *middlewares* de computación que gestionan las infraestructuras *grid* a las que tiene acceso el grupo de investigación GIDHE [W1]. La implementación de un nuevo mediador que gestione la ejecución en otro sistema puede realizarse de forma similar a la utilizada para implementar los dos mediadores ya desarrollados, incluyendo las particularidades de ejecución que presente ese sistema.

En este anexo, vamos a analizar el diseño utilizado para realizar dichos mediadores y después analizaremos las particularidades de implementación más relevantes de los mediadores de Condor y gLite.

VIII.1 - Arquitectura general de un mediador

En la figura 50, se muestra el diseño arquitectural realizado para los mediadores, independientemente del *middleware* concreto con el que interactúa el mediador. En la figura puede observarse como el mediador está dividido en una serie de componentes. El primero de ellos es el que se conoce como *Job Manager*. Este componente se encarga de gestionar y coordinar la ejecución de los diferentes trabajos que son encargados al mediador. Las principales tareas de las que se encarga este componente son las siguientes:

- Interactuar con el *broker* de mensajes Linda, para obtener los trabajos que hay que ejecutar en el *grid* y para escribir los resultados de los mismos en el *broker*.
- Mover los datos generados por el trabajo a su ubicación final correspondiente, ya sea dentro del mismo *grid* o entre *grids*, utilizando para ello el componente de movimiento de datos.
- Mantener una lista de los trabajos que se encuentran en ejecución. Esta lista indica la correspondencia entre los identificadores asignados por el *grid* concreto en el que se ejecuta un trabajo y el propio trabajo. Sirve saber qué trabajo ha finalizado en cada momento.

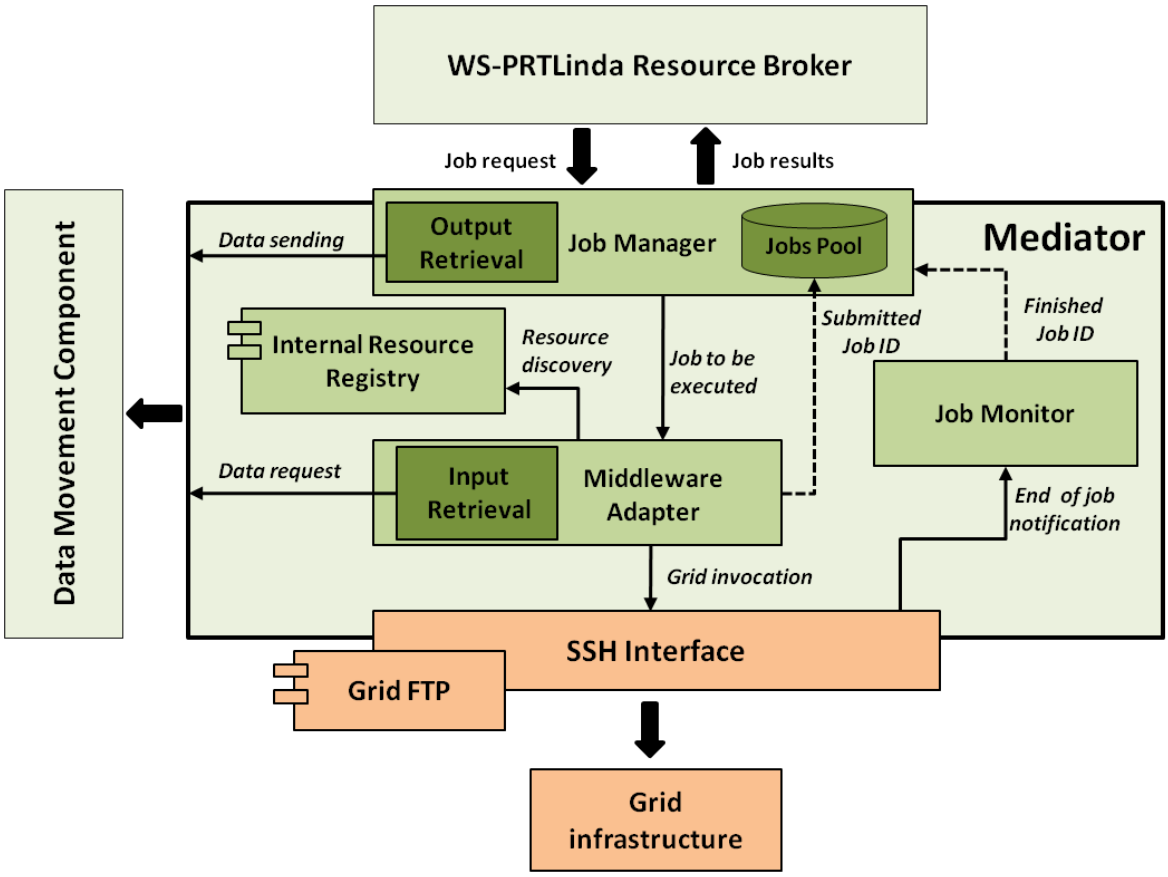

Figura 50: Arquitectura general de un mediador

Por su parte, el *Middleware Adapter* se encarga de transformar la descripción del trabajo obtenida del *broker*, la cual es independiente del entorno de ejecución, en una descripción que pueda ser ejecutada por el *middleware* de la infraestructura *grid*. Asimismo, prepara los datos necesarios para poder ejecutar el trabajo y realiza el despliegue del mismo. Para recuperar los datos necesarios, se utiliza el componente de movimiento de datos, que es común a todos los mediadores. Mientras que, para realizar el despliegue del trabajo se utiliza una interfaz SSH que nos permite establecer una conexión segura con la infraestructura. Una vez desplegado el trabajo, el *middleware* utilizará internamente protocolos como GridFTP para mover los datos a su ubicación final dentro del *grid*.

El *Internal Resource Registry* se utiliza a la hora de realizar la transformación comentada anteriormente. Este componente, obtiene información referente a la aplicación que se desea ejecutar y el usuario que va a ejecutar dicha aplicación utilizando la misma para definir el trabajo a ejecutar.

El último componente del mediador es el *Job Monitor*. La necesidad del mismo viene determinada por el comportamiento asíncrono del sistema, en el cual, un trabajo es lanzado pero no se sabe cuándo va a terminar el mismo. Para detectar el fin del trabajo se introduce este componente. La detección de la finalización del trabajo puede realizarse de diferentes formas dependiendo del *middleware* utilizado.

Un mecanismo habitual que permite detectar la finalización de un trabajo es utilizar el correo electrónico. Cuando el trabajo finaliza, el *middleware* se encarga de enviar un correo electrónico a la dirección especificada al encargar la ejecución del trabajo. En otros

middlewares, este mecanismo no está soportado por lo que es necesario utilizar un mecanismo de encuesta para detectar si un trabajo determinado ha finalizado.

De esta forma, cuando un trabajo finaliza, se detecta esta situación con alguno de los mecanismos anteriores y se notifica de este hecho al *Job Manager* indicando el identificador utilizado por el *grid* para el trabajo que ha finalizado. Este identificador, permite recuperar al *Job Manager* los datos del trabajo, realizar el movimiento de los datos de salida, si es necesario, y notificar al *broker* de dicho evento.

Otro aspecto importante que debe ser controlado por el mediador es todo lo relativo a la aparición de errores. Los errores pueden aparecer tanto antes de ejecutar el trabajo, debido a fallos en la definición del trabajo o en el movimiento de los datos necesarios, por ejemplo, como después de ejecutar el mismo, es el caso de errores de ejecución del trabajo o de movimiento de los datos de salida. Si se produce un fallo, el mediador debe detectar esta circunstancia y notificar al *broker* de mensajes del mismo para que el fallo sea gestionado por el componente de gestión de fallos.

VIII.2 - Job Manager

El *Job Manager* tiene la responsabilidad de interactuar con el *broker* de mensajes para obtener los trabajos que se deben ejecutar en el *grid* correspondiente al mediador y situar en el *broker* las tuplas que indican el fin de un trabajo o el error que se ha producido al ejecutar el mismo. El *Job Manager* espera una tupla mediante una instrucción de lectura destructiva (*in*) utilizando el patrón `["DEPLOYED", "?", "?", "?", ["?", grid]]` para obtener tuplas que vayan guiadas a ese *grid* o que hayan sido emparejadas por la política de emparejamiento competitivo definida en el *broker* WS-PTRLinda. Una vez recibida la tupla, el *Job Manager* crea un nuevo hilo de ejecución para que la misma sea tratada por el *Middleware Adapter* y espera una nueva tupla.

Cuando la tupla ha sido gestionada por el *Middleware Adapter* y se ordena la ejecución del trabajo asociado, éste envía al *Job Manager* el identificador asignado por el *grid* al trabajo. Este identificador es almacenado junto con la tupla correspondiente al trabajo en el *Jobs Pool*. Este directorio nos permite recuperar los datos de un trabajo cuando finaliza el mismo para poder crear la tupla resultado correspondiente al trabajo. La implementación elegida para este almacén es una tabla *hash* para que el almacenamiento y recuperación de la relación entre el identificador y la tupla que describe el trabajo sea muy rápida.

Al finalizar un trabajo, además de obtener la tupla correspondiente al identificador del trabajo que ha finalizado hay que realizar dos tareas adicionales. La primera de ellas es recuperar la salida de dicho trabajo y moverla, utilizando el componente de movimiento de datos, al lugar adecuado. La segunda es recuperar el *log* de ejecución de dicho trabajo que es pasado en la tupla de salida de forma transparente para el usuario.

Por tanto, la tupla resultado contiene la palabra clave `EXECUTED`, seguida del identificador externo del trabajo (compuesto por el nombre de usuario, lista de ficheros de salida, salida estándar y salida de error), el estado de finalización del trabajo y el mensaje de *log*. Una descripción de esta tupla puede observarse en la figura 51.

```
[ "EXECUTED", usuario, lista de ficheros de salida, salida estándar, salida de
error, estado, log ]
```

Figura 51: Descripción de una tupla de salida

Si el trabajo no finaliza *correctamente* y se produce un error, la tupla depositada en el *broker* utiliza la palabra clave `ERROR` seguida de la descripción del trabajo, en el mismo formato que la tupla de entrada y una cadena que contiene el mensaje que identifica al fallo producido para que éste sea tratado adecuadamente por el componente de gestión de fallos. En la figura 52 puede observarse una descripción de una tupla de error.

```
[
  "ERROR",
  [aplicación, argumentos, lista de ficheros de entrada, lista de ficheros de
salida],
  [entrada estándar, salida estándar, salida de error],
  [usuario, grid],
  [fecha, presupuesto],
  mensaje de error
]
```

Figura 52: Descripción de una tupla de error

VIII.3 - Middleware Adapter

El *Middleware Adapter* tiene la función de convertir la representación independiente del entorno de ejecución utilizada en las tuplas para describir los trabajos, en una representación que el *middleware* sea capaz de interpretar para llevar a cabo la ejecución del mismo.

Antes de encargar la ejecución del trabajo, el componente debe examinar las diferentes URIs de los ficheros involucrados en la ejecución del trabajo. Estas URIs determinan la ubicación exacta de cada uno de los ficheros y permiten al componente tanto identificar qué ficheros deben ser movidos al *grid*, encargando dicha transferencia al componente de movimiento, como establecer las rutas locales correspondientes a cada fichero.

Para realizar la traducción de la información de la tupla al formato entendido por el *middleware* del *grid*, el componente utiliza información proporcionada por el *Internal Resource Registry*. Esta información indica dónde se encuentra la aplicación a ejecutar dentro del *grid* y una serie de opciones específicas para la ejecución de dicha aplicación dentro del *grid*. De esta forma, se permite al usuario establecer parámetros específicos para la ejecución del trabajo, haciendo más personalizable y ajustable la ejecución del mismo.

Una vez que se ha realizado la traducción, se realiza el despliegue de la aplicación encargando al *middleware* la ejecución efectiva del trabajo. Como resultado de este proceso, se obtiene un identificador del trabajo desplegado. El *Middleware Adapter* devuelve dicho identificador al *Job Manager* para que este guarde la correspondencia entre el identificador y la tupla del trabajo.

Finalmente, el componente crea o comprueba la existencia, dependiendo del caso en el que nos encontremos, de un nuevo componente *Job Monitor* que se encargue de detectar el final del trabajo para la posterior recuperación de los resultados.

VIII.4 - Internal Resource Registry

El registro de recursos interno se utiliza para almacenar y consultar información sobre el propio *grid*, los usuarios que pueden ejecutar en el mismo y las aplicaciones que es posible ejecutar dentro del *grid*. La implementación del mismo se ha realizado de la misma manera que el resto de registros del sistema.

Más detalles acera de su implementación pueden ser consultados en el *Anexo IV - Registros del sistema*.

VIII.5 - Job Monitor

El *Job Monitor* es el componente encargado de detectar el fin de los trabajos que están siendo ejecutados en el *grid*. Este aspecto es altamente dependiente del *middleware* que gestiona la infraestructura y por ello varía enormemente. En general, para la notificación del fin de un trabajo se pueden usar dos mecanismos diferentes, un mecanismo en el que el *grid* avisa al usuario del fin de un trabajo o un mecanismo en el que es el usuario el que tiene que preguntar al *middleware* si el trabajo ha finalizado.

En el primer caso, el mecanismo más habitual es la utilización del correo electrónico para notificar la finalización de cada trabajo. Al definir el trabajo, el usuario indica una dirección de correo electrónico y cuando el mismo finaliza, o sufre un error, el *middleware* se encarga de enviar un correo electrónico a dicha dirección con el estado de terminación del trabajo o el error producido. Este es el mecanismo utilizado por Condor.

La segunda alternativa deja la responsabilidad de detectar el final de un trabajo al usuario del *grid*. Éste debe ejecutar algún comando que le indique el estado del trabajo para poder detectar si el mismo ha finalizado. Este tipo de mecanismo es el utilizado por el *middleware* gLite.

En lo que respecta a la implementación del componente, existe una diferencia fundamental entre ambos tipos de mecanismos de notificación. En el caso del aviso por correo electrónico, es necesario utilizar un proceso por cada dirección de correo electrónico que se encuentre activa en el sistema. Estos procesos se encargarán de chequear si ha llegado alguna notificación a la dirección de correo asignada. Por su parte, en el caso de que el usuario sea el encargado de preguntar al *grid* el estado de un trabajo, es necesario utilizar un proceso por cada trabajo que se está ejecutando, de forma que, cada proceso se encarga de un único trabajo. Otra alternativa consistiría en utilizar un solo proceso por cada usuario. Esta opción utiliza menos recursos pero dificulta enormemente la gestión de las notificaciones y perjudica la eficiencia, por lo que no es recomendable.

