DEPARTAMENTO DE ARQUITECTURA Y TECNOLOGÍA DE SISTEMAS INFORMÁTICOS

Facultad de Informática
Universidad Politécnica de Madrid

TESIS DOCTORAL

ARQUITECTURA DISTRIBUIDA DE CONTROL PARA SISTEMAS CON CAPACIDADES DE DATA MINING

Autor
José María Peña Sánchez
Licenciado en Informática

Directores
Ernestina Menasalvas Ruiz
Doctora en Informática

Pedro de Miguel Anasagasti
Doctor Ingeniero Industrial

Año: 2001

A todos aquellos que creyeron que la terminaría....
... así como a aquellos que no lo hicieron.

A ambos debo la motivación para haber llegado hasta aquí.

Agradecimientos

Nunca creí que me fuera tan difícil poner en orden mis ideas para escribir esta parte. La razón es que debo muchas cosas a tanta gente que la sensación de dejarme a alguna de estas personas sin mencionar es en extremo embarazosa. Antes de nada, y para salvaguardarme de posibles críticas, soy consciente que habrá personas que por un desgraciado despiste no mencionaré, espero que aquellos puedan perdonarme,... alguna vez.

En primer lugar, me gustaría agradecer a mis padres y hermano Luis su apoyo a lo largo de tanto tiempo y por haber hecho posible que me dedicara a lo que me gusta, facilitándome el camino.

Mis compañeros de carrera, que a lo largo de varios años hemos compartido tantas cosas, les he de agradecer lo mucho que de ellos he aprendido y su ayuda en muchas más ocasiones de las que ellos se pueden llegar a imaginan. Muchas gracias a Andrés, Arturo, Borja, Cristian, Sole, Myriam, Alberto, Vicente, Paco, Javi Crespo, Sam y otros tantos más.

Junto con esta 'vieja guardia' otros muchos amigos dentro de la Facultad me han prestado ayuda en muy diferentes facetas, mi primo Alberto o Juanfran son sólo alguno de ellos, pero también he de dar las gracias a Rojas, Paquito, Lola, Claudio, Nacho, Oscar Marbán, Oscar Delgado, Fernando, Javi Soriano, Salva, Javi y Luna entre otros.

Los antes mis profesores y ahora mis compañeros han sido también una pieza fundamental en mi trabajo durante estos últimos años. A ellos debo cosas tanto profesional como personalmente. Gracias a Santiago, Nicolás, Javier y Julio, y más recientemente a Almudena, Antonio, María, Fernando, Fran, Juan, Paco y Victor. Una mención especial reservo para Covadonga a quien le debo muchas cosas.

Fuera del ámbito de la Universidad debo dar las gracias a muchos muy buenos amigos, tanto de aquí cerca: Toño, Iñaki, Nacho, Ali, Gema o Quique como de mucho más lejos: Aracely, Fazel, Sylvain o Alan. A todos vosotros os he de agradecer ese apoyo incondicional que me habéis dado en las horas más bajas....habéis sido fundamentales.

Para el final he querido reservar a mis dos directores de Tesis. A Pedro le debo agradecer la insistencia con la que me ha perseguido para que terminase este trabajo y cómo me ha facilitado la integración en el pequeño grupo que formamos en la asignatura. Por último quiero agradecer a

Ernestina....bueno tantas cosas. Probablemente ella sea la principal 'culpable' de que al final haya conseguido concluir este trabajo, asimismo le debo a su ejemplar dedicación a su trabajo gran parte de mi vocación y a su excepcional carácter mi más sincera amistad. Muchísimas gracias.

José María Peña Sánchez
Madrid, 11 de Diciembre de 2000

Resumen

Data Mining o KDD son términos que designan las técnicas de análisis de datos para la búsqueda de patrones ocultos en los mismos. Estas técnicas se usan sobre bases de datos con millones de registros y centenares o miles de atributos por registro y consisten en la aplicación de diferentes procesos de preparación de datos, algoritmos de análisis y técnicas de presentación de resultados. Las aplicaciones y sistemas necesarios para la realización de estas tareas en unos plazos razonables requieren de un uso especialmente eficiente de los recursos disponibles (por ejemplo, CPU, memoria o almacenamiento secundario).

Dentro de este entorno, la computación distribuida posibilita el reparto de carga computacional entre varios nodos, usando los recursos locales (memoria y disco) de forma conjunta. Sobre un escenario de estaciones de trabajo heterogéneas y dando soporte a todas las diferentes técnicas, algoritmos e implementaciones aportadas por los investigadores en *Data Mining* la complejidad de estas aplicaciones es su principal característica. Este problema se agrava cuando varios usuarios hacen uso del sistema en paralelo, de forma que varias consultas son formuladas simultáneamente.

Como salida a esta problemática es necesario un control de los recursos de los nodos, una priorización de las tareas del sistema, una distribución inteligente de la carga y, en resumen, un conjunto de decisiones relativas al rendimiento del sistema que se han denominado **decisiones de control**.

Esta tesis propone una solución a la problemática de control de los sistemas de distribuidos de *Data Mining* en base a dos elementos: (i) por un lado una arquitectura distribuida de control, descrita y formalizada de forma genérica y (ii) un diseño de un sistema de *Data Mining* distribuido sobre dicha arquitectura.

Abstract

Data Mining and KDD are both terms that define data analysis techniques that are able to find hidden patterns in the data. These techniques are used to explore databases with millions of records and hundreds or thousands of attributes per record. These techniques group together data preparation processes, analysis algorithms and result presentation issues. The efficient achievement of these tasks requires applications and systems with specially resource management capabilities (for instance: CPU, memory and secondary management).

On this field, load balancing and the combination of local and remote resources provided by distributed computation technology is a very important factor. But the use of Data Mining techniques running on a cluster of independent workstations is a non-trivial problem. The design of new algorithms and the definition of new techniques in this field also adds mode complexity of Data Mining systems in order to be able to support them as they appear. This complexity increases when multiple user access to the system is provided in parallel.

As a solution to tame this environment, issues like resource management, task priority scheduling, intelligent load balancing and, in general, system performance criteria are necessary. This kind of issues have been called **control decisions**.

This Thesis propose a solution of control problems in distributed data mining systems. This solutions is provided by: (i) a generic distributed control architecture, formally defined and (ii) a design of a distributed data mining system defined over this this architecture.

Índice General

INTRODUCCIÓN

Desde su definición a principios de la década de los noventa, el término KDD (*knowledge discovery in databases*) o descubrimiento de información en bases de datos ha descrito un campo que ha evolucionado rápidamente. Partiendo de unos orígenes fuertemente relacionados con la Inteligencia Artificial y la estadística, se trataba de una disciplina experimental en la que algoritmos de aprendizaje se aplicaban sobre centenares de registros para determinar dependencias, relaciones o reglas existentes entre esos datos. Posteriormente, y debido al incremento exponencial de la información almacenada por las organizaciones, ciertos sectores comerciales, así como investigadores de otros campos vieron el increíble potencial que técnicas de KDD les proporcionaban para analizar datos históricos de los clientes de una compañía o los datos recopilados por los sensores en un experimento científico o en un proceso industrial. Estos nuevos campos de aplicación exigían herramientas capaces de analizar miles o millones de registros para extraer de ellos información útil, oculta entre el inmenso volumen de datos.

Estos nuevos escenarios de aplicación diferían de los casos en los cuales se había comenzado a trabajar en dos factores, en primer lugar, los órdenes de magnitud de los datos tratados y en segundo lugar la complejidad de los mismos. Los datos reales tienen información incompleta, errónea, redundante, la aplicación de los algoritmos, a su vez, requiere datos adecuados para la ejecución de los mismos y una vez extraídos los resultados esos han de ser presentados al usuario de una forma apropiada. Las herramientas desarrolladas desde entonces han proporcionado un soporte completo de todas las etapas del proceso: preproceso, análisis y postproceso. Comercialmente, y a partir de este momento, al proceso global se ha denominado *Data Mining*, en lugar del nombre tradicional de KDD.

Dentro del marco actual del campo de *Data Mining* surgen tres claras líneas de investigación y desarrollo. Por un lado se encuentran los investigadores que, en la línea tradicional de este campo, desarrollan nuevos algoritmos y teorías para análisis de datos. Por otro lado, la aplicación de las técnicas de *Data Mining* a problemas reales implica la integración de los mecanismos de análisis de datos con la experiencia dentro del dominio de aplicación y la resolución de las peculiaridades de los datos de cada dominio. Equidistante entre estas dos líneas de desarrollo se encuentran los

avances en la definición de nuevas arquitecturas y nuevos sistemas de *Data Mining*. Estos sistemas darán soporte a las diferentes técnicas y algoritmos desarrollados por la primera de las líneas de investigación y servirá como herramienta para los expertos que definan soluciones para un campo determinado.

Características Actuales de los Sistemas de *Data Mining*

En el contexto de herramientas de *Data Mining* existen numerosas soluciones actualmente implantadas. El interés comercial sobre sistemas de *Data Mining* se encuentra en demanda creciente y numerosas empresas han desarrollado potentes aplicaciones en este campo. Dos características fundamentales de los sistemas actuales son la *integración* con los Sistemas Gestores de Bases de Datos (SGBD) y la *extensibilidad* de sus funcionalidades. La primera, la *integración*, permite la realización de ciertas tareas de análisis o preproceso dentro de los SGBD, aumentando considerablemente la eficiencia del sistema en general. Por su parte, la *extensibilidad* permite la ampliación del conjunto de operaciones y técnicas soportadas por el sistema, de forma que un nuevo algoritmo, necesario para la resolución de un problema particular puede ser añadido al sistema fácilmente. Estas dos características son elementos fundamentales de los sistemas de última generación, tales como Clementine/SPSS de ISL o Intelligent Miner de IBM y deben de ser mantenidas en los nuevos sistemas.

En los últimos años los procesos de *Data Mining* han sido incluidos dentro del grupo de tareas que requieren computación masiva. Muestra de ello es que un porcentaje representativo de las instalaciones de supercomputación existentes se encuentran dedicadas al tándem *Data Warehousing-Data Mining*. Esto ha hecho que se desarrollen mecanismos de alto rendimiento para resolver problemas de *Data Mining*. Estos mecanismos se encuentra englobados en lo que se denomina *Data Mining distribuido* o *Data Mining paralelo*. El desarrollo de nuevos sistemas distribuidos de *Data Mining* está surgiendo ahora como solución tanto a estos requisitos de más altas prestaciones como a la naturaleza distribuida de muchas fuentes de datos, que por motivos tecnológicos o de privacidad no pueden ser centralizados en única base de datos para su análisis.

Objetivos del Trabajo

Con la intención de mantener las características que actualmente se encuentran resaltadas como fundamentales en los sistemas de *Data Mining* (*integración* y *extensibilidad*) y de incluir las nuevas tendencias en dichos sistemas, como es la *distribución* se pretende definir una nueva arqui-

tectura genérica, distribuida y basada, en componentes. Esta nueva arquitectura resalta la faceta de la *flexibilidad* de control, definida como la capacidad para modificar y optimizar el comportamiento del sistema de forma fácil y flexible por medio de lo que se han denominado *políticas de control*. Dichas políticas permiten describir las decisiones relacionadas con el reparto de recursos, arbitraje de prioridades o equilibrado de carga, entre otras.

Todo sistema complejo está compuesto por tareas meramente operacionales y por decisiones de control que rigen cómo y cuándo son realizadas dichas tareas. Alterando únicamente las decisiones de control (representadas por medio de las *políticas*) el comportamiento del sistema y su rendimiento en determinadas circunstancias es modificado. En aplicaciones tales como los sistemas de *Data Mining* que han de procesar grandes volúmenes de datos, las decisiones de control orientadas a la eficiencia del mismo son de vital importancia. Tradicionalmente, la división entre estas dos facetas no se analiza. En los pocos casos en los que se realiza un análisis concienzudo entre tareas operacionales (mecanismos) y decisiones de control (políticas), este estudio se plasma en un diseño en el cual estas dos facetas están definidas estáticamente.

La principal aportación de esta tesis se centra en la definición de un nuevo modelo de aplicación de las decisiones de control que permita su modificación dinámicamente, sin rediseñar ni implementar de nuevo el sistema, incluso durante su ejecución. La aplicación de una nueva arquitectura basada en dicho modelo de políticas de control permite la modificación y optimización del sistema por medio del refinamiento progresivo de las políticas y facilita la experimentación de diferentes estrategias en la resolución de dichas decisiones. Este enfoque es completamente novedoso dentro de un campo en rápida evolución como es el de los sistemas de *Data Mining*.

En base a esta arquitectura genérica se replantea el diseño de un sistema de *Data Mining integrado*, *extensible* y *distribuido* cuyo control se encuentre definido de forma *flexible* por medio de *políticas de control*. Asimismo, se plantean una serie de pautas de implantación, tanto de la arquitectura como del sistema.

Organización del Trabajo

El contenido de el resto de esta Tesis se encuentra organizado de la siguiente forma:

- La primera parte de la Tesis recorre el trabajo realizado hasta el momento en una serie de campos, en la intersección de los cuales se encuadra tanto el problema planteado como la solución propuesta.

- ❍ El **Capítulo 1** repasa el estado actual del proceso de KDD, de los sistemas de *Data Mining* actuales y de los prototipos experimentales en desarrollo.
- ❍ El **Capítulo 2** muestra las diferentes tecnologías de desarrollo de sistemas distribuidos, centrándose principalmente en las de objetos distribuidos y en particular la tecnología CORBA de OMG.
- ❍ El **Capítulo 3** presenta la técnicas de Agentes Software como mecanismo de desarrollo de componentes autónomos y cooperantes. Este capítulo complementa al anterior a la hora de dar un enfoque más formal y menos tecnológico al desarrollo de componentes distribuidos.

❒ La segunda parte de este trabajo presenta el problema tratado y la solución planteada como respuesta.

- ❍ En el **Capítulo 4** se realiza un estudio detallado del problema a resolver, haciendo hincapié en los términos usados para definir los objetivos: *integración*, *extensibilidad*, *distribución* y *flexibilidad de control*.
- ❍ El **Capítulo 5** expone la arquitectura distribuida de control MOIRAE. Esta arquitectura genérica da soporte a cualquier sistema distribuido cuya configuración de control sea modificable de forma dinámica.
- ❍ El **Capítulo 6** plantea la aplicación de la arquitectura MOIRAE al diseño de un sistema distribuido de *Data Mining* que satisfaga los requisitos detallados en los objetivos.
- ❍ En el **Capítulo 7** se proporcionan unas pautas generales de implementación y desarrollo de las cualidades más relevantes tanto de la arquitectura como del sistema.
- ❍ El **Capítulo 8** recoge los escenarios experimentales que se han desarrollado. Dichos escenarios evalúan las capacidades derivadas de aplicar la arquitectura propuesta a problemas de control de entornos distribuidos.

❒ Finalmente, la última parte del trabajo recoge, en el **Capítulo 9** las conclusiones extraídas de esta investigación y en el **Capítulo 10** las futuras líneas de investigación abiertas tras la realización de este trabajo.

Parte I

ESTADO DE LA CUESTIÓN

Capítulo 1

SISTEMAS DE DATA MINING Y KDD

Índice General

1.1 Introducción

Durante los últimos años el mundo empresarial e industrial ha abordado el reto de la automatización de sus procesos internos de negocio. De esta forma, las compañías han desarrollado aplicaciones para asistir el proceso de producción industrial, las tareas de facturación, el control

de *stocks*, etc. En el núcleo de todas estas aplicaciones se encuentran los sistemas de información de las organizaciones. Estos sistemas proporcionan soporte para mantener la información que la organización requiere para realizar sus procesos internos y de mantenimiento. Por lo general, la tecnología que soporta todos estos procesos se basa en los sistemas gestores de bases de datos (SGBD), la mayoría de los cuales en la actualidad son sistemas gestores de bases de datos relacionales (SGBDR).

Como resultado del proceso antes citado, a lo largo de los últimos años se ha presentado un nuevo reto. Si bien el proceso de automatización de los mecanismos internos de las compañías ha sido resuelto en mayor y menor medida por las grandes organizaciones, actualmente se han presentado nuevas necesidades a satisfacer. Los volúmenes de información almacenados a lo largo de años por las organizaciones representan una herramienta potencialmente muy útil para comprender el funcionamiento de la organización dentro de su entorno. El análisis de la información recogida por una empresa puede explicar los patrones de compra de sus clientes, las características que determinan el éxito de sus productos o de las campañas de *marketing* asociadas, etc. Sin embargo, el valor potencial de estos datos no se encuentra disponible a no ser que dichos datos sean limpiados y analizados. Este proceso de análisis ha de ser realizado por medio de algún mecanismo capaz de extraer a partir de grandes volúmenes de datos la información útil que en ellos se oculta. La respuesta a estas necesidades se encuentra representada por los términos KDD y *Data Mining*.

1.2 KDD y *Data Mining*

A menudo los términos KDD y *Data Mining* se tratan de forma indistinta para hacer referencia al proceso de análisis de datos. Desde el punto de vista formal se podría indicar que KDD es tradicionalmente el proceso global de análisis de información que cubre desde que se seleccionan los datos a tratar hasta que el usuario que solicita la información obtiene la información y está satisfecho con los resultados. Por su parte *Data Mining* se restringe a una sola de las etapas del proceso anterior encargada de extraer modelos de conocimiento que han de ser presentados y verificados por el usuario en base a unos datos previamente preparados.

La confusión de ambos términos está especialmente presente dentro del mundo empresarial en el cual se denomina *Data Mining* al proceso completo de análisis (lo que formalmente es KDD). Sin embargo, dentro de los círculos académicos más puristas, se establecen de forma clara la división entre ambos términos. A lo largo de este trabajo se ha hecho uso de *Data Mining* como término más comúnmente aceptado para hacer referencia al proceso de análisis que ahora se definirá.

1.2.1 Definición de KDD

El término KDD fue acuñado formalmente por Piatetsky-Shapiro en 1991 [PS91b] el cual definió el término como:

> **Definición** PROCESO DE KDD: El proceso no trivial de identificación de patrones válidos, novedosos, potencialmente útiles y comprensibles en los datos.

Por **datos** se entiende un conjunto de hechos y patrones que se expresan en algún lenguaje. Por lo general el formato de representación de estos datos son tuplas de una base de datos. El término **patrones** designa a los modelo extraídos en base a los datos y que representan una descripción de alto nivel de los datos de los que provienen. Los patrones extraídos pueden representarse como reglas, árboles, grafos o cualquier otra forma de representar información de alto nivel.

El término **proceso**, por otra parte, implica que KDD se compone de varios pasos que incluyen la selección de los datos, discretización de valores, refinamiento de los datos, búsqueda de los patrones, evaluación del conocimiento extraído y posiblemente iteraciones sobre todas las fases para el posterior refinado. El proceso es no **trivial** puesto que no es un proceso obvio sino que requiere de distintas fases en distintas iteraciones. Por otra parte, los patrones que se descubran deberán ser **válidos** con algún grado de certeza y **novedosos** al menos para el sistema que los descubre y preferiblemente para el usuario que los requiere. Por último, es interesante destacar que el conocimiento extraído deberá proporcionar algún beneficio al usuario o tarea que requiere de ese conocimiento y por supuesto debe ser **comprensible** bien directamente o bien después de algún procesamiento por el usuario final.

1.2.2 Fases del proceso de KDD

A lo largo de la evolución del término KDD han existido dos distintas interpretaciones de las diferentes fases en las que dicho proceso se descompone.

1.2.2.1 Enfoque tradicional

Tradicionalmente [PS91a] el proceso se ha encontrado dividido en cinco fases:

① **Selección**: Fase en la que se determinan los datos que van a ser tratados a lo largo del proceso. En esta fase se seleccionan de los datos almacenados en diferentes bases de datos, ficheros o cualquier otro soporte de datos, que van a ser objeto de estudio. Esta fase es de vital importancia porque una mala selección implica partir de unos datos que no proporcionarán resultado alguno.

② **Limpieza**: Como resultado del proceso anterior los datos de estudio contienen información nula, no válida o datos que al ser combinados contienen errores e inconsistencias. Esta fase elimina los problemas derivados de la combinación de las fuentes de datos realizada anteriormente.

③ **Codificación**: Debido a que los mecanismos utilizados para extraer los patrones plantean una serie de restricciones en relación a los datos sobre los que se aplican, resulta necesario realizar una serie de transformaciones o adaptaciones previas de los datos. Ejemplos de transformaciones de preparación puede ser discretización de valores continuos o generalización en base a jerarquías de conceptos.

④ ***Data Mining*** : Esta fase constituye el eje del proceso, su cometido es extraer modelos en base a algoritmos y técnicas de análisis de los datos proporcionados.

⑤ **Interpretación y Evaluación**: Una vez extraídos los modelos estos han de ser validados por el usuario de forma que se verifique que cumplen las características requeridas de los resultados del proceso. Esta fase comprende mecanismos de visualización o representación de resultados para que el operador humano sea capaz de interpretar los resultados.

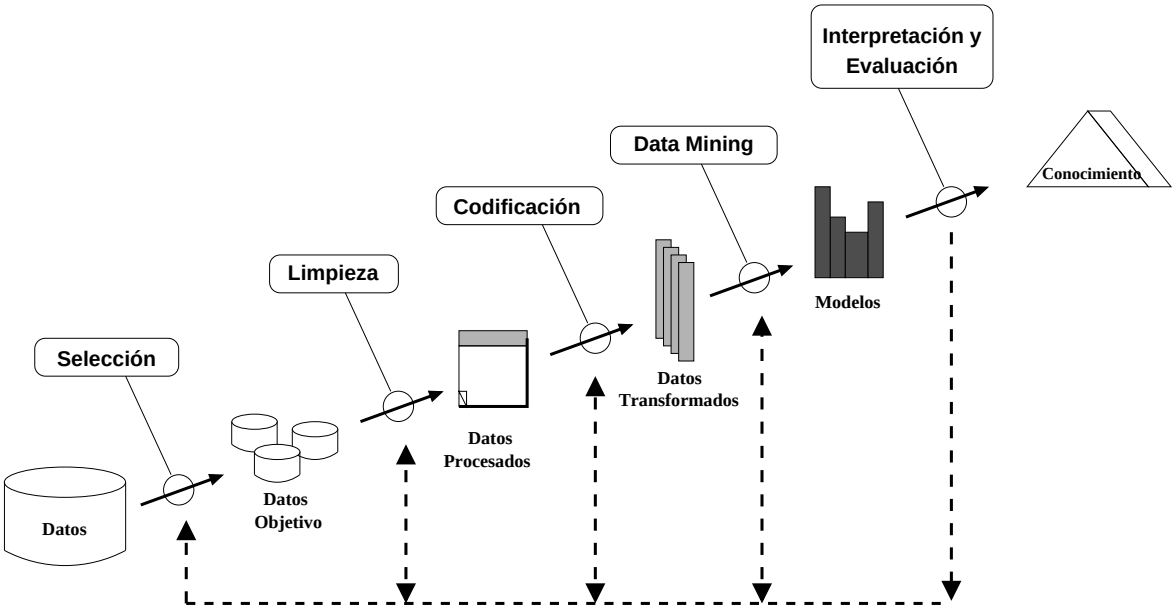


Figura 1.1: Fases del proceso KDD [PS91a]

Esta división en procesos surgió como resultado de la intersección de los diferentes campos de investigación que convergieron en la línea de KDD. Entre los campos que participaron existió una gran influencia de una rama de la Inteligencia Artificial denominada *Machine Learning*. En base a dicha influencia inicial, el proceso se representó mediante el esquema mostrado en la figura 1.1.