VIII.6 - Mediador de Condor

Condor es el *middleware* utilizado por el *cluster* Hermes del I3A. El mediador que permite desplegar trabajos en dicha infraestructura *grid* se ha implementado de acuerdo a los aspectos comentados anteriormente.

En lo que respecta al *Job Monitor*, se ha desarrollado un cliente IMAP para obtener los correos electrónicos que notifican el final de un trabajo. De esta forma, cuando un trabajo es desplegado en el *grid*, el *Middleware Adapter* indica al *Job Manager* el identificador asignado al trabajo y éste comprueba si existe un servicio IMAP asociado a la dirección de correo electrónico del usuario, creando el cliente IMAP en el que caso de que no exista el mismo.

En lo que respecta al resto de la implementación del componente, el único aspecto reseñable es que no se utiliza la lista de ficheros de salida para generar las opciones que permiten desplegar el trabajo. Esto se debe a que Condor detecta automáticamente los ficheros de salida generados y los transfiere al directorio desde el que se invocó la ejecución del trabajo cuando éste finaliza. En cualquier caso, sigue siendo necesario el movimiento de los datos de salida tras la finalización del trabajo.

VIII.7 - Mediador de gLite

El *middleware* encargado de gestionar los *grids* Aragrid y Piregrid es *gLite*. La implementación de un mediador que interactúe con dicho *middleware* nos ha permitido desplegar trabajos en ambas infraestructuras sin tener que realizar ningún cambio al mediador.

En este caso, gLite no proporciona un mecanismo de notificación basado en el envío de correos electrónicos al usuario, en su lugar, es el usuario el que tiene que consultar al *middleware* el estado del trabajo a través de un comando. Por tanto, la implementación del componente *Job Monitor* cambia radicalmente respecto al caso de Condor. Ahora, es necesario implementar un mecanismo de encuesta que pregunte al *middleware* el estado del proceso de una forma periódica. Otra diferencia con el caso anterior, es que anteriormente con un mismo cliente podíamos detectar el fin de todos los trabajos asociados a un mismo usuario, ahora es necesario utilizar un cliente para cada trabajo desplegado. Agrupar varios trabajos en un mismo cliente también sería posible pero se descarta esta opción por cuestiones de simplicidad y eficiencia.

Por otra parte, en lo correspondiente al proceso de traducción de la descripción del trabajo en forma de tupla a la representación JDL utilizada en gLite, no se presentan grandes cambios. En este caso, hay que especificar tanto los ficheros de entrada como los de salida, incluyendo la propia aplicación y los ficheros de entrada y salida estándar y salida de error en los mismos.

Una particularidad del mediador gLite es que cuando el trabajo finaliza no se transmiten los datos de salida automáticamente, el usuario tiene que ejecutar un comando para recuperar los mismos en el directorio deseado. Una vez recuperados los mismos pueden ser movidos a la dirección destino final utilizando el componente de movimiento de datos de la forma habitual.

A continuación, en las figuras 53 y 54, mostramos una traza de eventos que muestra el comportamiento del mediador de gLite. La traza se muestra en dos imágenes por cuestiones de espacio y claridad. En esta traza se han obviado algunos detalles y no se ha incluido la interacción con el componente de movimiento de datos por simplicidad.

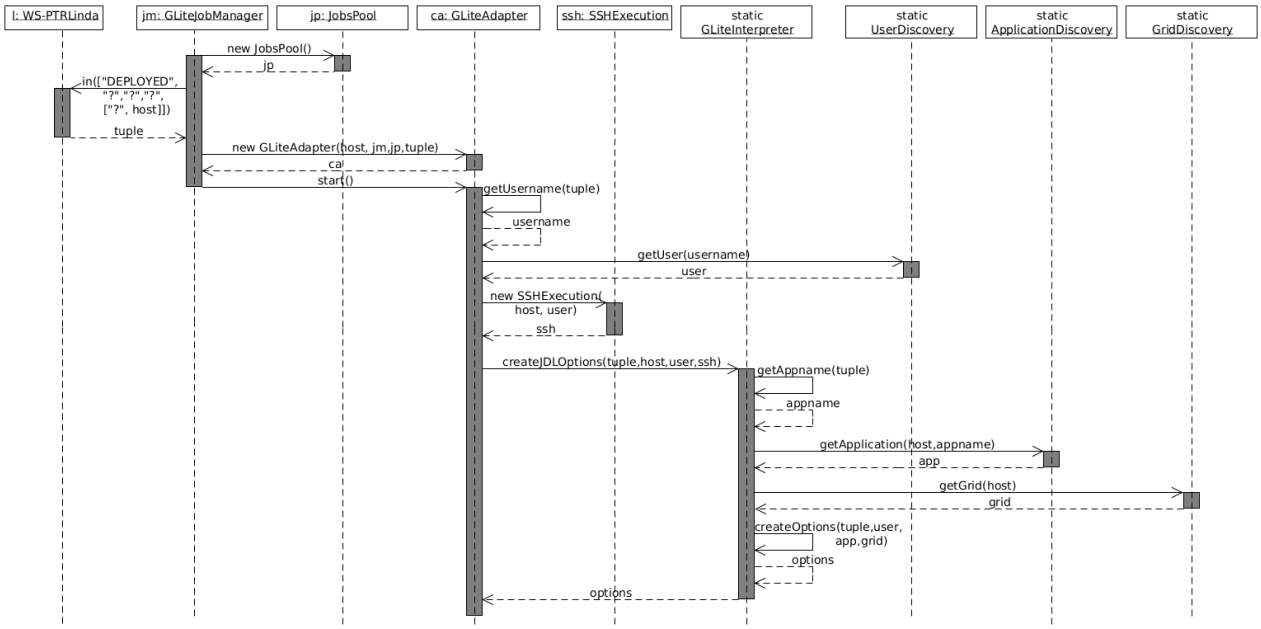

Figura 53: Traza de eventos que muestra el comportamiento del mediador de gLite $\frac{1}{2}$

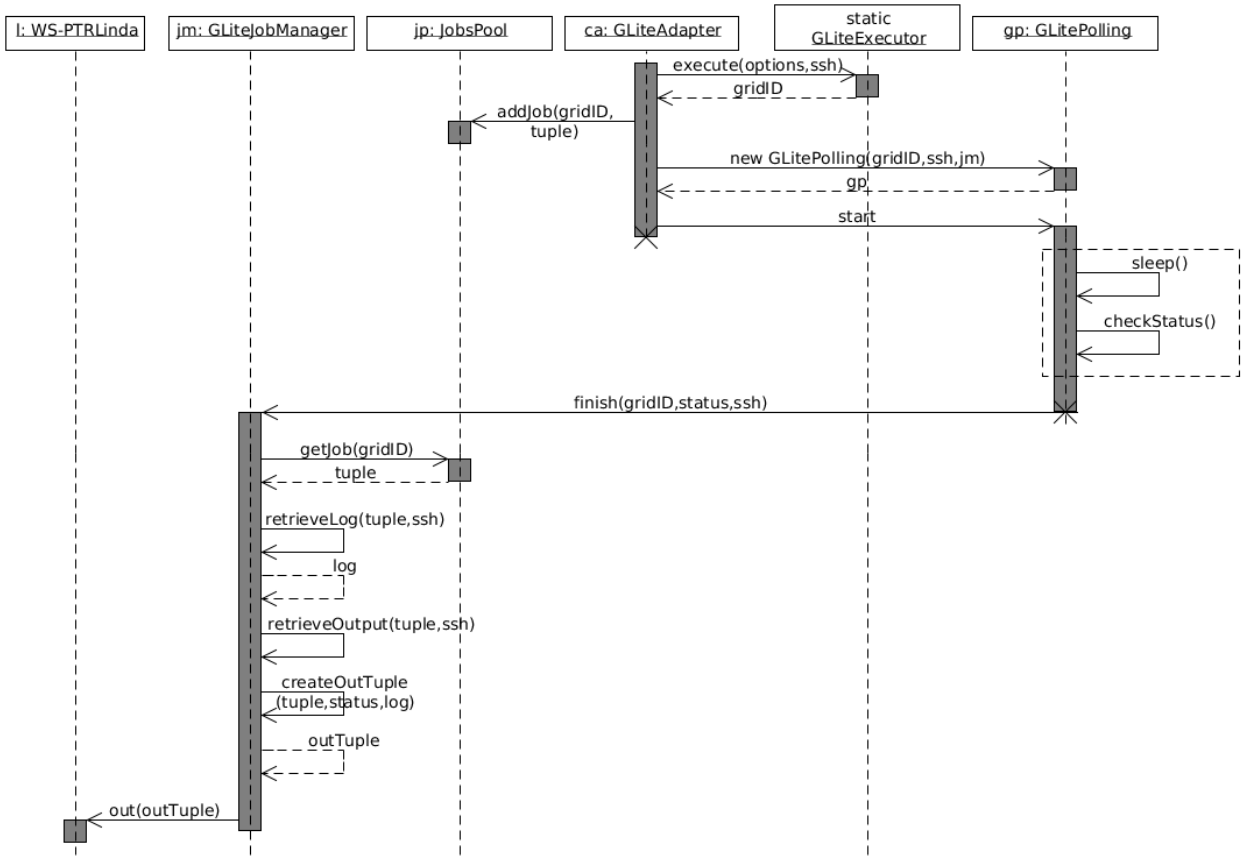

Figura 54: Traza de eventos que muestra el comportamiento del mediador de gLite 2/2

En primer lugar, el `GLiteJobManager` crea el componente `JobsPool`, una tabla *hash* que almacena la concordancia entre los trabajos y el identificador asignado por el *grid*. Después, solicita una tupla al *broker* `WS-PTRLinda` y, cuando la obtiene, crea un nuevo hilo de ejecución que gestione la ejecución de esa tupla. Este hilo corresponde con la clase `GLiteAdapter`.

El hilo obtiene el usuario asociado al trabajo y con su información establece una sesión SSH con el grid utilizando la clase `SSHExecution`. Esta clase es la que nos permite ejecutar comandos en el *grid*. Una vez establecida la conexión, se solicita al `GLiteInterpreter` la transformación de la información de la tupla a un formato que sea entendible por la infraestructura. Este componente, obtiene la información asociada a la aplicación y a la propia infraestructura y realiza la transformación, utilizando para ello tanto los datos de la tupla como los datos personalizados obtenidos del usuario, el *grid* y la aplicación. en este punto, es cuando se contempla el movimiento de los datos necesarios al *grid*. Una vez obtenidas las opciones de ejecución por parte del `GLiteAdapter`, el componente `GLiteExecutor` es el encargado de solicitar la ejecución del trabajo y devolver el identificador que asigna la infraestructura al mismo. Ese identificador es almacenado en la tabla *hash* que mantiene la clase `JobsPool`, indicando la tupla a la que corresponde el mismo.

El siguiente paso, consiste en iniciar el mecanismo de encuesta para detectar el fin del trabajo. Para ello se crea un nuevo hilo de ejecución, el `GLitePolling`, indicando al mismo el identificador del trabajo asignado por el Grid. Este hilo, duerme el tiempo establecida y comprueba el estado del trabajo al despertar, repitiendo este bucle hasta detectar que el trabajo ha finalizado. En este momento, indica el fin del trabajo al `GLiteJobManager`.

La primera acción que realiza el `GLiteJobManager` al recibir la notificación de que un trabajo ha finalizado es obtener la tupla asociada al mismo. Una vez hecho esto, recupera el *log* de ejecución del trabajo y la salida del mismo utilizando el componente de movimiento de datos si fuera necesario. Finalmente, el componente crea la tupla de salida asociada al trabajo e introduce la misma en el *broker* para que sea leída por el entorno de ejecución.

VIII.8 - Mediador de Amazon EC2

La utilización de Amazon Elastic Compute Cloud (EC2) [W9] como infraestructura de computación introduce tres grandes diferencias respecto a la utilización de un *grid*:

1. En Amazon EC2 no existe un *middleware* que gestione la ejecución del trabajo en la infraestructua, sino que se obtiene una máquina virtual que nos permite desplegar directamente los trabajos como si estuviésemos trabajando de forma local.
2. El propio usuario define las características de la infraestructura de computación en lo que se refiere a sistema operativo, memoria, potencia de cálculo y número de máquinas disponibles de acuerdo a los diferentes servicios ofertados por Amazon.
3. Es posible disponer de una infraestructura basada en una serie de máquinas virtuales contratadas, y que sólo se ponen en marcha en caso de necesitarlo. Este caso reduce drásticamente los costes derivados del mantenimiento de las infraestructuras clásicas de *grid*.

Asimismo, una de las características fundamentales de Amazon EC2 es la introducción de IPs elásticas. Una IP elástica es una IP estática que permite la conexión con una instancia de una máquina virtual con la característica de que el usuario puede establecer manualmente el mapeo existente entre la IP y la máquina virtual. Tiene la limitación de que una misma IP elástica corresponde a una única instancia, si bien puede modificarse a qué instancia corresponde y cada usuario puede utilizar 5 IPs elásticas a la vez. Las principales ventajas de este mecanismo son la posibilidad de disponer de una IP estática para acceder a la máquina virtual sin tener que esperar el tiempo habitual de propagación de los servidores DNS yla posibilidad de utilizar una misma IP para acceder a diferentes máquinas virtuales.

A pesar de los cambios que se producen a la hora de ejecutar un trabajo, no es necesario realizar ningún tipo de modificación en la arquitectura propuesta para permitir la ejecución de trabajos en Amazon EC2. La única diferencia reside en la ejecución del trabajo, momento en el que debe obtenerse una máquina virtual para ejecutar el mismo y ordenar la ejecución del mismo de forma explícita. Para ello, Amazon proporciona una API que facilita estas operaciones y que nos permite establecer una conexión SSH con nuestra máquina virtual. Respecto a la detección del fin de un trabajo, se procede de la misma forma que en el mediador de gLite, chequeando el estado del trabajo para detectar si este ha finalizado. El procedimiento seguido para la ejecución de los trabajos se describe a continuación.