1.2.2.2 Enfoque orientado a las Bases de Datos

Debido a que el volumen de información a tratar rápidamente alcanzo unos ordenes de magnitud de Gigabytes, un factor crucial para el éxito del proceso era el tratamiento eficiente de los datos, lo cual requería la intervención de mecanismos de gestión de datos de grandes prestaciones. Esta necesidad hizo que los sistemas gestores de bases de datos tomasen un mayor protagonismo en el proceso, en detrimento del almacenamiento de información en ficheros característico del proceso anterior. Esta nueva influencia hizo que el proceso se adaptase para encuadrar mejor dentro de las necesidades y características de los sistemas gestores de bases de datos que daban soporte al mismo.

Esta nueva tendencia condujo a que en 1997 Mannila [Man97] plantease una nueva división del procesos de KDD en las fases:

❶ **Preproceso**: Esta fase engloba la comprensión del dominio del problema, la selección de los datos de estudio de las fuentes y la preparación de los mismos, considerando como tareas de preparación la eliminación de valores no válidos y la preparación de los datos para la fase posterior. Es importante destacar que la misma preparación de los datos puede utilizarse para varias consultas de *Data Mining* y que en un caso real muchos de las operaciones de preparación y codificación de los datos realizadas en esta fase son utilizadas en multitud de tareas de la fase de *Data Mining*.

❷ ***Data Mining*** : Una vez preparados los datos, esta fase aplica los algoritmos necesarios para extraer información útil de los mismos. Las técnicas utilizadas en esta fase son muy variadas, desde algoritmos de inducción de *Machine Learning* hasta redes neuronales pasando por enfoques estadísticos. En multitud de casos las soluciones necesarias para resolver un problema real requieren la combinación de varias de estas técnicas.

❸ **Postproceso**: Una vez construidos los modelos (reglas, árboles, . . .) esta fase trata la información extraída para presentársela al usuario. En ciertos casos los modelos generados son complejos de entender para el usuario o la información que proporcionan no es válida para su aplicación directa al dominio del problema. En estos casos es necesario un tratamiento posterior de la información extraída por la fase de *Data Mining*. Dentro de esta fase se encuentran técnicas de visualización (2D, 3D, grafos, . . .).

1.2.2.3 Modelo de Proceso CRISP-DM

El consorcio de fabricantes y empresas CRISP-DM, entre las cuales destacan como líderes del consorcio NCR, DaimlerChrysler, ISL y OHRA, han formalizado un modelo de soporte para una metodología de implantación de soluciones de *Data Mining*. El documento del estándar [CCK+00]

describe las fases de construcción de un modelo de *Data Mining* y su presentación en términos comunes para la interlocución con los usuarios de la solución.

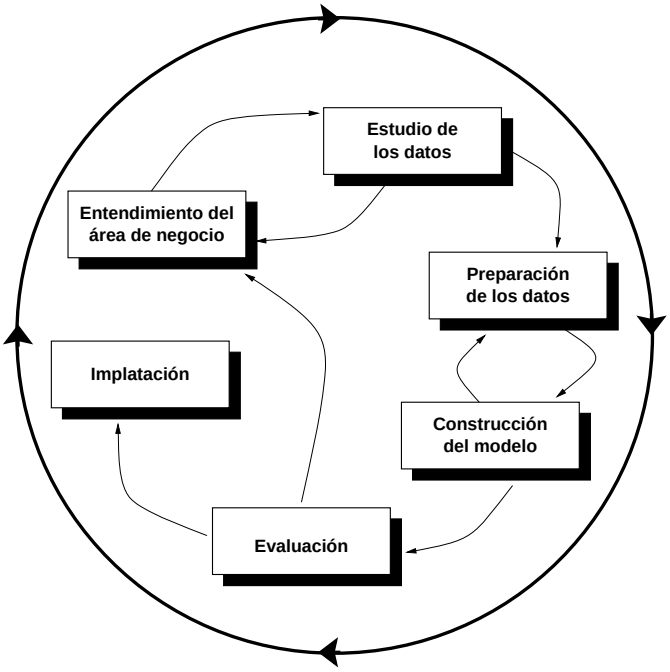


Figura 1.2: Fases de CRISP-DM

Este último paso en la evolución del término KDD o *Data Mining* está directamente enfocado hacia su aplicación dentro del campo empresarial y de negocio de las organizaciones. A diferencia de los dos enfoques anteriores orientados hacia la división de fases de un proceso complejo bajo dos diferentes perspectivas, el modelo propuesto por CRISP-DM describe un ciclo de vida, es decir, las fases de desarrollo de un proyecto de *Data Mining* y la dependencia entre dichas fases.

La relación entre las diferentes fases del modelo de ciclo de vida de un proyecto tal y como está descrito por CRISP-DM se pueden ver en la figura 1.2. Estas fases descritas por CRISP-DM son:

① **Entendimiento del Área de Negocio**: En esta fase de reconocen los objetivos y requerimientos del proyecto. La descripción de estos elementos se realiza desde la perspectiva del negocio y como resultado se define un problema de *Data Mining* y se diseña un plan con los objetivos del proyecto.

② **Estudio de los Datos**: Esta fase comprende la recuperación y selección de los datos de trabajo. El objetivo de la misma es la comprensión y familiarización con los datos. un análisis

adecuado de los datos debe proporcionar pautas de selección de los atributos más relevantes y de cuáles pueden ser los valores derivados de interés.

③ **Preparación de los Datos**: A lo largo de esta fase se construye el conjunto final de datos de trabajo para las fases siguientes. Esta fase está compuesta por una serie de etapas cíclicas que concluyen cuando el conjunto de datos y atributos que los describen es satisfactorio:

- ❒ *Selección de los datos*: Selección de los registros y atributos a usar.
- ❒ *Limpieza de los datos*: Eliminación de valores nulos, erróneos, incompletos y de mala calidad.
- ❒ *Construcción de nuevos datos*: Derivación de nuevos atributos en base a los originales u otros atributos creados, cuya utilización por el proceso de construcción del modelo sea más adecuada.
- ❒ *Integración de los datos*: Combinación de múltiples tablas de datos que completan la información de los datos.
- ❒ *Dar formato a los datos*:Modificación de la representación, escala o cualquier otro aspecto de formato.

④ **Construcción del Modelo**: Se trata de una fase iterativa en la cual se generan diferentes modelos y se refina la solución de forma progresiva. El objetivo de esta fase es encontrar una representación adecuada de la solución al problema de *Data Mining* planteado. Las etapas que componen esta fase son:

- ❒ *Selección de la técnica de modelización*: Elección de la técnica o técnicas de análisis de datos usadas para el problema.
- ❒ *Generación del test del modelo*: Antes de construir el modelo se determina el mecanismo de evaluación de sus resultados.
- ❒ *Construcción del modelo*: Por medio de una herramienta o programa se extraerá el modelo o modelos. Dicha fase requiere el ajuste de los parámetros de comportamiento que cada algoritmo tiene.
- ❒ *Evaluación del modelo*: Evaluación de los resultados obtenidos en base al proceso de verificación establecido y el conocimiento en el campo de negocio.

⑤ **Evaluación**: Esta fase es complementaria a la tarea de evaluación del modelo particular. Aquí se juzga no sólo la calidad del modelo sino toda la estrategia seguida para la construcción del mismos. Las etapas de esta fase son:

- ❒ *Evaluación de resultados*: Se evalúa la utilidad del modelo dentro del problema de negocio que intenta resolver (no sólo su exactitud y generalidad, antes evaluadas).

- *Revisión del proyecto*: Se juzga el proyecto de forma global, estudiando la existencia de otras tareas derivadas o pendientes de interés.
- *Determinar los pasos siguientes*: Como resultado del proceso de revisión se definen los siguientes pasos a realizar, es decir si el proyecto de termina o si es necesaria la revisión de alguna fase.

Implantación: Una vez que se ha encontrado la solución, en esta fase se plantea la aplicación de la misma a proceso de negocio de la organización. Esta fase implica la presentación de los resultados obtenidos de una forma ordenada a los usuarios o clientes. Esta fase se aborda en las siguientes etapas:

- *Creación de un plan de implantación*: Definición de las fases en las cuales la solución puede ser incluida dentro del proceso de negocio.
- *Creación de un plan de supervisión/mantenimiento*: Se plantea la estrategia de revisión de los resultados obtenidos a lo largo del tiempo y su refinamiento progresivo.
- *Creación del informe final*: Se recapitulan los resultados del proyecto y los hitos del desarrollo del mismo.
- *Revisión global del proyecto*: Se evalúan los resultados positivos y negativos de proyecto completo así como futuras líneas de aplicación del mismos.

El consorcio CRISP-DM acoge a un grupo de interes (SIG) que agrupa a usuarios de *Data Mining* encargados de refinar y actualizar el documento estándar así como de debatir diferentes soluciones y propuestas dentro del campo de aplicaciones de *Data Mining*.

1.2.3 Líneas de Investigación

Dentro del área de *Data Mining* o KDD existen numerosas líneas de investigación, aunque se pueden agrupar bajo tres grandes ramas:

- Desarrollo de nuevas aportaciones en algoritmos, basados en nuevos modelos matemáticos de representación de la información (como *rough sets* [Paw82] o *fuzzy sets* [Zad93]) o derivados de otros campos del conocimiento como la estadística, la inteligencia artificial [Qui86, Qui92] o las bases de datos [AIS93b].
- Aplicación de las técnicas de *Data Mining* para resolución de problemas dentro de otros campos, tales como el análisis de datos industriales [PSBK+96, PLF99, PFL00], la biomedicina [Blu82, Szo95, TT96] o el análisis de datos financieros [MR92, Hut93, SS94].

- ❒ Diseño de sistemas capaces de aplicar los algoritmos y técnicas de forma eficiente sobre datos dentro de los dominios de aplicación de dichos campos. Es dentro de esta línea donde se encuadra esta aportación.

La línea de investigación que tradicionalmente ha tenido más peso dentro de este campo es la relacionada con el desarrollo de nuevas técnicas y algoritmos. Hasta no hace mucho tiempo, la aplicación práctica de las técnicas se restringía a experimentos muy puntuales y no había despertado el interés de los profesionales de dichos campos de aplicación. Por otro lado, los primeros trabajos desarrollados se derivaban directamente de las líneas de investigación de *Machine Learning* y consistían en implementaciones de algoritmos concretos que procesaban ficheros de datos con un centenar de registros de media.

En el momento en el cual *Data Mining* empezó a cobrar vigencia la aplicación de sus técnicas a casos reales y el desarrollo de sistemas que integrasen varios algoritmos fue una realidad. Incluso los investigadores centrados en el desarrollo de algoritmos redirigieron parte de sus esfuerzos a otras fases del proceso de KDD que no eran únicamente la extracción de patrones. Nuevos trabajos de investigación en técnicas de preproceso y limpieza de datos [Kri95] o visualización de resultados [GSSW92, EJ95, FKZ97, DKR97] empezaron a aparecer. Todas estas técnicas y algoritmos fueron integradas en sistemas cada vez más complejos de *Data Mining* capaces de ejecutar diversos algoritmos sobre datos recuperados de varias fuentes y cuya utilización fue adoptada por los profesionales de otros campos. Aun tratándose de una línea de trabajo con un cierto bagaje y muy evolucionada, el diseño de sistemas de *Data Mining* aun presenta ciertos problemas que no han sido resueltos de forma satisfactoria hasta el momento y constituyen por lo tanto líneas de investigación abiertas. Es en esta línea donde se centra la principal aportación de este trabajo.

1.3 Sistemas de *Data Mining*

En primer lugar, se presenta la definición del término tal y como va a ser manejado a lo largo de este trabajo.

> **Definición** SISTEMAS DE *Data Mining*: Aplicación integrada diseñada para ejecutar varios algoritmos de *Data Mining* y otras operaciones del proceso de KDD sobre un conjunto de datos bien gestionados por el propio sistema o accesibles de diversas fuentes de datos externas, tales como SGBDs.

Esta definición diferencia el concepto de sistema de *Data Mining* aquí expuesto de meras implementaciones de algoritmos de *Data Mining*. El concepto de integración permite que la ejecución de varias operaciones dentro del proceso de análisis pueda ser realizada dentro del propio

sistema aplicando para ello varios algoritmos y transformaciones. Otra cualidad importante es que bien el propio sistema o bases de datos externas son las encargadas de almacenar los datos, considerándose que en todo momento el volumen de dichos datos es apreciable.

1.3.1 Características de un Sistema de *Data Mining*

Entrando más en detalle se pretende desglosar las características de los sistemas de *Data Mining* actuales. Este análisis de características permitirá evaluar los sistemas actuales, para posteriormente definir cuáles son las líneas de investigación abiertas y en qué dirección han de aportar nuevas características a estos sistemas.

1.3.1.1 Integración de Varias Técnicas

La solución a la mayoría de los procesos de análisis de datos no es por lo general alcanzable por la aplicación de un único algoritmo que genere los resultados esperados. El proceso de análisis es mucho más complejo y, a menudo, requiere la aplicación de varios filtros y transformaciones antes de la aplicación de uno o varios algoritmos.

Es necesario para los usuarios de sistemas de *Data Mining* tener la posibilidad de combinar múltiples algoritmos para obtener los resultados. Cuanto mayor sea el conjunto de algoritmos y más fácilmente se pueda ampliar, el sistema podrá ser aplicable a más entornos.

1.3.1.2 Pre-proceso y Post-proceso

El proceso de KDD se ha visto que comprende más tareas que la mera extracción de patrones. Las funcionalidades para añadir operaciones de preparación, limpieza y transformación de los datos son fundamentales para poder aplicar los algoritmos de *Data Mining* de una forma adecuada. Una vez aplicado un algoritmo de extracción de patrones, su resultado ha de ser tratado también para su presentación al usuario final. Dentro de esta fase se encuentran las operaciones de representación de patrones, visualización, aplicación de patrones, predicción de valores o *scoring*.

1.3.1.3 Acceso a Múltiples Fuentes de Datos

El objeto de las operaciones de un sistema de *Data Mining* son los datos sobre los que se aplicadon los algoritmos. A diferencia de los primeros sistemas, en la actualidad la aplicación de procesos de análisis sobre ficheros planos es raramente utilizada, puesto que los datos a analizar se encuentran por lo general recogidos en bases de datos relacionales o *Data Warehouses*. La capacidad de un sistema para recuperar los datos de dichas bases de datos es fundamental para su uso en entornos reales.

Esta característica apunta a la capacidad para obtener datos de fuentes externas aunque internamente se gestionen de otra forma. La gestión interna de datos puede realizarse bien por el sistema o también por medio de una base de datos que el sistema utilice como soporte para sus operaciones.

1.3.2 Sistemas Actuales de *Data Mining*

El desarrollo de sistemas de *Data Mining* ha evolucionado notablemente en los últimos años, de tal forma que los sistemas disponibles han salido de los circuitos puramente académicos y de investigación para dar lugar a versiones comerciales de dichos sistemas. Esto ha sido posible gracias al interés de profesionales de otros campos en usar estas herramientas para analizar los datos de su campo.

1.3.2.1 DBMiner

La Universidad de Simon Fraser en Canadá ha desarrollado el sistema de *Data Mining DBMiner* basándose en la experiencia obtenida en el desarrollo del prototipo inicial denominada *DBLearn*. Esta es una de las primeras aportaciones que integraban la visión del proceso de *Data Mining* desde la perspectiva tradicional de *Machine Learning* y su dependencia con las tecnologías de bases de datos.

Las principales características de *DBMiner* son:

- Un conjunto de algoritmos bastante amplio y basado en consultas definidas en base a jerarquías conceptuales. Dichas jerarquías describen determinados conceptos a diferentes niveles de generalidad (desde valores muy específicos hasta conceptos generales). Por medio de las jerarquías conceptuales es posible extraer conocimiento a diferentes niveles de granularidad.
- Un mecanismo de interacción con el sistema basado tanto en un interfaz gráfico como en un lenguaje de interrogación propio denominado DMQL (*Data Mining Query Language*) [Mal97].
- Integración de los algoritmos con el SGBD orientado hacia la obtención de una mayor eficiencia.

La arquitectura de *DBMiner* (representada en la figura 1.3) consta de cinco elementos fundamentales. Por un lado el interfaz de usuario (gráfico aunque también admite consultas mediante DMQL). En segundo lugar, los **módulos de descubrimiento** compuestos por las diferentes funcionalidades de análisis del sistema, los algoritmos y las operaciones de procesamiento de datos

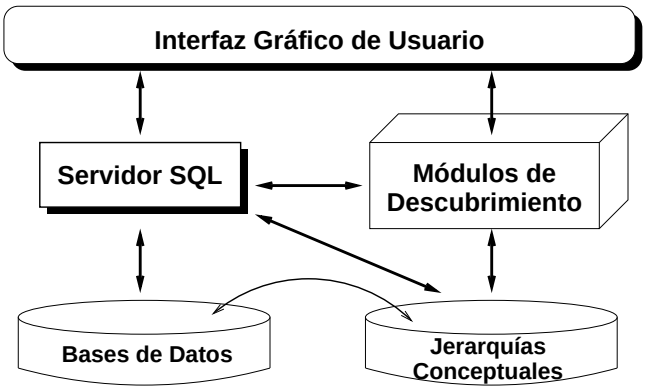


Figura 1.3: Arquitectura de *DBMiner*

(entre ellas el **gestor de jerarquías**). Al mismo nivel que los módulos de descubrimiento se localiza el gestor relacional, el cual es consultado por los algoritmos durante su operación. La integración alcanzada entre las consultas de *Data Mining* y el SGBD es a muy alto nivel y no aprovecha de este último muchas de las funcionalidades útiles para el proceso de *Data Mining*. Por último, los datos de análisis de las jerarquías de conceptos representan las fuentes de información usadas como entrada en los procesos de análisis.

Una explicación mas detalladade *DBMiner* puede encontrarse en [HFW+96b, HFW+96a, The97].

1.3.2.2 Quest

El proyecto *Quest* se realiza en el centro de investigación de IBM en Almaden (EE.UU.). El sistema desarrollado se planteó como una plataforma para el desarrollo de algoritmos y sus ejecución bajo una serie de premisas:

- El objetivo es descubrir patrones en grandes bases de datos y no el verificar si un conjunto de patrones se encuentran presentes.
- Garantizar la propiedad de *completitud* que asegure que todos los patrones de un determinado tipo han sido encontrados.
- Proporcionar un rendimiento *quasi-lineal* cuando se escala a grandes bases de datos reales (múltiples gigabytes).

Las operaciones soportadas por las últimas versiones del sistema incluyen algoritmos de cálculo de asociaciones (Apriori de Agrawal [AIS93b]), patrones secuenciales, segmentación en base a series temporales y clasificación. Adicionalmente el sistema proporciona operaciones de generalización en base a jerarquías de conceptos y varias funcionalidades de preprocesamiento.

Centrando la perspectiva del sistema en sus características de soporte a procesamiento de grandes prestaciones, destacan dos funcionalidades:

- Análisis incremental (o *incremental mining*) [AP95] técnica basada en el particionamiento de los datos en base a un criterio temporal con una granularidad dependiente de la aplicación. Posteriormente, el algoritmo de *Data Mining* se aplica a cada uno de los fragmentos, obteniendo los patrones y almacenándolos en una base de datos de históricos. Los patrones son posteriormente usados para definir activadores (o *triggers*) dentro de la base de datos que contiene la información, de forma que cuando la modificación de los datos afecta a alguno de los patrones previamente calculados estos son recalculados de forma automática.
- Paralelismo en la ejecución de algoritmos. Esta funcionalidad está resuelta a nivel de algoritmos, es decir que ciertos algoritmos implementados en el sistema han sido desarrollados para poder ejecutarse de forma paralela sobre un sistema basado en una arquitectura SP2 de cluster de multiprocesadores. Los algoritmos desarrollados de esta forma son una versión paralela de Apriori [ZOPL96] y el clasificador paralelo SPRINT [SAM96b].

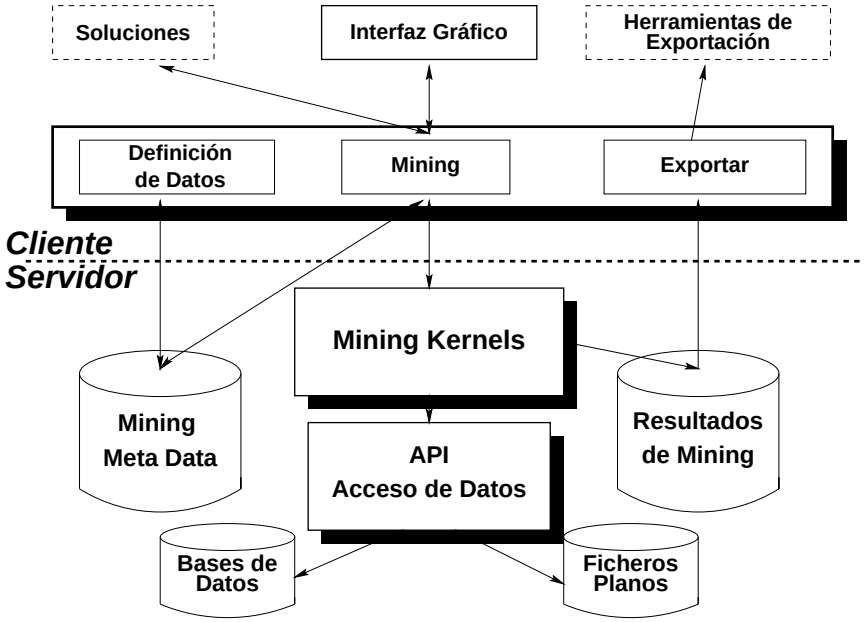


Figura 1.4: Arquitectura de *Quest*

La arquitectura de *Quest* (figura 1.4) se basa en dos partes: una serie de módulos cliente que ejecutan en un terminal de control y un servidor que se ejecuta próximo a las fuentes de datos, que son indistintamente gestores relacionales (DB2 de IBM preferentemente) o ficheros planos. La aplicación cliente proporciona funcionalidades de interacción con el usuario vía Interfaz Gráfico

de Usuario (GUI) y de exportación de resultados a otras herramientas. La parte servidor se basa en una serie de motores de algoritmos que interactúan en base a una API de acceso con las fuentes de datos. Uno de los mayores logros de *Quest* es la arquitectura de entrada/salida de los algoritmos que permite que éstos recojan del repositorio de datos toda la información de ejecución.