Cuando se ordena la ejecución de un trabajo al mediador de Amazon, se consulta la configuración del usuario y se obtienen las diferentes instancias activas del mismo. De acuerdo a las características del trabajo se selecciona la más adecuada para la ejecución del mismo y se realiza una conexión a SSH a la misma a través de la dirección indicada al seleccionar la máquina, habiendo habilitado el puerto 22 previamente. En este punto se podría utilizar una IP elástica para acceder a la máquina pero esto tiene el problema de que la asociación entre la IP y la máquina no es instantánea y tarda unos minutos. Una vez realizada la conexión, se ejecuta el trabajo como si estuviésemos trabajando en nuestro sistema de forma local.

En este caso, al tratarse de un primer prototipo, y debido al coste económico que supone su utilización, se ha limitado el uso de Amazon EC2 a la ejecución de pequeñas experimentos de forma manual. En el futuro se plantea la utilización de *frameworks* de gestión específicos como, por ejemplo, RightScale [W48] que faciliten las labores de despliegue de los trabajos y añadan funcionalidades como monitorización completa de los mismos o balanceo de carga eficiente.

Anexo IX - Evaluación del sistema

La evaluación del rendimiento del sistema desarrollado es un aspecto fundamental para comprobar que se cumplen algunos de los objetivos marcados al inicio del proyecto. En nuestro caso, esta evaluación consiste en el análisis del rendimiento y la escalabilidad del sistema desarrollado y su comparación con otro sistema, como Taverna [W8].

IX.1 - Métricas de evaluación

Los dos aspectos fundamentales que vamos a evaluar son el rendimiento del sistema y la escalabilidad del mismo.

En lo referente al rendimiento del sistema queremos medir la sobrecarga que introduce el sistema en referencia a la ejecución manual de los trabajos. En general, este no es un aspecto crítico en un sistema de gestión de *workflows* científicos ya que los trabajos desplegados son computacionalmente muy costosos y la introducción de una pequeña sobrecarga no es significativa. En cualquier caso, creemos que es importante conocer el rendimiento del sistema en cualquier situación y garantizar que la sobrecarga introducida por el sistema no es un aspecto crítico. Para ello realizamos dos tipos de experimentos:

1. En un primer experimento, queremos comprobar el rendimiento del sistema al ejecutar un trabajo de forma aislada. El objetivo de este experimento es comprobar la sobrecarga natural que introduce el sistema.
2. El segundo experimento busca comprobar el rendimiento del sistema cuando se ejecuta un trabajo en una situación en la que existe una gran carga en el sistema, en cuanto a número de trabajos pendientes de ejecución. Este experimento tiene el objetivo de comprobar si la existencia de una gran carga en el sistema afecta a la ejecución aislada de un trabajo por lo que también es importante para la escalabilidad del sistema.

Por su parte, en lo que respecta a la escalabilidad del sistema, vamos a medir el rendimiento del *broker* cuando se somete al mismo a una gran carga en cuanto a número de trabajos que se ejecutan a la vez. Este aspecto es fundamental a la hora de determinar el número de trabajos que pueden ejecutarse al mismo tiempo sin que se reduzcan gravemente las prestaciones. Para ello, vamos a desplegar la misma tarea un gran número de veces de forma que haya un gran número de peticiones de lectura y peticiones de escritura a la vez.

Tanto para medir el rendimiento del sistema como para medir la escalabilidad del mismo, realizaremos los experimentos con el objetivo de comparar el sistema desarrollado con Taverna, otro sistema de gestión de *workflows* científicos. Esta comparación aborda tanto las capas de modelado como la capa de ejecución del sistema por lo que se contemplan tres posibilidades: la utilización de Taverna de forma aislada, la integración de Taverna con el sistema desarrollado y la utilización de Renew como entorno de modelado del sistema.

IX.2 - Definición de experimentos

Para evaluar el sistema se han definido una serie de experimentos dedicados a medir los aspectos relatados anteriormente. Algunos de los experimentos están dedicados a la medición

de la escalabilidad del sistema, mientras que otros miden el rendimiento del mismo en términos de la sobrecarga que introduce el sistema en la ejecución de los trabajos de un *workflow* científico.

Experimento 1: Rendimiento al ejecutar un trabajo de forma aislada

En la figura 55, se muestra el modelo del experimento realizado con el objetivo de medir el rendimiento del sistema al ejecutar un trabajo de forma aislada. En primer lugar se ordena la ejecución de la tarea o trabajo, se ejecuta la misma y, finalmente, se detecta su finalización. Además, este proceso se repite varias veces, de forma que el experimento consiste en la ejecución en serie de una misma tarea un número definido de veces que puede ser configurado como entrada. El objetivo de que el número de tareas sea variable es conseguir mejor precisión en la medida, detectando en qué momento el rendimiento se estabiliza, así como comprobar que el rendimiento no depende del número de tareas ejecutadas ya que estas se ejecutan de forma aislada.

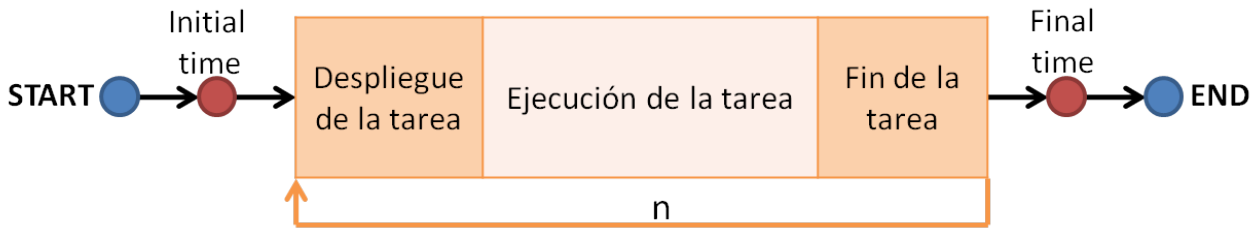

Figura 55: Modelo del experimento realizado para medir el rendimiento del sistema al ejecutar un trabajo de forma aislada.

El trabajo que se va a ejecutar simula una tarea con duración determinada y conocida. Esto nos permite conocer con exactitud qué parte de la ejecución se debe a la sobrecarga introducida por el sistema y qué parte de la misma corresponde a la ejecución propia de la tarea. En caso de la ejecución nativa de la tarea, las fases de despliegue de la tarea y detección del final de la tarea son automáticas. Por su parte, en caso de utilizar el *framework* desarrollado, independientemente de si utilizamos Renew o Taverna como entorno de modelado, la fase de despliegue corresponde con una operación `in` y la fase de detección del fin corresponde con una operación `out`.

Finalmente, en la figura 52 aparece la implementación concreta del experimento en el caso de utilizar Taverna como herramienta de modelado. La figura de la izquierda muestra el modelo general del *workflow* científico desarrollado. Este modelo es el mismo para los dos experimentos modelados con Taverna. La única diferencia es que en el caso de ejecución nativa, el componente *Task* es directamente la aplicación simulada, mientras que, en el caso de la integración de Taverna con el sistema desarrollado, la implementación del componente es la mostrada en la figura de la derecha. Como puede observarse en el modelo se toma el tiempo al inicio de la simulación y al final de la misma, obteniendo como resultado el tiempo medio de ejecución.

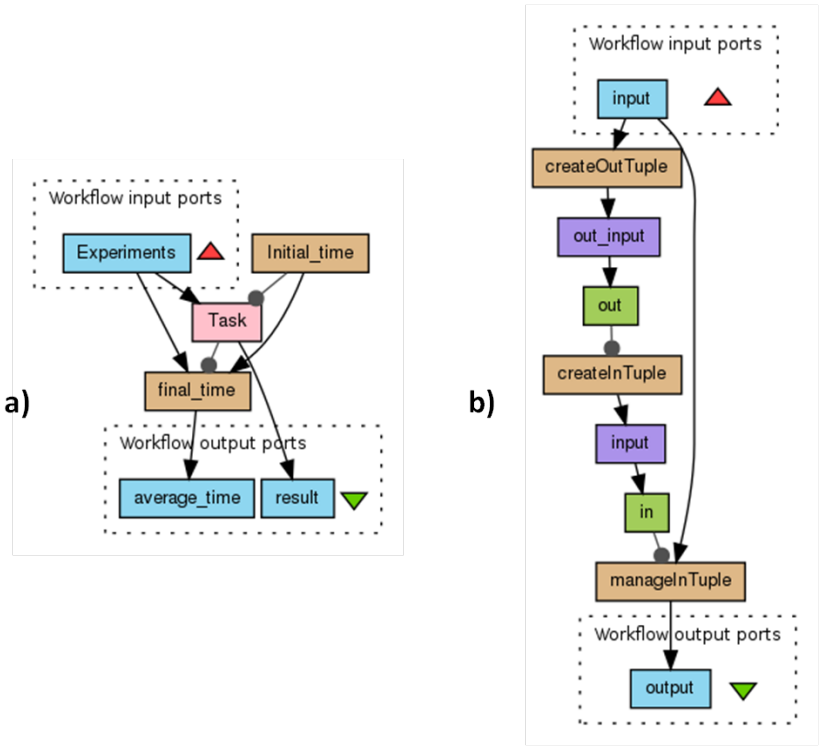

Figura 56: *Modelado del experimento con Taverna.* **a)** *Modelo global del workflow.* **b)** *Modelo de la tarea en el caso de integración de Taverna con el framework*

De la misma manera, en la figura 57 se muestra la implementación realizada utilizando Renew como entorno de modelado y las redes de referencia como herramienta de modelado del *workflow* correspondiente al experimento. Como puede observarse la implementación es análoga a la presentada en el caso anterior. La única diferencia es que en este caso el bucle que repite las tareas se muestra de forma explícita y en el caso de Taverna se muestra de forma implícita.

Experimento 2: Rendimiento al ejecutar un trabajo en situación de estrés

Dada la naturaleza no determinista de las operaciones sobre el espacio de tuplas, se plantea un calentamiento previo o *warm-up* del espacio de tuplas. Esto provoca que el espacio contenga un elevado número de potenciales tuplas candidatas a la hora de realizar el emparejamiento durante la ejecución de los experimentos, lo que correspondería con una situación de estrés en el repositorio, y aumenta los tiempos de respuesta, resultando en un escenario muy próximo a la realidad [B2, B19].

En la figura 58, puede observarse el modelo de este experimento. En primer lugar, se realiza la fase de calentamiento del espacio de tuplas del *broker* (*tuple space warm-up*) y a continuación se sitúa el modelo utilizado para el primer experimento.

En este caso tan sólo utilizamos Renew para realizar el mismo. No es necesario utilizar Taverna porque obtendríamos los mismos resultados con la única diferencia de la sobrecarga propia de cada sistema que ya se observó en el experimento anterior.

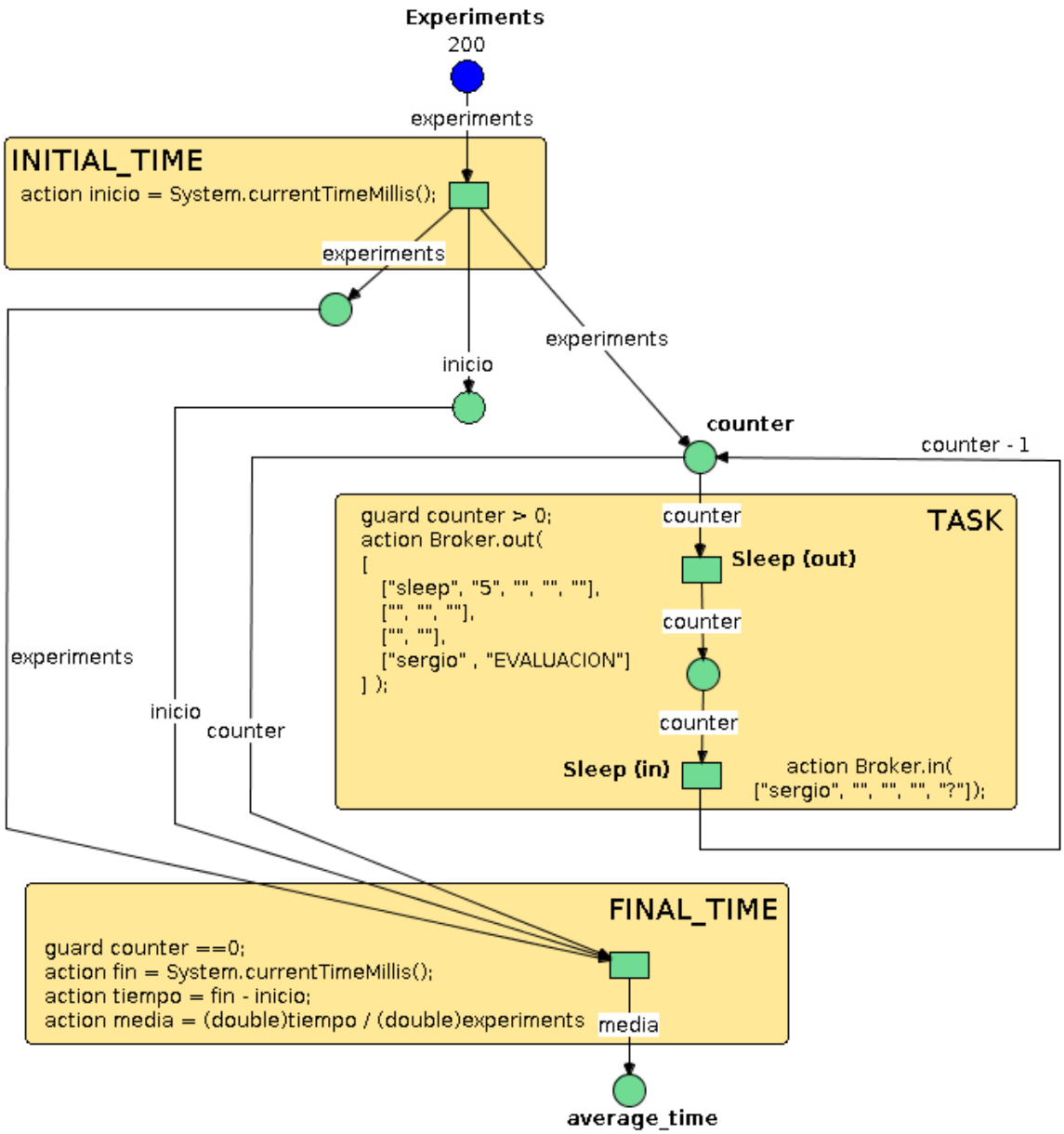

Figura 57: Modelado del primer experimento con Renew.