Quest es uno de los sistemas experimentales más avanzados. La implementación de ciertos algoritmos de forma paralela le proporciona una escalabilidad y un rendimiento muy importante. La pega es que dichas prestaciones se consiguen por medio de implementaciones de dichos algoritmos específicamente optimizados para la arquitectura hardware de soporte del sistema. *Quest* carece de mecanismos de paralelismo o distribución de carga fuera de los expresamente aplicados en el diseño de los pocos algoritmos paralelos que proporciona, desaprovechando el potencial de la arquitectura hardware que lo soporta.

Más información sobre el proyecto *Quest* y sus resultados se puede consultar en [ACF+94a, ACF+94b, AMS+96].

1.3.2.3 Intelligent Miner

Intelligent Miner o *IMiner* es la versión comercial desarrollada por IBM a partir del proyecto experimental *Quest*. Básicamente presenta una arquitectura y funcionalidades similares, pero un mayor número de algoritmos y técnicas disponibles. El planteamiento de diseño es igual al abordado durante el desarrollo de *Quest* y se diferencia únicamente en la inclusión de:

- Un módulo de gestión de resultados. Descrito por medio de una API de salvaguarda de resultados de las operaciones.
- Un módulo de exportación de resultados. Un módulo de comunicación con herramientas externas por medio de otra API.
- Unas herramientas de visualización de resultados.
- Una librería de procesamiento que facilitan la carga de datos de diferentes bases de datos.

La arquitectura de *IMiner*, mostrada en la figura 1.5, como se puede observar, guarda una gran similitud con la de *Quest* antes presentada.

A nivel del análisis de características que se quiere realizar a lo largo de este capítulo esta arquitectura no presenta desde el punto de vista de diseño ninguna innovación relevante en relación a la anterior. Las mejoras que diferencian a este sistema de *Quest* van orientadas más en la dirección de una mayor robustez, flexibilidad de instalación y mayor cantidad de algoritmos de análisis, más

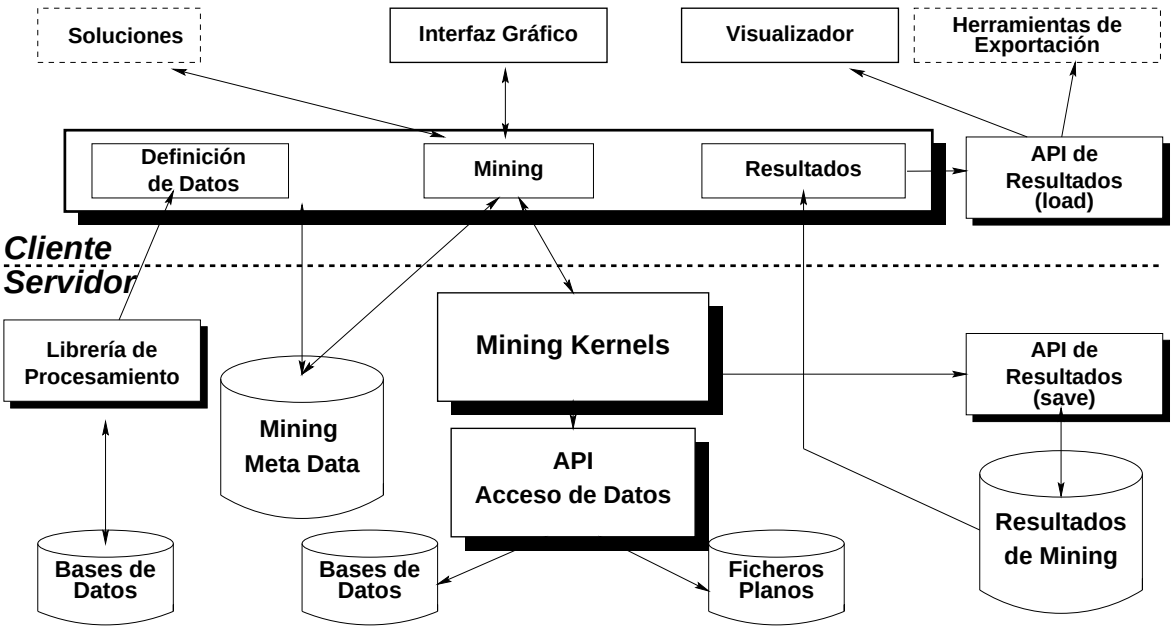


Figura 1.5: Arquitectura de *Intelligent Miner*

que en la dirección de una revolución en el diseño del producto. Para más información, se refiere al lector a [Tka98, IBM99].

1.3.2.4 MLC++

$\mathcal{MLC}^{++}$ (*Machine Learning library in C++*) son un conjunto de librerías inicialmente diseñadas por la Universidad de Stanford, aunque su desarrollo final ha sido realizado por *Sillicon Graphics* (SGI). El objetivo de los desarrolladores de estas librerías está orientado en dos direcciones:

- En primer lugar, hacia los usuarios finales que desean realizar procesos de análisis de datos. Para ellos, las $\mathcal{MLC}^{++}$ proporcionan un gran número de algoritmos y técnicas de *Data Mining* para dicha tarea.
- Por otro lado, los investigadores que desarrollan nuevos algoritmos de *Data Mining* pueden usar las $\mathcal{MLC}^{++}$ como base para dicho desarrollo. Para estos usuarios, las librerías proporcionan funcionalidades comunes para múltiples algoritmos que pueden ser reutilizadas en diferentes diseños.

Estos dos objetivos trazados en el diseño de las librerías se basan en la idea postulada por los autores (Ron Kohavi et al.) de que: "*no existe un mejor algoritmo para todos los dominios y problemas*". Considerando como criterio de comparación no sólo la precisión de la predicción realizada, sino también factores como la comprensibilidad y simplicidad de los resultados y el tiempo de ejecución de los mismos. Este planteamiento induce a la construcción de conjuntos de algoritmos

suficientemente extensos como para permitir que para un problema dado, los investigadores puedan usar múltiples aproximaciones ya desarrolladas o diseñar nuevos algoritmos *ad hoc* para el problema. Esta idea se traduce en un conjunto muy amplio de algoritmos (especialmente centrados en problemas de clasificación) y en un diseño modular de los mismos que permite ampliarlos y reutilizar código en el desarrollo de nuevos algoritmos.

El diseño de las librerías $\mathcal{MLC}^{++}$ se encuentra dividido en una serie de funcionalidades principales:

- **Clases de Soporte Generales**: Que proporcionan soporte a una serie de operaciones generales, no necesariamente relacionadas con *Machine Learning* o *Data Mining*. Estas librerías proporcionan estructuras de datos y funcionalidades de representación manejo de dichas estructuras. A este nivel se han de desatacar las librerías LEDA de Stefan Näher [MN95, MNSU96, MN99a].

- **Clases del Núcleo**: Que proporcionan funcionalidades compartidas por varios algoritmos, tales como entrada/salida, conversión de formatos, generación de resultados y recubrimiento o *wrappers* de algoritmos (usado para la evaluación de la precisión, métodos de validación y curvas de aprendizaje).

- **Categorizadores y *Mappers***: Encargados en asignar a los datos de entrada a diferentes categorías o clases en el caso de los primeros o asignarles un valor real de pertenencia a un determinado conjunto. Dentro de este grupo de elementos se encuentran diferentes implementaciones de distintas técnicas. Estos componentes representan los modelos de representación manejados por las librerías.

- **Algoritmos de Inducción**: Encargados de definir los elementos anteriores, se corresponden con algoritmos de diferentes técnicas y representan el núcleo de capacidades de la librería. La variedad de algoritmos presentados por las últimas versiones de la librería es especialmente relevante.

La flexibilidad a la hora de incorporar nuevos algoritmos radica en la existencia de un proceso de ejecución muy bien definido en el cual cada elemento es modularmente intercambiable por otro con la misma funcionalidad. Este proceso (ver figura 1.6) está compuesto por el recubrimiento o *wrapper* (encargado del proceso de generación de conjuntos de entrenamiento/prueba y la combinación de los resultados), un algoritmo de inducción y un validador de resultados.

La principal aportación de las librerías $\mathcal{MLC}^{++}$ a los sistemas de *Data Mining* es la modularidad de su modelo de proceso, definido en base a una serie de etapas claramente especificadas que permiten una gran flexibilidad en el desarrollo de nuevos algoritmos en base al mismo proceso. Un

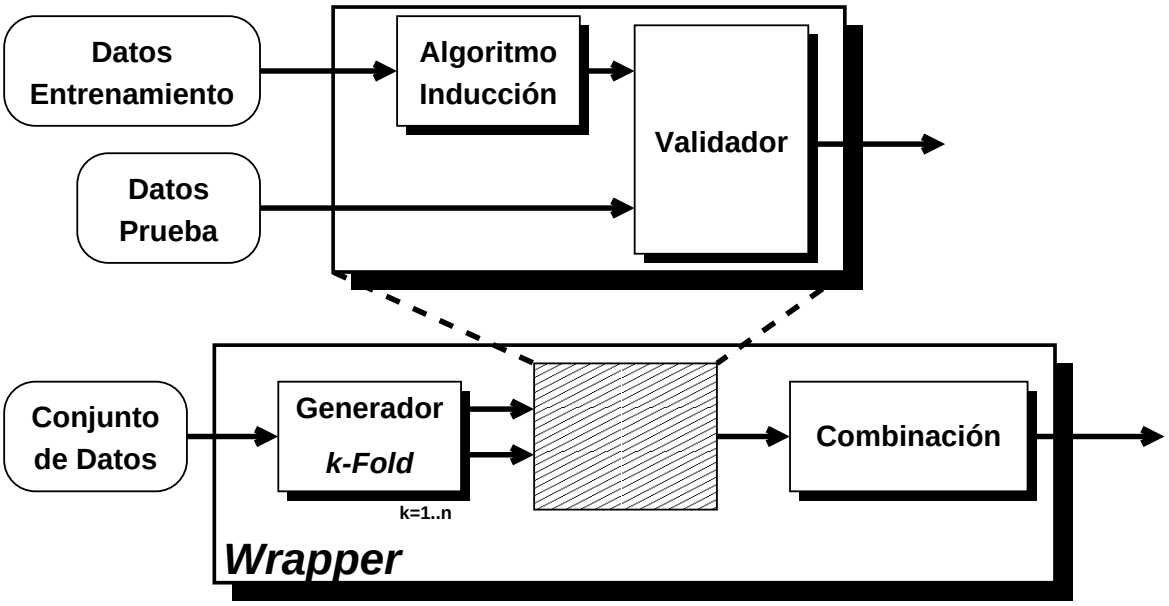


Figura 1.6: Modelo de proceso de las librerías MLC++

nuevo algoritmo puede ser definido modificando el proceso de generación de conjuntos de entrenamiento/prueba o por medio de un nuevo algoritmo de inducción. Por desgracia, la flexibilidad se restringe a un modelo de proceso muy concreto, requiriendo que la definición de un proceso más complejo o con diferentes tipos de etapas tenga que ser definido fuera del entorno definido por las librerías, usando estas únicamente como un elemento de un proceso más amplio.

Para un estudio más en profundidad de las funcionalidades de estas librerias, el lector puede enontrar información en [KJL+94, Lon94, Koh95, KSD96, KS96].

1.3.2.5 MineSet

El producto comercial de la empresa *Sillicon Graphics* (SGI) dentro del campo de *Data Mining* se denomina *MineSet*. Originalmente concebido como una herramienta de visualización, posteriormente se han ido añadiendo al diseño funcionalidades de análisis muy significativas. El procesamiento de consultas se realiza en base al conjunto de librerías $\mathcal{MLC}^{++}$ vistas anteriormente.

La arquitectura de *MineSet* (figura 1.7) está dividida en dos niveles, por un lado, una herramienta cliente compuesta por una serie de módulos de visualización y un interfaz de control de las funcionalidades del sistema. Por el otro, el núcleo del sistema compuesto por un módulo denominado *Data Mover* encargado de la recuperación de los datos y de la coordinación de una serie de elementos, compuestos por los algoritmos de *Data Mining* de las librerías de análisis. El módulo

Data Mover se conecta con las fuentes de datos (ficheros, bases de datos o *Data Warehouses*) que pueden localizarse dentro del mismo computador o en un servidor diferente, definiendo un tercer nivel en la arquitectura del sistema.

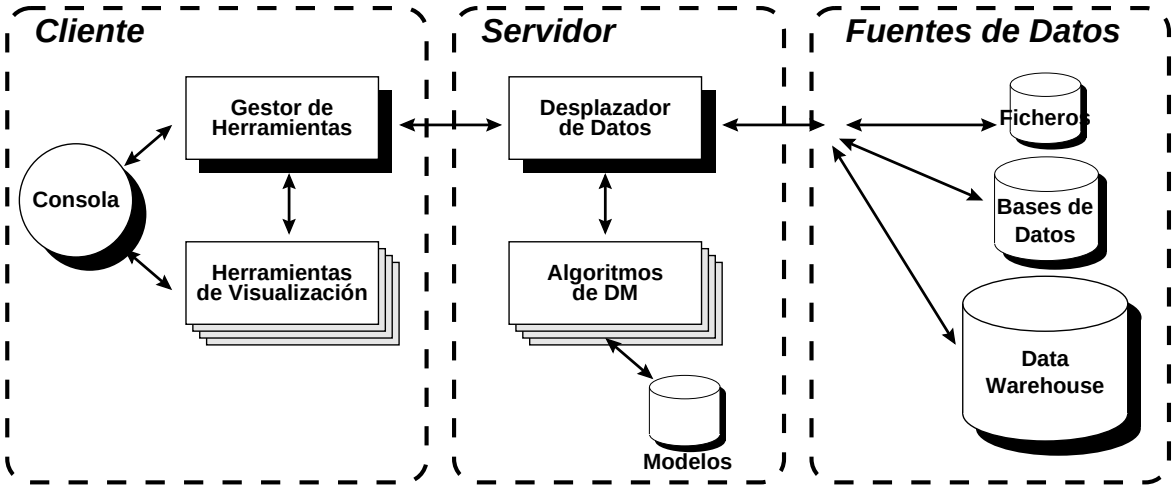


Figura 1.7: Arquitectura de *MineSet*

Las características más reseñables del sistema son:

- Importantes funcionalidades de visualización. La idea original del sistema y la impronta característica de la empresa han influido notablemente en las prestaciones de esta herramienta a ese nivel. Sin duda las capacidades de representación visual de resultados son las más avanzadas del mercado.

- Soporte de desarrollo. La implementación del sistema ha sido soportado por dos conjuntos de librerías. Por un lado, las funcionalidades de representación visual se han extraído de los paquetes Open Inventor y OpenGL. Y por otro lado, las capacidades de análisis se sustentan en las librerías $\mathcal{MLC}^{++}$. Ambos conjuntos de librerías son punteras en sus campos de aplicación.

- Existen dos versiones de la herramienta una orientada a estaciones de trabajo, para la cual cliente, y servidor se ejecutan en una misma computadora. La otra versión esta orientada a instalaciones de grandes prestaciones en las que el cliente se ejecuta en una estación de trabajo de control y el procesamiento de datos se realiza en un servidor de mayor potencia de cálculo.

Desde el punto de vista de prestaciones, *MineSet* no está tan evolucionado como *Quest* o *IMiner* a la hora de soportar *Data Mining* distribuido o algoritmos paralelos. Además, los algoritmos

de *MineSet* asumen que todo el conjunto de datos puede ser almacenado en memoria y salvo la implementación de SPRINT [SAM96b] el resto de algoritmos realizan numerosas pasadas sobre el conjunto de datos. Esta filosofía esta apoyada en la opinión de que, de forma generalizada, el espacio de memoria usado por los servidores en poco tiempo alcanzará los órdenes de magnitud de los datos actualmente analizados. Por supuesto, esta teoría considera que dicho volumen de información no se escalará de la misma forma. Estas dos características limitan la aplicación de esta herramienta a cierto tipo de problemas sin la inversión en hardware adecuada.

En [BKK97, Koh97, RV98] se encuentra la información detallada de *MineSet*.

1.3.2.6 Clementine

La empresa *Integrated Solutions Limited* (ISL) comercializa el sistema *Clementine* diseñado como fruto de la colaboración con varios investigadores de diferentes universidades europeas. Este sistema proporciona unas excelentes capacidades para la definición de procesos de *Data Mining* complejos en base a un *toolbox* de múltiples elementos, tales como algoritmos de diferentes técnicas (para clasificación, cálculo de asociaciones y segmentación de instancias, principalmente), procesos de transformación de datos (estadísticos y relacionales) y multitud de componentes de importación y exportación de datos y resultados. En conjunto el paquete de elementos disponibles en *Clementine* cubren gran parte de las tareas del proceso completo de KDD, desde ciertas funcionalidades de limpieza de datos (no todas las posibles) hasta la elaboración de informes para usuarios finales.

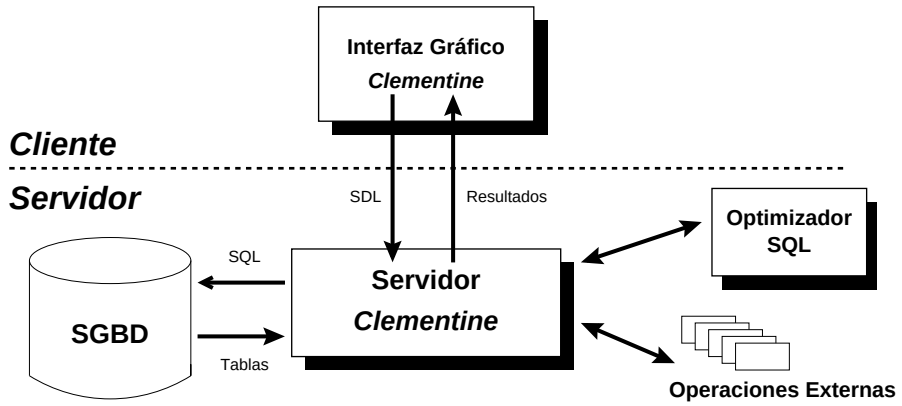


Figura 1.8: Arquitectura de *Clementine* para el servidor distribuido

Una de las principales deficiencias de este sistema es la limitación del volumen de datos que es capaz de procesar en un tiempo razonable. Para solventar esta desventaja ISL ha desarrollado

la arquitectura distribuida de servidor (*Clementine Server Distributed Architecture*) [ISL99]. Esta arquitectura (ver figura 1.8) presenta un modelo de cooperación cliente/servidor de tres niveles (a diferencia del modelo original *standalone*). En la arquitectura distribuida, el cliente proporciona al usuario la capacidad de definir mediante un interfaz gráfico de usuario el proceso a realizar, transmitiéndolo en formato SDL (*Stream Definition Language*) al servidor. Este servidor analiza la cadena de operaciones y diferencia entre:

- ❒ operaciones que se pueden solicitar a la base de datos vía SQL y
- ❒ operaciones externas a la base de datos

De esta forma el servidor de la arquitectura actúa como un distribuidor de tareas entre los dos tipos de operaciones a realizar. Las operaciones realizadas dentro de la base de datos son operaciones puramente relacionales (*joins*, ordenaciones, proyecciones o selecciones). Dichas operaciones pueden ser realizadas con un alto rendimiento dentro de los SGBD. Las operaciones externas, por su parte, son operaciones de análisis y directamente relacionadas con algoritmos de *Data Mining* u operaciones de preprocesamiento más sofisticadas.

Otra de las diferencias de la arquitectura distribuida de *Clementine* en relación a la versión anterior y a otros muchos sistemas es que no se asume que las operaciones externas son realizables en memoria y por lo tanto es el propio sistema el que gestiona el almacenamiento de resultados finales e intermedios entre memoria y disco.

Esta versión de alta prestaciones de *Clementine* avanza pues un paso hacia el proceso de integración entre las herramientas de *Data Mining* y los SGBD donde los datos se almacenan. Sin embargo, esta interacción es únicamente parcial, pues internamente los algoritmos no pueden hacer uso de funcionalidades relacionales del gestor de bases de datos y estas se restringen únicamente a las primeras etapas del proceso de KDD. De todas formas esta mejora consigue disminuir uno de los principales problemas de las arquitecturas de tres niveles de *Data Mining* que es el consumo de recursos de comunicación (ancho de banda en la red) entre el SGBD y la herramienta de análisis. Esta reducción se basa en que como resultado de las operaciones realizadas en el SGBD el conjunto de datos transmitido es menor.

Más información en relación a los productos de *Data Mining* de ISL y al diseño de *Clementine* se pueden encontrar en [ISL95, KS95, She96, Gri96]. Una referencia muy interesante en relación al sistema *Clementine* es el proyecto CITRUS [WSG+97]. Mediante este proyecto se pretende dotar al sistema de una herramienta que asista al usuario en la definición de la cadena de operaciones que definen el proceso de análisis sobre los datos.

1.3.2.7 DAMISYS

DAMISYS es un sistema no comercial desarrollado en la Facultad de Informática de la Universidad Politécnica de Madrid [FPM+99, FMP+00]. El sistema tiene como criterios de diseño, por un lado la flexibilidad en la definición de nuevos mecanismos de análisis y por otro la integración del proceso de *Data Mining* con la tecnología de bases de datos.