La implementación del experimento, es la misma que la realizada en el caso anterior con una operación `out` inicial seguida de una operación `in`. La figura 59 muestra el modelo concreto del experimento realizado. Un detalle importante es que se ha buscado que las tuplas introducidas en la sobrecarga del sistema sean lo más parecidas posible a las tuplas reales para aumentar el realismo en la comparación.

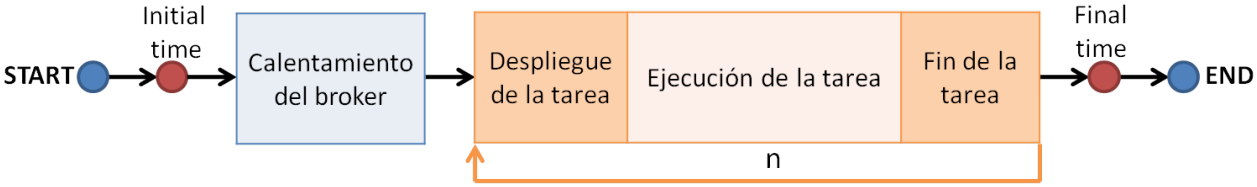

Figura 58: Modelo del experimento realizado para medir el rendimiento del sistema al ejecutar un trabajo en una situación de estrés.

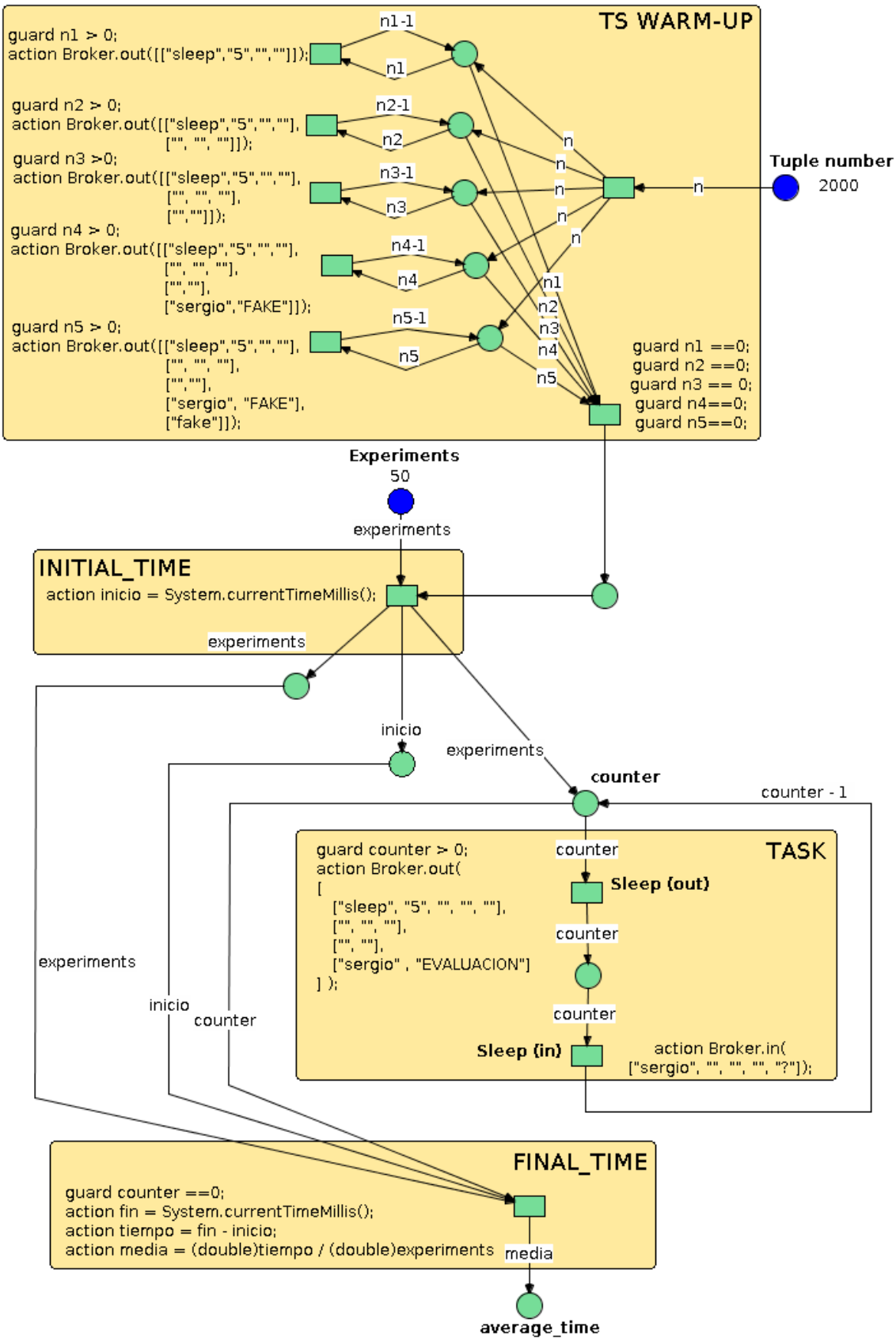

Figura 59: Modelado del segundo experimento con Renew.

Experimento 3: Escalabilidad del sistema

El modelo del último experimento realizado puede observarse en la figura 60. El experimento consiste en ejecutar en paralelo diferente número de tareas de forma que se realicen a la vez un gran número de operaciones de escritura y de lectura en el *broker*.

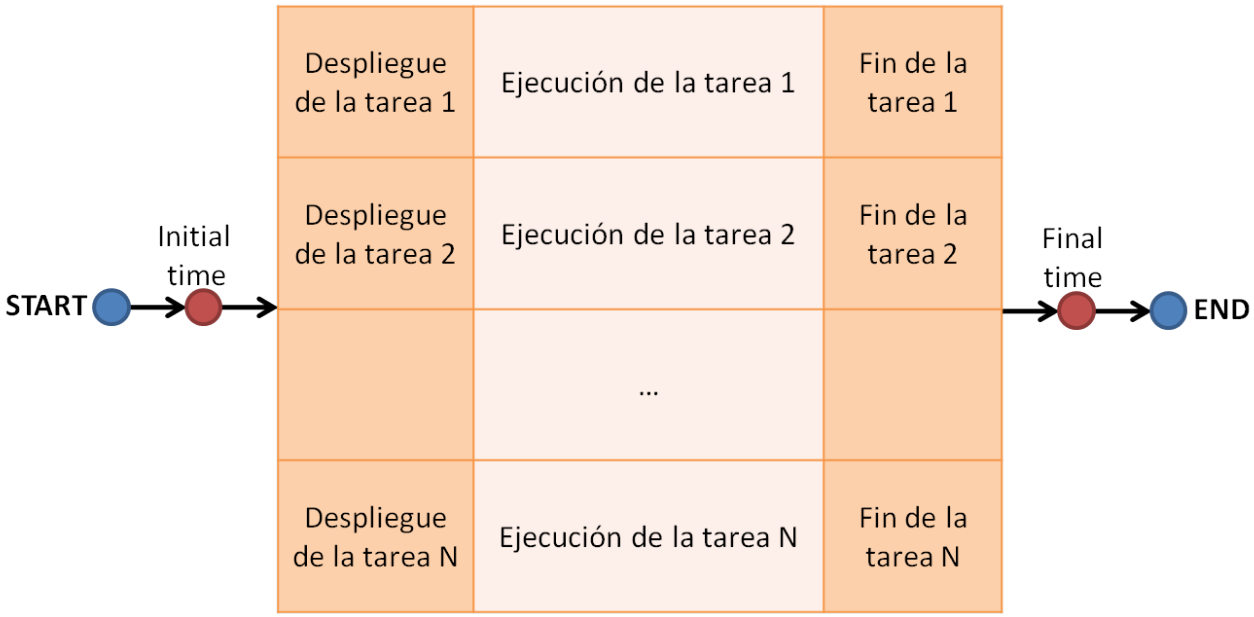

Figura 60: Modelo del experimento que comprueba la escalabilidad del sistema.

Para realizar este experimento se va a utilizar tanto Renew como Taverna. En Renew, el despliegue de tareas en paralelo surge como un aspecto natural. Sin embargo, en Taverna existe la limitación de que no hay mecanismos que permitan definir la ejecución de tareas en paralelo por lo que hay que añadir manualmente tantas ramas como ejecuciones en paralelo deseemos.

De esta forma, en la figura 61, podemos observar el modelo del *workflow* utilizado en Taverna para el caso de 5 tareas ejecutándose en paralelo. Para el caso de un número diferente de tareas, tan sólo hay que variar el número de ramas indicadas. La implementación concreta de cada tarea es la presentada en la figura 2 b).

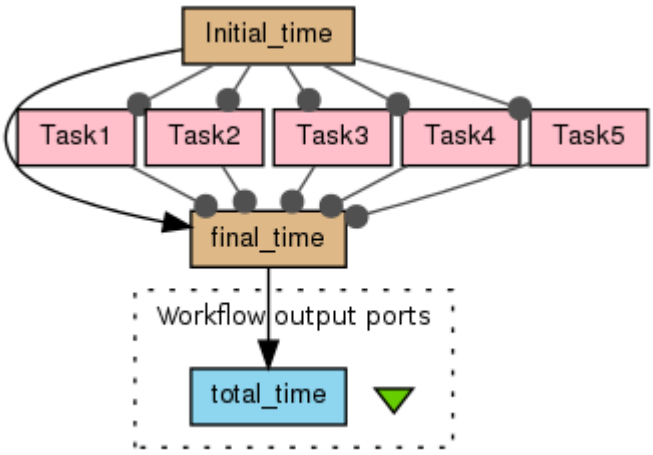

Figura 61: Modelado del tercer experimento con Taverna.

Por su parte, en la figura 62 se proporciona el modelo de *workflow* utilizado para realizar el experimento en Renew. En este caso, se proporciona un dato de entrada que permite fijar el número de tareas que se ejecutan en paralelo, facilitando enormemente la realización del experimento.

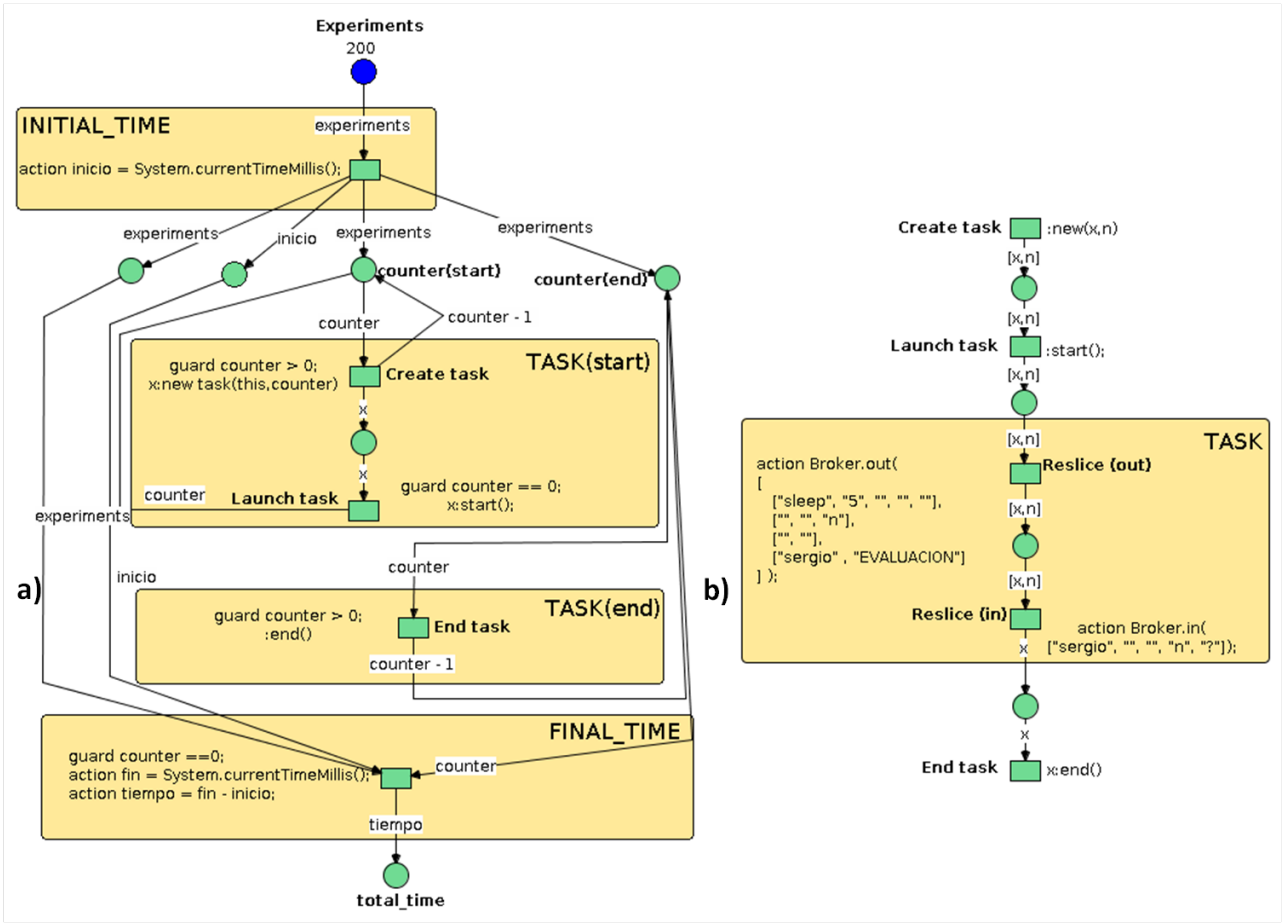

Figura 62: Modelado del tercer experimento con Renew. **a)** Modelado del *workflow*. **b)** Modelado de la tarea ejecutada.

En los experimentos realizados con Taverna, vamos a utilizar una tarea de duración fija de 5 segundos mientras que en los experimentos realizados con Renew vamos a modificar la duración de la tarea. En total vamos a realizar tres experimentos con las siguientes características:

1. En el primer experimento vamos a utilizar una tarea de duración fija de 5 segundos, con el objetivo de poder comparar las herramientas Renew y Taverna en cuanto a escalabilidad.
2. En el segundo experimento vamos a utilizar una tarea de duración fija de 20 segundos. Este experimento tiene como objetivo comprobar si distanciar las operaciones de escritura y lectura en el *broker* para que estas no coincidan mejora el rendimiento del sistema.
3. En el tercer experimento vamos a utilizar una tarea cuya duración puede oscilar entre 5 y 20 segundos. Este experimento tiene como objetivo comprobar como afecta al rendimiento del sistema variar el momento de finalización de las tareas.