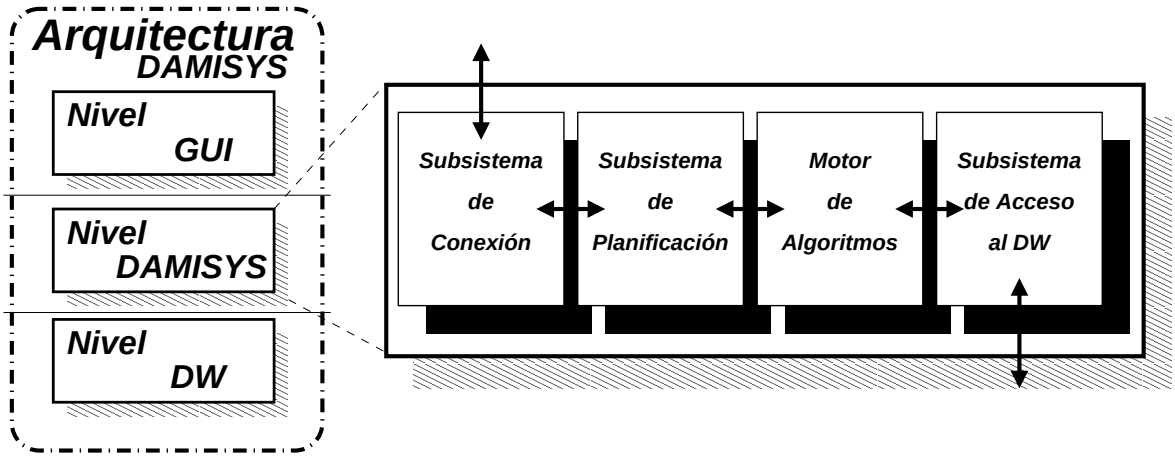


Figura 1.9: Arquitectura de *DAMISYS*

El diseño general del sistema contempla tres niveles dentro de la arquitectura. Un primer nivel con funcionalidades de acceso al sistema, en base a herramientas gráficas de usuario y administrador así como de un lenguaje de interrogación denominado RSDM-QL. Un nivel intermedio donde se definen las funcionalidades de análisis del sistema. Por último un nivel de almacenamiento de datos de soporte al sistema o *Data Warehousing*.

El nivel fundamental de la arquitectura es el intermedio (figura 1.9) que esta compuesto por una serie de subsistemas:

- **Subsistema de Conexión**: Encargado de controlar las sesiones en el sistema, controlando el acceso a los usuarios autorizados y una vez que estos se han conectado interpretando las operaciones transmitidas en RSDM-QL (tanto directamente como desde los interfaces gráficos de usuario, que también generan peticiones en base a esa gramática).

- **Subsistema de Planificación**: Encargado de analizar las peticiones y definir planes de ejecución para las mismas. Los planes son tanto de sentencias de administración como consultas propiamente dichas. En este último caso éstas se encuentran definidas en base a algoritmos descritos en RSDM/Alg, lenguaje procedimental en el cual se expresan la secuencia de

operaciones para resolver la consulta. Las operaciones elementales de los algoritmos se denominan operadores y un plan de ejecución de una consulta esta formado por cadenas de dichos operadores interconectados entre si.

- **Motor de Algoritmos** [Mar99]: Responsable de la ejecución de las cadenas de operadores transmitidas por el subsistema anterior. Los operadores que componen las cadenas son cargados dinámicamente en dicho momento y ejecutados de forma paralela sincronizada por el propio motor.

- **Subsistema de Acceso al *Data Warehouse*** [Lóp99]: Recoge todas las peticiones realizadas por el resto de subsistemas, especialmente el motor y las transmite al nivel de *Data Warehouse* donde se gestionan y almacenan los datos.

Las funcionalidades más relevantes del sistema *DAMISYS* son:

- La gran flexibilidad alcanzada al no disponer del código de las operaciones de *Data Mining* ligado al propio sistema sino como elementos externos que el sistema carga de forma dinámica. Esto permite que tanto algoritmos (en RSDM/Alg) como operadores (en código nativo) sean añadidos dinámicamente al sistema incluso durante la ejecución del mismos. La modularidad de los algoritmos y la división de los mismos en operaciones elementales, asimismo facilita la reutilización de código y la optimización de las ejecuciones.

- El modelo de ejecución de los operadores dentro del sistema es altamente paralelo. El motor de algoritmos se encarga de coordinar múltiples hilos de ejecución en los cuales cada tarea es un operador, consiguiendo un grano muy fino de paralelismo muy adecuado a arquitecturas SMP. Adicionalmente, el motor de algoritmos es capaz de gestionar la memoria otorgada al sistema mediante un mecanismo de paginación interno que optimiza el uso de memoria para el tratamiento de grandes volúmenes de datos.

- Todos los datos gestionados por el sistema, así como el catálogo de entidades del propio sistema, tales como usuarios, algoritmos, operadores, etc. es gestionado por un *Data Warehouse* [Par98]. Este nivel es capaz de almacenar resultados finales e intermedios con el fin de optimizar la ejecución de futuras consultas.

Las principales desventajas de *DAMISYS* se deben a la débil integración del sistema con el SGBD que sustenta el *Data Warehouse*. Esto hace que el proceso de recuperación de datos para su análisis implique el trasiego de datos entre tres componentes, el SGBD, el nivel de *Data Warehousing* y el nivel *DAMISYS*. Este proceso retarda notablemente la ejecución de ciertas consultas. Por otro lado, el conjunto de algoritmos actualmente disponible es muy reducido, aunque su fácil ampliación hace presumir que aumente rápidamente.

1.3.3 Taxonomía de los Sistemas de *Data Mining*

Si se analizan los sistemas expuestos a lo largo de esta sección se pueden representar por medio de una tabla (Tabla 1.1) las características que cada uno de ellos aporta en diferentes categorías:

- **API**: Dispone o no de un API o interfaz de programación para añadir nuevos algoritmos al sistema. Determina la extensibilidad del sistema.

- **Cliente/Servidor**: Tiene una arquitectura cliente/servidor. Diferencia entre funcionalidades de presentación de resultados y de cálculo.

- **Integración**: Las operaciones realizadas por el sistema están integradas con las funcionalidades del SGBD o únicamente usa éste como soporte de datos.

- **Visualización**: Capacidades visuales de representar los resultados al usuario.

	API	**Cli/Srv** (Niveles)	**Integración**	**Visualización**
DBMiner	NO	SI (3)	MEDIA	MEDIA
Quest	SI	SI (3)	ALTA	MEDIA
IMiner	SI	SI (2/3)	ALTA	MEDIA
$\mathcal{MLC}^{++}$	SI	NO	NO	NO
MineSet	SI	SI (3)	MEDIA	ALTA
Clementine	SI	NO	MEDIA	MEDIA
DAMISYS	SI	SI (3)	MEDIA	BAJA

Tabla 1.1: Características de los sistemas de *Data Mining* estudiados.

Si se centra el análisis en el factor de extensibilidad del sistema, la existencia de un API de desarrollo para nuevos algoritmos posibilita la ampliación de las funcionalidades del sistema por medio de nuevos algoritmos. Estos interfaces de programación, de todas formas tienen sus limitaciones. Por ejemplo, en el caso del sistema *DAMISYS* se han desarrollado tres diferentes versiones del interfaz de desarrollo de operadores hasta alcanzar la versión actual. Estos cambios han servido para refinar el API, debido a que ciertos algoritmos no era posible implementarlos con las versiones anteriores.

Si se analiza la capacidad y eficiencia en el procesamiento de consultas, además de *benchmarks* o conjuntos de pruebas (KDDCup'99 y otros) se pueden estudiar dos factores relacionados con ella:

- El grado de integración que mide el uso de las funcionalidades del gestor relacional sobre el que se apoya. De esta manera, una mayor integración permite un mayor aprovechamiento de las capacidades internas del gestor.

- El tamaño máximo de datos a procesar. Esta es una medida relativa que depende del ordenador que ejecute el sistema, la carga del mismo y parámetros propios de los datos (número y tipo de los atributos).

La gráfica 1.10 muestra el prosicionamiento de cada uno de los sistemas estudiados en base a estas dos características. Este análisis se ha realizado sin atender a resultados concretos que dependan de implementaciones más o menos eficientes de algoritmos concretos. Se han tomado únicamente las consideraciones arquitectónicas relevantes a este efecto.

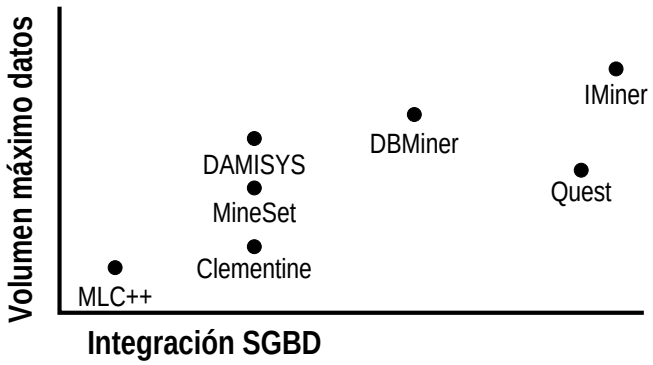


Figura 1.10: Evaluación de la eficiencia de los sistemas de *Data Mining* estudiados

1.4 *Data Mining* Distribuido

Los sistemas actuales de *Data Mining* aun siendo herramientas ampliamente utilizadas en diversos campos tienen una serie de limitaciones debidas a dos factores principalmente:

- La velocidad a la que las organizaciones y centros de investigación almacenan información crece de una forma vertiginosa y a un ritmo muy superior al que los sistemas de *Data Mining* y las arquitecturas sobre los que se sustentan son capaces de procesar. La concepción de conjuntos de datos que pueden ser almacenados en memoria de una estación de trabajo para su análisis dista mucho de los órdenes de magnitud manejados en ciertos campos. La inversión económica asociada a una computadora capaz de procesar dichos volúmenes de información en los casos, en los que es tecnológicamente posible, tiene un coste exorbitante.

- Por otro lado, la naturaleza de muchas fuentes de datos es intrínsecamente distribuida y el planteamiento de su centralización en un único almacén de información plantea problemas almacenamiento, ancho de banda para la transmisión o privacidad entre otros. El caso más

paradigmático de esta información distribuida es la propia red *Internet* la cual soporta una gran base de datos distribuida de páginas *Web* con multitud de información potencialmente útil y de momento oculta entre la inmensidad de datos que la conforman. También se da esta circunstancia en grandes organizaciones o en sistemas de monitorización basados en múltiples sensores.

1.4.1 Limitaciones de los Sistemas de *Data Mining* Actuales

Varios grupos de investigadores en el tema [Ree95], como Robert Grossman y otros, que colaboran en el marco del proyecto M3D2 (*Management and Mining Massive Distributed Data*) [GKM+99], han estudiado las limitaciones actuales de los sistemas de *Data Mining*. El trabajo realizado en este proyecto ha identificado varias áreas de investigación en el campo de *Data Mining* dentro de las cuales se encuadran: (a) la escalabilidad de los sistemas de *Data Mining* y (b) la capacidad para gestionar conjuntos de datos topológicamente distribuidos.