IX.3 - Resultados

A continuación, vamos a analizar los resultados de los experimentos realizados. Para ello se proporcionan las tablas con los resultados obtenidos y las gráficas que nos permiten observar dichos datos de una forma más sencilla.

Experimento 1: Rendimiento al ejecutar un trabajo de forma aislada

En la tabla 5, se pueden observar los resultados de la ejecución de los tres experimentos realizados para medir el rendimiento del sistema. En la misma, podemos observar la sobrecarga media que experimenta el sistema una vez restado el tiempo de ejecución de la tarea simulada. Estos datos pueden observarse de forma gráfica en la figura 63.

	5	10	25	50	100	200
Taverna + infraestructura	212,35	194,97	190,92	175,08	173,5	172,07
Framework	60,4	55,1	53,73	56,49	54,22	54,17
Taverna nativo	52,27	48,27	35,78	31,32	24,03	26,26

Tabla 5: Resultados del experimento de rendimiento del sistema al ejecutar un trabajo de forma aislada.

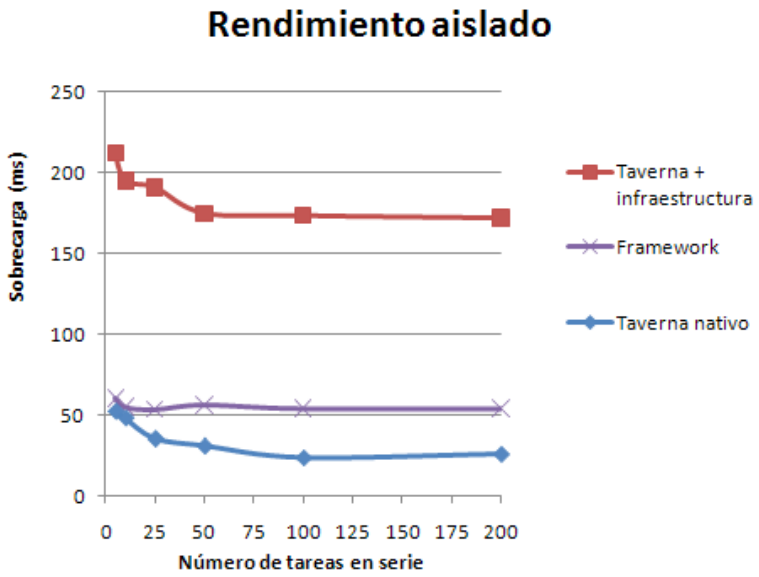

Figura 63: Gráfica que muestra la sobrecarga media del *framework* al realizar diferente número de experimentos en serie de forma aislada.

Los mejores resultados se obtienen en el caso de la ejecución nativa utilizando Taverna en los que la sobrecarga se sitúa en torno a 25 milisegundos. Si utilizamos Renew como herramienta de modelado junto con el *framework* desarrollado para ejecutar el trabajo, obtenemos aproximadamente el doble de sobrecarga, algo más 50 milisegundos. En cualquier caso, la sobrecarga introducida sigue siendo muy leve y no afecta al rendimiento del sistema. Si integramos Taverna y el sistema desarrollado, se obtienen los peores resultados, en torno a 175

milisegundos. Este resultado era de esperar debido a la utilización de la interfaz de servicio web del *broker* que añade una sobrecarga extra al sistema. En cualquier caso, para cualquiera de las alternativas la sobrecarga es muy leve y no perjudica el rendimiento global del *workflow* ejecutado.

Asimismo, podemos comprobar que el rendimiento de Renew es más estable independientemente del número de tareas ejecutadas, mientras que el rendimiento de Taverna decrece al aumentar el número de tareas que se ejecutan en serie. Esto se debe a que Taverna introduce una mayor sobrecarga al ejecutar los experimentos por primera vez, mientras que la ejecución de las tareas en Renew es independiente de este aspecto.

Experimento 2: Rendimiento al ejecutar un trabajo en situación de estrés

Los resultados de este experimento se muestran en la tabla 6. De la misma manera que para el experimento anterior, se proporciona la sobrecarga media obtenida en los experimentos, es decir, el valor de restar al tiempo de ejecución total el tiempo de ejecución de la tarea simulada. Asimismo, se proporciona la figura 64 en la que aparece una gráfica mostrando los contenidos anteriores.

	500	1000	2500	5000	7500	10000
Sobrecarga	437,07	567,93	965,20	1729,20	2448,07	3326,77

Tabla 6: Resultados del experimento de rendimiento del sistema en situación de estrés.

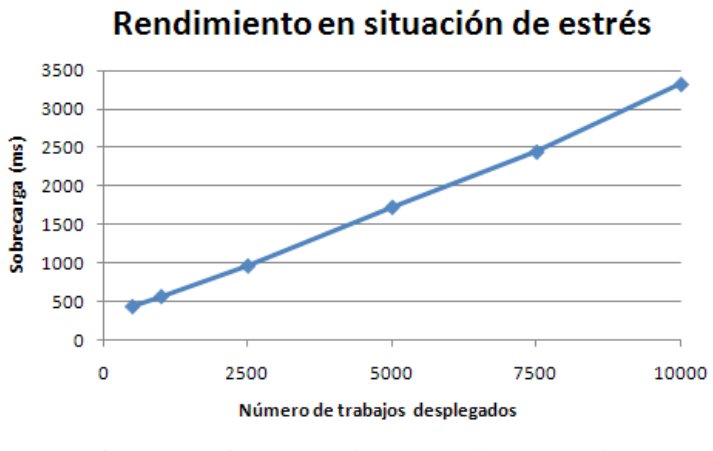

Figura 64: Gráfica que muestra la sobrecarga media del *framework* en situación de estrés.

En este caso, se puede apreciar como al someter al sistema a una situación de estrés se introduce sobrecarga en la ejecución aislada de un trabajo. En cualquier caso, la sobrecarga extra introducida no es muy elevada, presentando un comportamiento lineal incluso para un número tan elevado de trabajos como 10000.

Experimento 3: Escalabilidad del sistema

En la tabla 7 se proporcionan los datos del experimento que nos permite medir la escalabilidad del sistema. En esta tabla tan sólo se proporcionan los resultados utilizando Renew ya que con Taverna no es posible ejecutar de forma paralela tantos trabajos. Asimismo, en la figura 65 se muestran los datos de la sobrecarga introducida por el sistema en función del número de tareas que se ejecutan en paralelo.

	25	50	75	100	125	150	175	200
***Framework* 5 segundos**	2291,11	8889,31	28639,12	62849,23	105821	208266,28	319640,74	453526,30
Framework 20 segundos	2117,23	8579,69	22829,85	57837,31	111804,1	215438,52	314718,69	449812,36
***Framework* Aleatorio 5-20 s**	938,67	7572,23	26892,54	59512,84	99458,96	196981,19	304346,20	424721,33

Tabla 7: Resultados del experimento de evaluación de la escalabilidad del sistema.

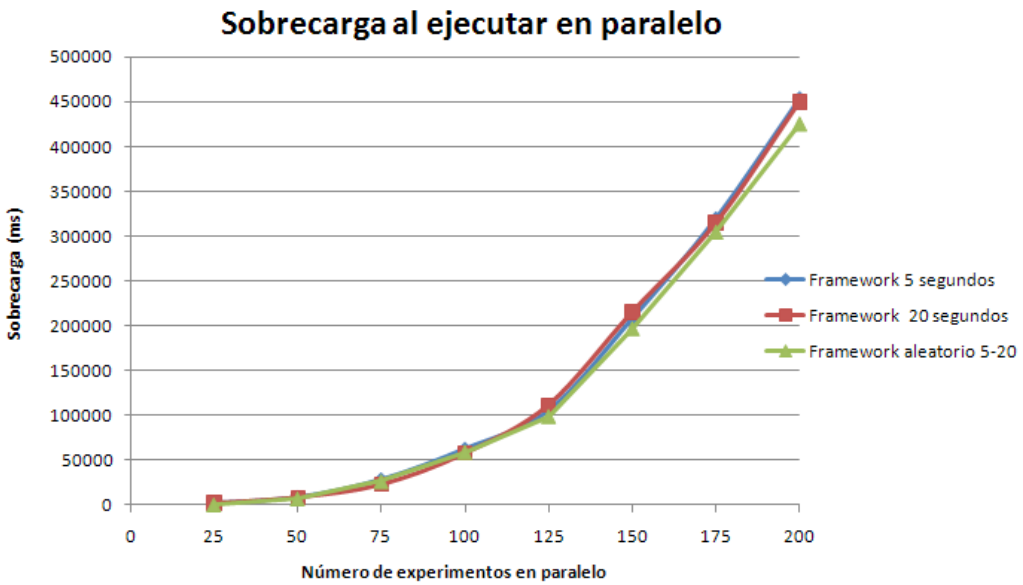

Figura 65: Gráfica que muestra la sobrecarga del *framework* al aumentar el número de tareas en paralelo.

En este caso podemos observar como la sobrecarga introducida en el sistema al incrementar el número de trabajos que se ejecutan en paralelo crece rápidamente. El incremento de la sobrecarga sigue una tendencia polinómica de tercer grado, casi exponencial. La sobrecarga es aceptable hasta un número de 125 trabajos ejecutándose en paralelo y se dispara si intentamos ejecutar un mayor número de trabajos.

Para comparar en cuanto a escalabilidad las herramientas de modelado Renew y Taverna realizamos un análisis más detallado de los resultados obtenidos hasta un máximo de 25 trabajos ejecutándose en paralelo. Los resultados de este análisis se muestran en la tabla 8 y en la figura 66.

	5	10	15	20	25
Taverna 5 segundos + infraestructura	924,26	1938,62	2647,37	4758,67	6776,39
***Framework* 5 segundos**	349,54	843,33	1314,00	1802,09	2291,11
***Framework* 20 segundos**	386,69	758,12	1262,82	1794,40	2117,00

Tabla 8: Resultados de la comparación entre Taverna y Renew en términos de escalabilidad.

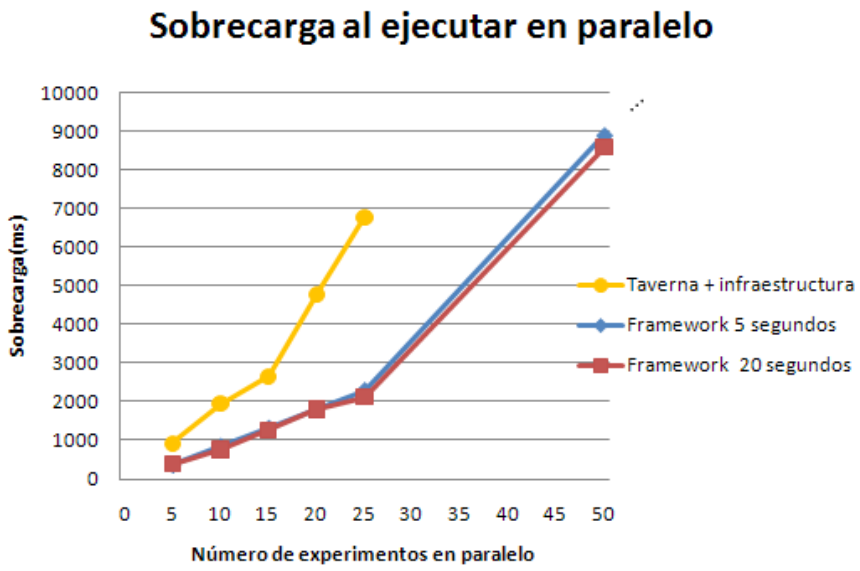

Figura 66: Gráfica que compara Taverna y el *framework* en términos de escalabilidad.

Como puede observarse en la gráfica el incremento de la sobrecarga limitando el número de tareas a 25 es lineal en Renew mientras que es polinómica de tercer grado en Taverna. Esto se debe a que Taverna no proporciona métodos para definir tareas en paralelo de forma implícita, a través de bucles o similar, y hay que hacerlo de forma explícita.

De esta forma, al incrementar el número de tareas el *workflow* se vuelve muy pesado en términos de memoria necesaria para gestionar y ejecutar el mismo provocando que se disparen las prestaciones. Asimismo, este aspecto también provoca que el modelado del *workflow* se ralentice enormemente y que la máquina se colapse. Por esta razón no se han podido realizar experimentos con un mayor número de tareas en paralelo utilizando Taverna.

IX.4 - Conclusiones

Los experimentos realizados demuestran que la sobrecarga introducida por el sistema desarrollado es pequeña y comparable a la sobrecarga introducida por otros sistemas de gestión de *workflows* científicos. Por tanto, el rendimiento del sistema se encuentra dentro de los límites esperados ya que se deben realizar comprobaciones extra por la naturaleza del sistema (elección del Grid, comprobación de si es necesario mover datos, etc.). Asimismo, dicha sobrecarga es estable e independiente de que se hayan ejecutado anteriormente otros trabajos.

La existencia de un gran número de trabajos, o lo que es lo mismo tuplas, en el sistema no afecta a la ejecución aislada de un trabajo, de forma que la sobrecarga introducida es proporcional al número de trabajos que están siendo gestionados por el *broker*. Además, esta sobrecarga no es significativa y se encuentra por debajo de 3 segundos aún cuando existen 10000 tuplas en el *broker* de coordinación.

Finalmente, como era de esperar, la sobrecarga del sistema crece enormemente al ejecutar un elevado número de tareas en paralelo. Esto se debe a que el *broker* se convierte en el cuello de botella del sistema al tener que atender un gran número de peticiones a la vez, siendo especialmente crítica la operación de lectura por la gran cantidad de tuplas que hay que comprobar. Las prestaciones se mantienen dentro de un límite aceptable hasta 125 tareas.

En cualquier caso, la utilización de las redes de referencia permite introducir un gran nivel de paralelismo de forma natural mientras que los sistemas de gestión de *workflows* científicos actuales no suelen proporcionar mecanismos para proporcionar ese nivel de paralelismo. En el caso de Taverna, hay que incluir el paralelismo de forma explícita, no se pueden realizar construcciones genéricas que nos permitan llegar a altos niveles de paralelismo. En lo que se refiere a las prestaciones, esto implica que el rendimiento de Taverna al paralelizar un elevado número de tareas disminuye muy rápidamente.

Anexo X - Caso de estudio: First Provenance Challenge

Como caso de estudio se utilizó el *workflow* científico propuesto en el First Provenance Challenge[W24]. A pesar de que el objetivo de este desafío no es la ejecución en sí del *workflow*, como se comentará a largo de este anexo, la elección del mismo se fundamenta en que es un *workflow* de referencia en la comunidad científica, por lo que ha sido muy estudiado y utilizado para la verificación de diferentes sistemas de gestión de *workflows*. Además, resulta un problema fácilmente escalable y que requiere del desplegado y ejecución de diversas tareas tanto de forma secuencial como paralela, por lo que resulta muy adecuado para demostrar la viabilidad de la solución propuesta.

X.1 - Concepto de provenance

Para entender este desafío hay que introducir en primer lugar el concepto de *provenance* o procedencia de la información, un aspecto muy importante dentro del ámbito de los *workflows* científicos. Al igual que en un experimento científico convencional, en un experimento realizado a través de un *workflow* científico, es necesario saber de dónde se ha obtenido la información, qué procesos y métodos han sido aplicados, cuáles son las relaciones existentes entre los procesos realizados, qué relación existe entre los resultados finales y los resultados intermedios, qué métodos se han utilizado para obtener los diferentes resultados, de qué datos iniciales se ha partido para llegar hasta estos resultados, etc. El concepto de *provenance* hace referencia a esa información y al proceso utilizado para responder a las cuestiones anteriores.

Un aspecto para el cual el *provenance* es de gran importancia, es para garantizar que un experimento es reproducible. Esto es, para garantizar que si realizamos de nuevo el experimento de la misma forma, obtendremos los mismos resultados. Además, éste debe poder reproducirse tanto por la misma persona que ha llevado a cabo el experimento, como por otra persona que desee llevar a cabo el mismo. Por lo tanto, es necesario recoger información precisa que permita reproducir el experimento de forma exacta.

Otro aspecto para el cual el *provenance* resulta interesante, es permitir la comparación de dos o más experimentos. En el ámbito científico, es habitual que dos experimentos no proporcionen exactamente el mismo resultado. Por tanto, es importante poder analizar las diferencias existentes entre ambos experimentos de cara a identificar las causas de que el resultado obtenido en un experimento sea diferente del resultado obtenido en otro experimento similar. De la misma manera, también puede resultar interesante comparar cómo afecta al resultado del experimento modificar alguna etapa del mismo.

En definitiva, es indispensable que, una vez realizado el experimento, seamos capaces de identificar todos aquellos aspectos que se consideren necesarios para conocer cómo se ha llegado hasta ese resultado, para asegurar que el resultado obtenido es correcto, para permitir que el experimento sea reproducible y para permitir el análisis y comparación del experimento con otros experimentos. Por tanto, el sistema que utilicemos para ejecutar los *workflows* científicos debe ser capaz de almacenar toda la información necesaria para dar respuesta a los aspectos comentados anteriormente.

X.2 - Aparición del First Provenance Challenge

Este primer desafío surge durante un debate sobre la estandarización de *provenance* en el International Provenance and Annotation Workshop en el año 2006 (IPAW'06 [W49]). En el mismo, la comunidad decidió que era necesario entender las diferentes representaciones usadas para *provenance*, los aspectos comunes, y las razones de sus diferencias. Como resultado, se acordó organizar un *Provenance Challenge* para comparar y entender los diversos modos de abordar el problema.

El primer desafío de *provenance* fue el *First Provenance Challenge*. Comenzó el 9 de Junio del 2006 y concluyó el 13 de Septiembre en Washington, DC con el evento final y la presentación de los resultados. El éxito de este desafío ha provocado que se hayan realizado otros tres desafíos más, el *Second Provenance Challenge* [W50], *Third Provenance Challenge* [W51] and *Fourth Provenance Challenge* [W52]. Como resultado de los mismos se ha establecido y consolidado un estándar para la representación de la información de *provenance*, el *Open Provenance Model* [W53].

X.3 - Objetivos del First Provenance Challenge

En esta sección vamos a analizar tanto los objetivos originales del *First Provenance Challenge* como los objetivos que nos planteamos nosotros al utilizar este *workflow* como caso de estudio.

Objetivos originales

El *First Provenance Challenge* tiene como objetivo introducir a los participantes en los aspectos relacionados con las capacidades que disponen los sistemas para obtener información de *provenance*, en particular en los siguientes detalles:

- La representación que los sistemas usan para documentar los detalles de los procesos que se han ejecutado.
- Las capacidades de cada sistema para responder a consultas relacionadas con *provenance*.
- Los aspectos que cada sistema considera dentro del ámbito del *provenance* (independientemente de si el sistema puede solucionar todos los problemas que se dan en ese ámbito).

Para ello, se proporciona un ejemplo de un *workflow* real dentro del ámbito de las imágenes obtenidas en resonancias magnéticas funcionales. El objetivo de este desafío, es la información de procedencia, no el despliegue del *workflow* en sí. Por este motivo, no es necesario utilizar los servicios reales, pueden implementarse procedimientos que devuelvan siempre la misma salida independientemente de los parámetros de entrada que son pasados al mismo.

Finalmente, para poder comparar los diferentes sistemas en lo que respecta a la información de *provenance*, se indican una serie de consultas que deben realizarse como resultado del desafío.

Objetivos propios

Nuestro objetivo principal consiste en verificar el correcto funcionamiento de la herramienta desarrollada utilizando un *workflow* científico real. La elección del *workflow* proporcionado en este First Provenance Challenge se debe a que es un *workflow* muy estudiado por la comunidad científica y que ha sido utilizado como caso de estudio en la presentación de diferentes herramientas de gestión de *workflows* científicos.

También queremos ilustrar la flexibilidad del *framework* desarrollado tanto en lo que respecta a la posibilidad de utilizar diferentes entornos de modelado como a la posibilidad de utilizar diferentes infraestructuras *Grid* de forma transparente. Con este propósito presentamos dos implementaciones diferentes una realizada con redes de referencia y otra realizada con el lenguaje de modelado de Taverna. En estas implementaciones se utilizarán los recursos *Grid* a los que tiene acceso el grupo de investigación GIDHE [W1]: Aragrid, Piregrid y Hermes.

Por tanto, nuestro objetivo dista bastante del objetivo original del problema. Sin embargo, esto no supone ningún problema si no que se convierte en una ventaja ya que añadimos valor al propósito original del problema y, además, nos proporciona la base para el desarrollo de una de las líneas futuras de investigación comentadas en el *Capítulo 6*, como es la gestión de *provenance* dentro del sistema.

X.4 - Definición del problema

Se plantea un *workflow* para crear "atlas" cerebrales de la población en base a imágenes de alta resolución del Centro de Datos de imágenes de resonancias magnéticas funcionales (fMRI) [W25]. El *workflow* se muestra en la figura 67.

El *workflow* está compuesto de varios procedimientos, mostrados como óvalos naranjas, y datos que fluyen entre ellos, mostrados como rectángulos. Pueden distinguirse 5 fases o etapas en la ejecución del mismo. Cada una de las etapas se muestra en una misma franja horizontal. Los procedimientos utilizan la suite AIR, para crear una imagen cerebral promedio a partir de una colección de imágenes tridimensionales de alta resolución, y la suite FSL [W11] para crear las imágenes bidimensionales de cada dimensión del cerebro que forman la salida.

Las entradas del *workflow* son un conjunto de imágenes cerebrales (*anatomy images*) y una única imagen de referencia (*reference image*). Todas las imágenes anatómicas de entrada, son escáneres tridimensionales de un cerebro con diferente resolución. Para cada imagen se proporciona el fichero con la imagen y un fichero de cabecera con metadatos sobre la misma.

Las etapas del *workflow* se describen a continuación:

1. Para cada imagen de entrada, el procedimiento **align_warp** compara la imagen de entrada y la imagen de referencia para determinar como debe alinearse la imagen de entrada con respecto a la imagen de referencia. La salida de cada proceso define una serie de parámetros que indican la transformación espacial a realizar sobre cada imagen.
2. Los procedimientos **reslice** realizan de forma efectiva la transformación indica en el paso anterior, utilizando los parámetros obtenidos como salida de align_warp. Como resultado se obtiene la imagen modificada.

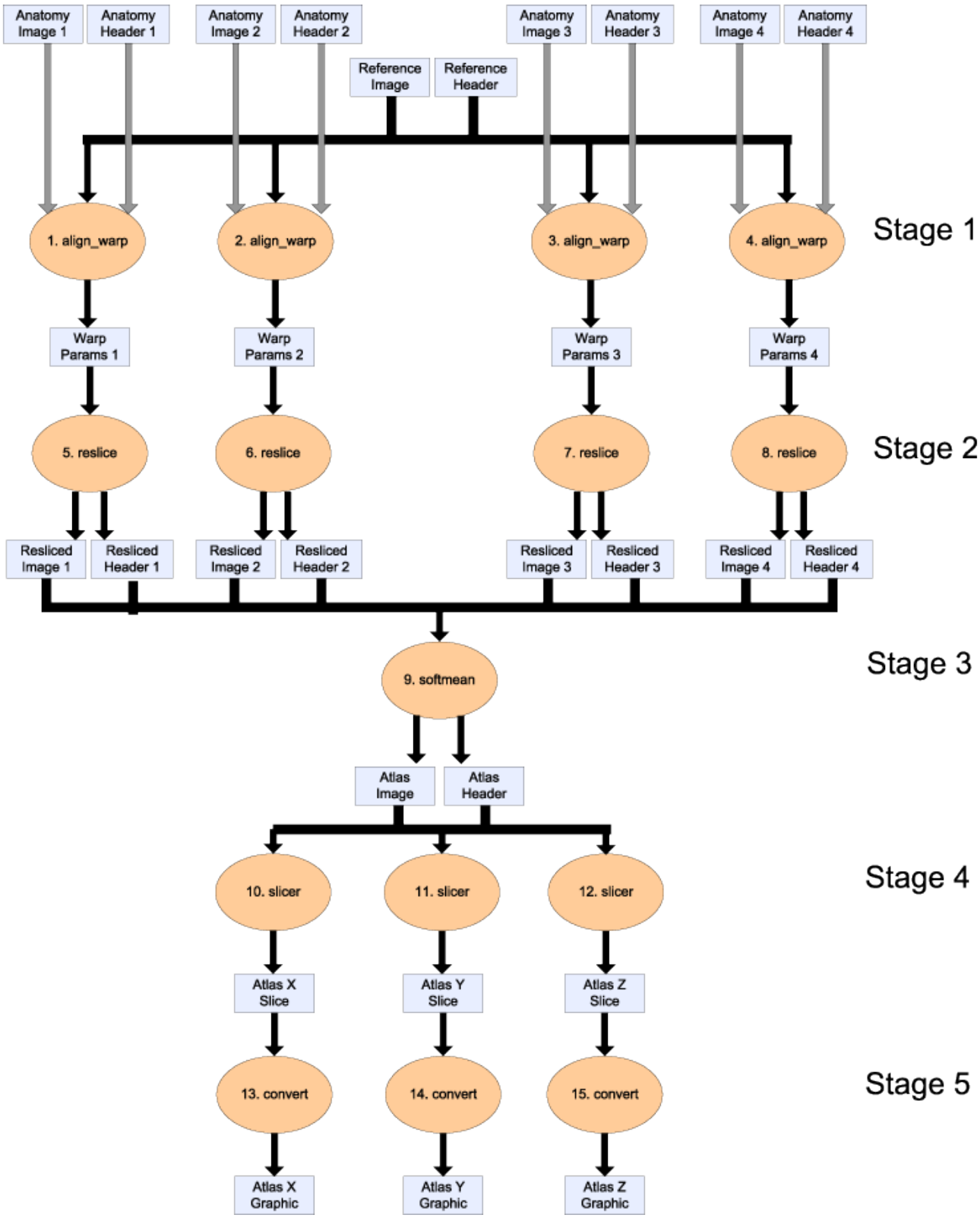

Figura 67: *Workflow* científico del First Provenance Challenge

3. El proceso **softmean** junta todas las imágenes anteriores en una sola imagen promedio de las mismas y obteniendo una imagen tridimensional de mayor calidad.

4. Cada uno de los procesos **slicer** separa la imagen tridimensional obtenida mediante softmean, en imágenes individuales de cada una de sus dimensiones (x, y, z). Estas imágenes en 2D se obtienen a partir del centro de la imagen 3D.

5. Finalmente, el proceso **convert** convierte la imagen de entrada obtenido con el proceso slicer a formato gif para un tratamiento más sencillo de la misma.

En la tabla 9, se proporciona información adicional sobre los procedimientos que componen el *workflow*.

Step	Procedure	Data Role	Item 1	Item 2	Item 3	Item 4
1	align_warp	Inputs	Anatomy Image 1	Anatomy Header 1	Reference Image	Reference Header
		Outputs	Warp Parameters 1			
		Parameters	-m 12 -q			
2	align_warp	Inputs	Anatomy Image 2	Anatomy Header 2	Reference Image	Reference Header
		Outputs	Warp Parameters 2			
		Parameters	-m 12 -q			
3	align_warp	Inputs	Anatomy Image 3	Anatomy Header 3	Reference Image	Reference Header
		Outputs	Warp Parameters 3			
		Parameters	-m 12 -q			
4	align_warp	Inputs	Anatomy Image 4	Anatomy Header 4	Reference Image	Reference Header
		Outputs	Warp Parameters 4			
		Parameters	-m 12 -q			
5	reslice	Inputs	Warp Parameters 1			
		Outputs	Resliced Image 1	Resliced Header 1		
		Parameters				
6	reslice	Inputs	Warp Parameters 2			
		Outputs	Resliced Image 2	Resliced Header 2		
		Parameters				
7	reslice	Inputs	Warp Parameters 3			
		Outputs	Resliced Image 3	Resliced Header 3		
		Parameters				
8	reslice	Inputs	Warp Parameters 4			
		Outputs	Resliced Image 4	Resliced Header 4		
		Parameters				
9	softmean	Inputs	Resliced Image 1	Resliced Header 1	Resliced Image 2	Resliced Header 2
		Inputs	Resliced Image 3	Resliced Header 3	Resliced Image 4	Resliced Header 4
		Outputs	Atlas Image	Atlas Header		

		Parameters	y null			
10	slicer	Inputs	Atlas Image	Atlas Header		
		Outputs	Atlas X Slice			
		Parameters	-x .5			
11	slicer	Inputs	Atlas Image	Atlas Header		
		Outputs	Atlas Y Slice			
		Parameters	-y .5			
12	slicer	Inputs	Atlas Image	Atlas Header		
		Outputs	Atlas Z Slice			
		Parameters	-z .5			
13	convert	Inputs	Atlas X Slice			
		Outputs	Atlas X Graphic			
		Parameters				
14	convert	Inputs	Atlas Y Slice			
		Outputs	Atlas Y Graphic			
		Parameters				
15	convert	Inputs	Atlas Z Slice			
		Outputs	Atlas Z Graphic			
		Parameters				

Tabla 9: Procedimientos que componen el *workflow* científico del First Provenance Challenge.

El resultado de la ejecución del *workflow* con los datos originales de entrada se puede observar en la figura 68.

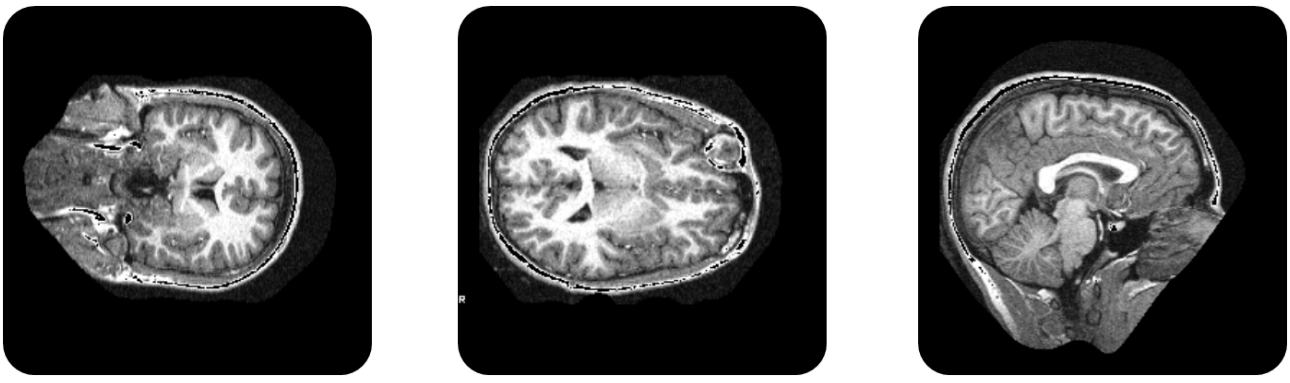

Figura 68: Imágenes cerebrales de alta resolución resultado de la ejecución del First Provenance Challenge.

X.5 - Modelado del problema

Para ilustrar el funcionamiento del sistema se ha modelado el *workflow* anterior utilizando el entorno de modelado propuesto, Renew, y un sistema de gestión de *workflows* científicos actual como es el caso de Taverna. De esta forma queremos demostrar la posibilidad de utilizar la infraestructura independientemente de la herramienta de modelado utilizada.

Modelado del problema con Renew

En la figura 69, se muestra el modelo del *workflow* científico correspondiente al *First Provenance Challenge* utilizando la herramienta Renew. Para dotar a la figura de una mayor claridad, en las etapas 1 y 2 sólo se incluye de forma completa la descripción de los procesos para la primera imagen. La representación es análoga para el resto de imágenes.

En lo que se refiere a las entradas, para demostrar la flexibilidad del sistema en lo que a la representación de los datos se refiere, se describen las entradas de diversas formas. Las dos primeras imágenes utilizan una descripción en la que se especifica el protocolo de transferencia a utilizar. La primera imagen es obtenida del *cluster* Hermes utilizando el protocolo *sftp* mientras que la segunda imagen es obtenida del *grid* Piregrid utilizando el protocolo *scp*. Nótese que no se describe la URI completa si no que falta la extensión que es proporcionada en los diferentes procesos. La tercera imagen es tomada del *cluster* Hermes utilizando una descripción independiente del protocolo de comunicación para indicar la ubicación de la misma, por tanto, cuando se despliegue el trabajo, el sistema utilizará el protocolo adecuado de acuerdo a la configuración del *Grid*. La cuarta imagen utiliza el mismo mecanismo de descripción pero es tomada del *grid* Aragrid. Por su parte, la imagen de referencia que es utilizada en todos los procesos de la primera etapa, es obtenida de la página web del grupo de investigación GIDHE, mostrando que no todos los ficheros deben estar en un *Grid*.

Asimismo, se pueden especificar diferentes aspectos como entradas del *workflow científicos*. En este caso, indicamos que Piregrid será el *grid* que se utilizará para desplegar el trabajo en la primera etapa. De la misma forma, se podría indicar el usuario utilizado para el despliegue, por ejemplo. Sin embargo, en nuestro caso hemos decidido poner el usuario directamente en cada tupla de descripción de trabajo. Un aspecto importante es que la URI que describe la imagen se utiliza tanto para los ficheros de entrada como para los ficheros de salida por lo que la salida correspondiente a cada imagen quedará ubicada junto a la entrada.

De esta forma, la etapa 1 es la encargada de ejecutar los 4 procesos `align_warp`. En este caso se ha utilizado un mecanismo asíncrono para desplegar el trabajo, realizando en primer lugar una operación `out` para encargar la ejecución del trabajo y posteriormente una operación `in` para recoger su finalización.

En la etapa 2 se realizan los procesos `reslice`. En esta etapa se ha optado por incluir directamente el *Grid* Hermes como infraestructura de ejecución en vez de tomarla de la salida. Otro aspecto importante es que la ubicación de los datos de entrada y salida se describe utilizando la salida de la etapa anterior lo que nos permite abstraer a esta etapa de la ubicación de los ficheros utilizados.

En la tercera etapa se juntan todas las imágenes de salida de la segunda etapa para realizar la media de las mismas. En este caso, como las imágenes proceden de diferentes infraestructuras, se indica directamente la ubicación de los ficheros de salida. Asimismo, en este caso no se indica ninguna infraestructura de computación por lo que el sistema decidirá de acuerdo a la política de emparejamiento competitivo la infraestructura de computación a utilizar.

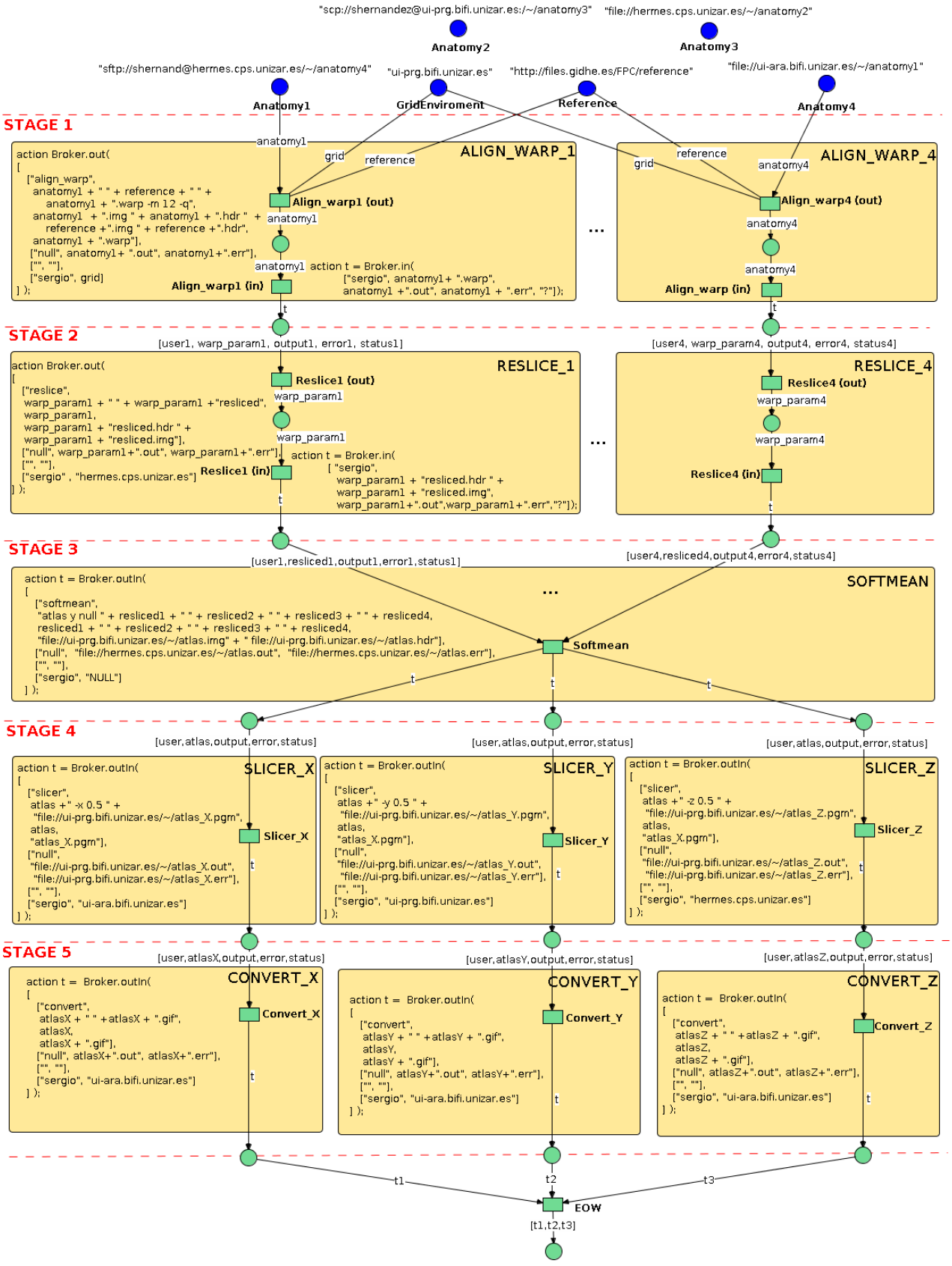

Figura 69: Modelo del *workflow* científico del *First Provenance Challenge* con Renew.

La cuarta etapa separa la imagen tridimensional en imágenes de cada una de las perspectivas. Podemos observar cómo cada una de las perspectivas de esta etapa se realiza en cada una de las tres infraestructuras de computación utilizadas, Aragrid, Piregrid y Hermes.

La quinta y última etapa del *workflow* convierte las imágenes anteriores a formato gif. Para esta última etapa se fija el *Grid* Aragrid como infraestructura de computación. De esta forma, cada infraestructura es la encargada de la ejecución de al menos una etapa.

Finalmente, se añade una transición con nombre `EOW` (*End Of Workflow*) que sirva para identificar el fin del *workflow*. Esta transición no es obligatoria pero se recomienda su utilización ya que permite detectar fácilmente que se ha llegado al final del problema.

Modelado del problema con Taverna

El modelo del *workflow* científico utilizando el sistema de gestión de *workflows* científicos Taverna puede observarse en la figura 70.

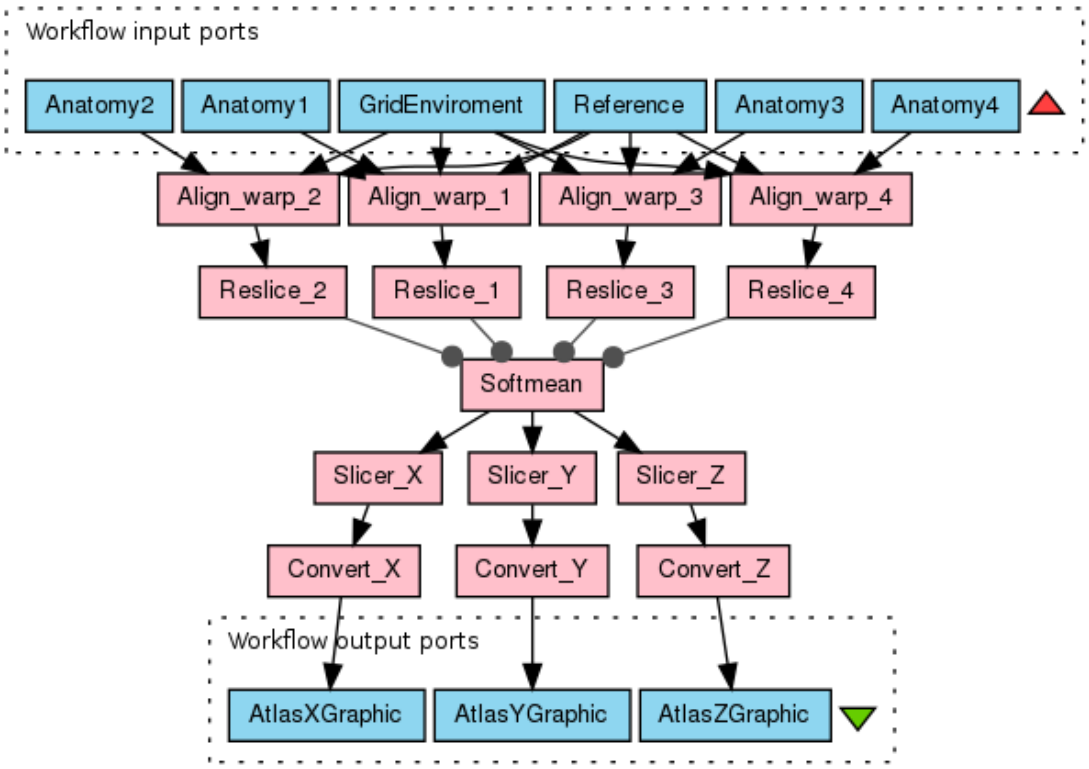

Figura 70: Modelo del *workflow* científico del *First Provenance Challenge* con Taverna.

En la parte superior del *workflow* aparecen las entradas del mismo, que son las mismas que en el caso anterior aunque la representación de Taverna sólo nos indica el nombre de la entrada, siendo el valor de la misma interno y no visible en el modelo.

De esta forma cada una de las cajas rosas representa la ejecución de cada trabajo. Estas cajas son a su vez otros *workflows* más sencillos que están compuestos por las mismas operaciones que en el caso del modelo con Renew. Por tanto, en la figura 66 sólo podemos observar la estructura del workflow y no el detalle de las operaciones realizadas en el mismo.

Finalmente, en la parte inferior del modelo se localizan las salidas del *workflow* que nos permiten recoger los resultados y detectar que el mismo ha finalizado.

La implementación de los diferentes *subworkflows* se puede realizar exactamente igual que en el modelo realizado con Renew, es decir, utilizando la clase `Broker`. Sin embargo, en este ejemplo presentamos otro modelo. El mismo corresponde a descomponer la clase `Broker` en sus diferentes tareas para utilizar la característica estándar de Taverna que nos permite invocar directamente un servicio web, en esta caso la interfaz de servicios web de WS-PTRLinda. En la figura 71, puede verse uno de los modelos de estos *subworkflows*, mientras que, el resto son análogos. En este caso, como el *broker* no proporciona la operación outIn, si no que esta es una característica propia de la clase `Broker`, se muestra la implementación de estos subworkflows usando una operación `out` seguida de una operación `in`.

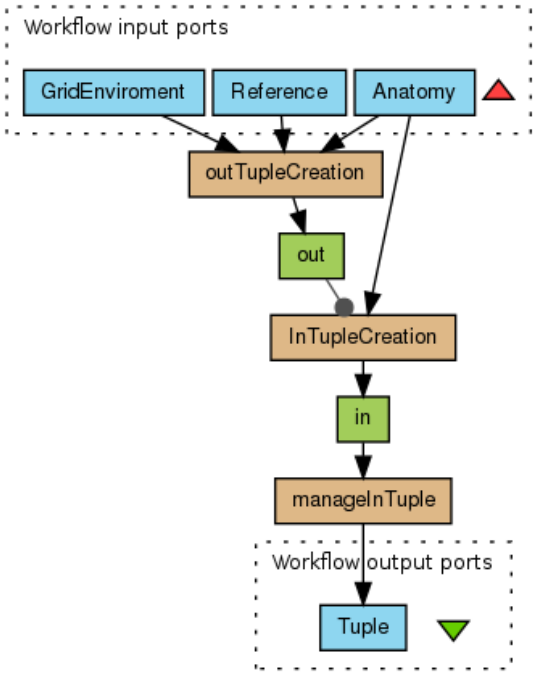

Figura 71: *Subworkflow* implementado la tarea `align_warp`

En primer lugar, se crea la tupla que se escribirá en el *broker* a partir de los parámetros de entrada. Esta operación la realiza el componente `createOutTuple`. Utilizando dicha tupla se realiza la operación de escritura `out` sobre WS-PTRLinda. El siguiente paso consiste en crear el patrón utilizado para recuperar la tupla que indica el fin del trabajo. El mismo es realizado por el componente `createInTuple`. Después se realiza la operación de lectura `in` que nos devuelve como resultado una tupla, la cual es gestionada por el componente `manageInTuple` para obtener el *log* de ejecución y proporcionar el resultado.

Glosario

- Amazon EC2 - *Amazon Elastic Compute Cloud* (Computación elástica en la nube de Amazon)
- API - *Application Programming Interface* (Interfaz de programación de aplicaciones)
- DAG - *Directed Acyclic Graph* (Grafo Acíclico Dirigido)
- HTTP - *Hypertext Transfer Protocol* (Protocolo de transferencia de hipertexto)
- JDL - *Job Description Language* (Lenguaje de descripción de trabajos)
- JSDL - *Job Submission Description Language* (Lenguaje de descripción para el envío trabajos)
- MPI - *Message Passing Interface* (Interfaz de paso de mensajes)
- N-w-N - *Nets-within-Nets* (Redes en redes)
- P2P - *Peer-to-peer* (Red entre iguales)
- QoS - *Quality of Service* (Calidad de Servicio)
- Renew - *Reference Nets Workshop* (Taller de redes de referencia)
- RPC - *Remote Procedure Call* (Llamada a procedimiento remoto)
- REST - *Representational State Transfer* (Transferencia de estado representacional)
- SOA - *Service-Oriented Architecture* (Arquitectura orientada a servicios)
- SOAP - *Simple Object Access Protocol* (Protocolo de acceso a objeto)
- SOC – *Service-Oriented Computing* (Computación orientada a servicios)
- SMTP - *Simple Mail Transfer Protocol* (Protocolo de transferencia de correo electrónico)
- URL - *Uniform Resource Locator* (Localizador uniforme de recursos)
- URI - *Uniform Resource Identifier* (Identificador uniforme de recursos)
- VO - *Virtual Organization* (Organización Virtual)
- XML - *eXtensible Markup Language* (Lenguaje de marcado extensible)
- WS - *Web Service* (Servicio Web)
- WSDL - *Web Services Description Language* (Lenguaje de descripción de servicios Web)

Bibliografía

Referencias bibliográficas

[B1] I. Taylor, E. Deelman, D. Gannon and M. Shiedls. *Workflows for e-Science*. Introduction, pp. 1-8, 2006.

[B2] J. Fabra, P. Alvarez, J.A. Bañares and J. Ezpeleta, *DENEB: A Platformn for the Development and Execution of Interoperable Dynamic Web Processes*. DOI: 10.1002/cpe.1795. Concurrency and Computation: Practice and Experience, 2011.

[B3] E. Elmroth and J. Tordsson, *An interoperable, standard-based Grid resource broker and job submission service*. First International Conference on e-Science and Grid Computing (E-SCIENCE 2005), Melbourne, Australia, pp. 212-220, 2005.

[B4] D. Gelernter: *Generative communication in Linda*. ACM Transactions on Programming Languages and Systems 7, pp. 80–121, 1985.

[B5] N. Carriero and D. Gelernter, *Linda in context*. Communications of the ACM, Vol.32, Num.4, pp. 444-458, 1989.

[B6] O. Kummer, *Introduction to petri nets and reference nets*. Sozionik Aktuell, Num. 1, pp. 19, 2001.

[B7] O. Kummer and F. Wienberg, *Renew - the Reference Net Workshop. Tool Demonstrations*. In 21st International Conference on Application and Theory of Petri Nets, pp. 87-89, 2000.

[B8] De Roure, D., Goble, C. and Stevens, R. *The Design and Realisation of the myExperiment Virtual Research Environment for Social Sharing of Workflows*. Future Generation Computer Systems 25, pp. 561-567, 2009

[B9] J. Mengual, *WS-PTRLinda: un sistema de coordinación temporizado y persistente basado en tecnologías de servicios web*. Proyecto Fin de Carrera, Universidad de Zaragoza, 2010.

[B10] J. Fabra, P. Álvarez, J.A. Bañares, and J. Ezpeleta. *RLinda: a Petri net based implementation of the Linda coordination paradigm for Web services interactions*. In 7th International Conference on Electronic Commerce and Web Technologies - EC-Web'06, volume 4082 of LNCS, pages 183-192. Springer Verlag, Septiembre 2006.

[B11] M. Rahman, R. Ranjan, and R. Buyya, *A Taxonomy of Autonomic Application Management in Grids*, Proceedings of the The 16th International Conference on Parallel and Distributed Systems (ICPADS 2010, IEEE CS Press, USA), Shanghai, China, December 8-10, 2010.

[B12] J. Yu and R. Buyya, *A Taxonomy of Workflow Management Systems for Grid Computing*, Journal of Grid Computing, Volume 3, Numbers 3-4, Pages: 171-200, Springer Science+Business Media B.V., New York, USA, Sept. 2005.

[B13] J. Yu and R. Buyya, *A novel architecture for realizing Grid Workflows using Tuple Spaces*. In 5th IEEE/ACM International Workshop on Grid Computing (GRID 2004), Pittsburgh, USA, pp. 119-128, 2004.

[B14] R. Sharma, V.K. Soni and M.K. Mishra and P. Bhuyan, *A survey of job scheduling and resource management in grid computing*. World Academy of Science, Engineering and Technology, Issue 64, pp. 461-466, 2010.

[B15] M. Wieczorek, R. Prodan and T. Fahringer, *Scheduling of Scientific workflows in the ASKALON grid environment*. ACM SIGMOD Record, Vol. 34, Issue 3, pp. 56.62, 2005.

[B16] N. Sample, P. Keyani and G. Wiederhold, *Scheduling under uncertainty: Planning for the ubiquitous Grid*. In 5th International Conference on Coordination Models and Languages (COORDINATION 2002), York, UK, Lecture Notes in Computer Science, Vol. 2315, pp. 300-316, 2002.

[B17] B. Allcock, J. Bester, J. Bresnahn, A. Chervenak, I. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnel and S. Tuecke. *Data management and transfer in high-performance computational grid environments*. Parallel Computing, vol. pp. 749–771, 2002.

[B18] W. Aalst, *The application of petri nets to workflow management*. J. Circ. Syst. Comput. 8, 1, 21–66, 1998.

[B19] D. Fiedler, K. Walcott, T. Richardson, G. Kapfhammer, A. Amer, and P. Chrysanthis. *SETTLE: A Tuple Space Benchmarking and Testing Framework*. Presented at the Ninth Jini Community Meeting, Chicago, Illinois, Octubre 2005.

Referencias Web

[W1] Sitio web del grupo de integración de sistemas distribuidos y heterogéneos (GIDHE). http://www.gidhe.es, 2011.

[W2] Wiki del cluster Hermes. http://web.hermes.cps.unizar.es/wiki/index.php/Inicio, 2011.

[W3] Sitio web del Instituto de Investigación en Ingeniería de Aragón (I3A). http://i3a.unizar.es/, 2011

[W4] Proyecto Condor. http://www.cs.wisc.edu/condor/, 2011.

[W5] Proyecto Piregrid. http://www.piregrid.eu/?idioma=espanol, 2011.

[W6] Proyecto Aragrid. http://www.aragrid.es/, 2011.

[W7] Proyecto gLite. http://glite.cern.ch/, 2011.

[W8] Proyecto Taverna. http://www.taverna.org.uk/, 2011.

[W9] Amazon Elastic Compute Cloud (Amazon EC2). http://aws.amazon.com/es/ec2/,2011.

[W10] Comunidad científica MyExperiment. http://www.myexperiment.org/, 2011.

[W11] Proyecto SALAMI (*Structural Analysis of Large Amounts of Music Information*).

http://salami.music.mcgill.ca/, 2011.

[W12] Proyecto Askalon. http://www.askalon.org/, 2011.

[W13] Proyecto GridBus. http://www.cloudbus.org/workflow/, 2011.

[W14] Proyecto Globus. http://www.globus.org/, 2011.

[W15] Proyecto Kepler. https://kepler-project.org/, 2011.

[W16] Proyecto Pegasus. http://pegasus.isi.edu/, 2011.

[W17] Proyecto Triana. http://www.trianacode.org/, 2011.

[W18] Proyecto UNICORE. http://www.unicore.eu/, 2011.

[W19] *Job Description Language Attributes Specification*. https://edms.cern.ch/document/590869, 2011.

[W20] *Job Submission Description Language Specification*. http://www.gridforum.org/documents/GFD.56.pdf, 2011.

[W21] Proyecto *Rule Markup Language* (RuleML). http://ruleml.org/, 2011.

[W22] Sitio web del *Open Grid Forum*. http://www.ogf.org/, 2011.

[W23] RFC 3986 - Especificación del estándar *Uniform Resource Identifier* (URI). http://tools.ietf.org/html/rfc3986, 2011.

[W24] Sitio web del *First Provenance Challenge*. http://twiki.ipaw.info/bin/view/Challenge/FirstProvenanceChallenge, 2011.

[W25] Centro de datos de imágenes de resonancias magnéticas funcionales. http://www.fmridc.org/, 2011

[W26] Suite AIR: Herramientas de tratamiento de imágenes anatómicas tridimensionales. http://air.bmap.ucla.edu/AIR5/index.html, 2011

[W27] Suite FSL. Herramientas de tratamiento de imágenes anatómicas tridimensionales. http://www.fmrib.ox.ac.uk/fsl/, 2011

[W28] Wiki utilizada para gestionar la documentación del proyecto. http://www.gidhe.es/shernandez/doku.php, 2011.

[W29] Proyecto DokuWiki. http://www.dokuwiki.org/es:dokuwiki, 2011.

[W30] Google Docs. https://docs.google.com/,2011.

[W31] Skype. http://www.skype.com/intl/es/home, 2011.

[W32] Subversion. http://subversion.apache.org/, 2011.

[W33] Dropbox. http://www.dropbox.com/, 2011.

[W34] Proyecto Apache ANT. http://ant.apache.org/, 2011.

[W35] Fundación Eclipse. http://www.eclipse.org/, 2011.

[W36] RFC 3501 describiendo el protocolo *Internet Message Access Protocol* (IMAP) http://tools.ietf.org/html/rfc3501, 2011.

[W37] Página del lenguaje de programación JAVA. http://www.oracle.com/technetwork/java/javase/downloads/index.html, 2011.

[W38] Biblioteca JavaMail. http://www.oracle.com/technetwork/java/javamail/index.html, 2011.

[W39] Biblioteca JDOM. http://www.jdom.org/, 2011.

[W40] Proyecto JSch. http://www.jcraft.com/jsch/, 2011.

[W41] Proyecto *Object Oriented jDREW* (OO jDREW). http://ruleml.org/oojdrew/, 2011.

[W42] Proyecto oXygen. http://www.oxygenxml.com/, 2011.

[W43] RFC 3501 describiendo el protocolo *Secure Shell* (SSH). http://www.ietf.org/rfc/rfc4251.txt, 2011.

[W44] Definición del estándar *eXtensible Markup Language* (XML). http://www.w3.org/TR/2006/REC-xml11-20060816/, 2011.

[W45] Página para comprobar la versión instalada de JAVA. http://www.java.com/es/download/installed.jsp, 2011.

[W46] Página del sistema gestor de base de datos MySQL. http://www.mysql.com/, 2011.

[W47] Manual de usuario de la herramienta Renew. http://www.informatik.uni-hamburg.de/TGI/renew/renew.pdf, 2011

[W48] *Right Scale Cloud Computing Management Platform*. http://www.rightscale.com/, 2011.

[W49] Página del *International Provenance and Annotation Workshop* (IPAW). http://www.ipaw.info/, 2011.

[W50] *Second Provenance Challenge*. http://twiki.ipaw.info/bin/view/Challenge/SecondProvenanceChallenge, 2011.

[W51] *Third Provenance Challenge*. http://twiki.ipaw.info/bin/view/Challenge/ThirdProvenanceChallenge, 2011.

[W52] *Fourth Provenance Challenge*. http://twiki.ipaw.info/bin/view/Challenge/FourthProvenanceChallenge, 2011.

[W53] *Open Provenance Model*, http://openprovenance.org/, 2011.