1.4.1.1 Escalabilidad de los Algoritmos

Las estrategias definidas para solucionar el problema de la escalabilidad de los algoritmos frente a grandes volúmenes de datos se han denominado técnicas de *Data Mining* paralelo, *Data Mining* distribuido o incluso *Data Mining* masivo. Todas ellas agrupan a los trabajos realizados en el desarrollo de nuevos algoritmos de *Data Mining* para estos problemas. El punto de partida de estos escenarios es un conjunto de información centralizado o distribuido a lo largo de una red de comunicaciones de altas prestaciones y varios nodos de computación con acceso a todos o a parte de los datos. Las soluciones alcanzadas son algoritmos que aprovechan de forma eficiente los nodos de computación, se coordinan por medio de mensajes que consumen un ancho de banda razonable y en conjunto obtienen unos resultados comparables con los de un algoritmo tradicional.

1.4.1.2 Información Distribuida

El segundo de los problemas del entorno, la información intrínsecamente distribuida es abordado por dos enfoques complementarios. Por un lado se encuadran las técnicas de recopilación de información (o *information gathering*) y por otro la teoría de *Data Mining* colectivo.

La recopilación de información se realiza por medio de una serie de elementos que se desplazan o se encuentran localizados en las fuentes de datos o transmiten a un nodo determinado toda o parte de la información realizando tareas de reconocimiento de datos útiles y algunas operaciones de limpieza de datos.

El enfoque de *Data Mining* colectivo, por su parte plantea la ejecución de gran parte del proceso de análisis sobre los datos de forma local con unas restricciones de comunicación mínima. Una vez completado el proceso de análisis local los resultados, los patrones identificados son combinados con los de diferentes fuentes por un elemento centralizado.

Ambas técnicas difieren de las de *Data Mining* paralelo o distribuido en la consideración de escenarios topológicamente diferentes con multitud de fuentes de datos interconectadas por líneas de media o baja velocidad (como *Internet*) en los cuales una comunicación masiva entre los componentes no es viable.

1.4.2 Análisis de las Necesidades

El diseño de sistemas de *Data Mining* distribuido en cualquiera de los dos enfoques vistos anteriormente, así como en enfoques mixtos se encuentra en una fase de desarrollo inicial. Existen numerosas tentativas que están comenzando a surgir.

Maniatty y Zaki [ZH00a] proponen un análisis de las necesidades a las que un sistema de *Data Mining* distribuido ha de responder, agrupadas en los siguientes términos:

- **Algoritmos Paralelos**
- **Soporte del Proceso**
- **Transparencia en el Acceso**
- **Disponibilidad y Movilidad**

1.4.2.1 Algoritmos Paralelos

El desarrollo de nuevos algoritmos o la adaptación de los ya existentes para su ejecución en paralelo tanto fuerte como débilmente acoplados (alto o bajo grado de interacción). Este desarrollo no sólo concierne a la división de un algoritmo en etapas ejecutables de forma concurrente sino que debe de considerar aspectos tales como la sincronización, la minimización de la comunicación, el equilibrio de carga y la distribución adecuada de los datos entre otros.

Actualmente existen diversos algoritmos paralelos para los distintos tipos de consultas de *Data Mining*. Por ejemplo, en [AS96a, CNFF96, HKK97, Zea97] se presentan aportaciones en algoritmos paralelos para la extracción de reglas de asociación, en [SK98, Zak00] se proponen algoritmos de patrones secuenciales y en [SAM96a, JKK98, ZHA99] se presentan enfoques paralelos para algoritmos de clasificación.

En paralelo al desarrollo de nuevos algoritmos, es necesario definir mecanismos para la comparación de los algoritmos, no solo en base a los parámetros tradicionales de los algoritmos secuenciales sino en los factores distribuidos de los mismos. En esta línea el estudio teórico de Skillicorn [Ski98, Ski99] proporciona una serie de medidas en relación a los algoritmos de *Data Mining* distribuidos.

1.4.2.2 Soporte del Proceso

No sólo los algoritmos de análisis (los correspondientes a la fase de *Data Mining*) deben de ser los únicos paralelizados. Todo el proceso de KDD debe de ser soportado por el sistema distribuido. Las tareas de pre-procesamiento representan como promedio para el caso de análisis de datos reales un 60% del tiempo invertido en el proceso.

Dentro de las operaciones adicionales que es necesario proporcionar, se destacan las tareas de pre-procesamiento de los datos, limpieza de datos, discretización y otras transformaciones, así como las funciones de post-procesamiento, combinación y aplicación de modelos, *scoring*, visualización, etc.

1.4.2.3 Transparencia en el Acceso

En el caso de sistemas orientados hacia *information gathering* la trasparencia en el acceso es fundamental. La información de naturaleza distribuida es también por norma heterogénea. Manejar dicha heterogeneidad en relación a formatos y tecnologías de almacenamiento de forma transparente no sólo al usuario del sistema sino a los receptores directos de dichos datos que son los algoritmos es una característica fundamental.

Otra ventaja asociada al acceso a los datos se obtiene de la explotación de las funcionalidades de los gestores de almacenamiento, por lo general SGBD, que resultan apropiadas para mejorar el rendimiento de los algoritmos. Muchas de las operaciones previas a la ejecución de los algoritmos e incluso el algoritmo entero puede ser realizado internamente dentro de los propios SGBD por medio de las capacidades de programación que proporcionan. Esta aproximación permite reducir los cambios de contexto y explotar las estructuras de gestión internas de las bases de datos. Un ejemplo de esta aproximación se encuentra en la versión fuertemente acoplada del algoritmo Apriori de Agrawal [AS96b].

1.4.2.4 Disponibilidad y Movilidad

Otros factores relevantes de un sistema de *Data Mining* distribuido es la disponibilidad y la tolerancia a fallos. Un sistema complejo compuesto por varios componentes distribuidos tiene un mayor riesgo de caída ante fallos puntuales de ciertos elementos. El propio mecanismo de distribución permite solventar este problema por medio de la replicación de funcionalidades y el enlace dinámico de los componentes que permite definir rutas alternativas a la resolución de operaciones.

La movilidad es otra característica fundamental, especialmente en las herramientas de *Data Mining* sobre grandes redes (e.g. *Internet*), tanto para recopilación de información como para *Data Mining* colaborativo. Asimismo, para *Data Mining* paralelo, la movilidad es una herramienta muy útil para gestionar el equilibrio de carga en un *cluster* de computadoras interconectadas.

1.4.3 Sistemas de *Data Mining* Distribuido

Los sistemas vistos anteriormente (sección 1.3), en la gran mayoría de casos presentaban arquitecturas de tipo cliente/servidor en dos o tres niveles. Los mas avanzados permiten que las tareas más pesadas se realicen aprovechando arquitecturas de multiprocesadores simétricos SMP con memoria compartida. Dichos entornos no son verdaderos entornos distribuidos, los sistemas presentados a continuación están diseñados para operar sobre varios (desde 2 o 3 hasta cientos) nodos de computación interconectados con memorias y dispositivos de almacenamiento independientes.

1.4.3.1 JAM

JAM son las siglas de *Java Agents for Meta-Learning* una arquitectura desarrollada por la Universidad de Columbia [SPT+97]. Los diseñadores de la arquitectura *JAM* describen dos factores como principales motivaciones del proyecto. Por una lado, la naturaleza de datos físicamente distribuidos y sujetos a restricciones para su centralización (privacidad o capacidad de almacenamiento). Por otro lado, el tratamiento de subconjuntos de datos que combinados no es posible almacenar en memoria principal.

La arquitectura *JAM* proporciona un mecanismo para extraer reglas de clasificación de un conjunto de datos explotando la técnica de *Meta-Learning* [CS93a, CS96]. Por medio de esta técnica es posible que los modelos extraídos localmente sean combinados para la obtención de modelos globales válidos. En base a esta fundamentación teórica el sistema calcula clasificaciones en cada una de las localizaciones de datos (denominadas *Datasites*) que son intercambiadas entre si para combinar los resultados.

'ARQUITECTURA DISTRIBUIDA DE CONTROL PARA SISTEMAS CON CAPACIDADES DE DATA MINING'

El sistema *JAM* está compuesto por dos tipos de elementos, por un lado los componentes asociados a cada *Datasite* y por otro los componentes globales del sistema. Los elementos desplegados en cada *Datasite* son:

- ❒ Una base de datos local.
- ❒ Uno o más agentes de aprendizaje (*learning agents*) representados por implementaciones en Java de algoritmos de clasificación. Dichos agentes tienen la capacidad de ser migrados de un nodo a otro independientemente de sistemas operativos y arquitecturas.
- ❒ Uno o más agentes de *meta-learning* (*meta-learning agents*).
- ❒ Parámetros de configuración local (fichero de configuración).

Los elementos globales del sistema son los encargados de la coordinación de las tareas de todos los *Datasites*. Estos componentes son:

- ❒ El gestor de ficheros de configuración o CFM (de *Configuration File Manager*).
- ❒ El interfaz gráfico de usuario y las funcionalidades de animación.

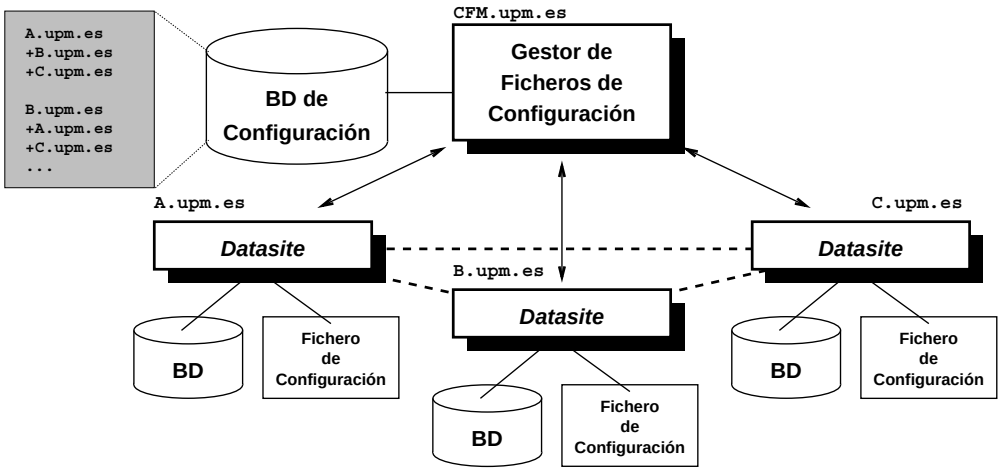


Figura 1.11: Arquitectura de *JAM*

Un escenario de aplicación de la arquitectura, como el representado en la figura 1.11 puede estar conformado por tres *Datasites* (`A.upm.es`, `B.upm.es` y `C.upm.es`). Dichos *Datasites* se registran en el CFM que proporciona las funcionalidades de un servicio de nombres, gestionando los nodos de trabajo activos y su disponibilidad. Una vez registrados, cada uno de los *Datasites* usa sus agentes de aprendizaje locales sobre sus datos generando clasificadores locales. Cada localización es capaz de importar clasificadores de otros *Datasites* y combinarlos por medio de los agentes de

meta-learning. Finalmente, una vez que los *meta-clasificadores* están definidos, se pueden usar sobre cualquier *Datasite* de forma simultanea e independiente. Los ficheros de configuración locales permiten que el administrador de cada *datasite* defina las bases de datos a las que se tiene acceso y cuales son los tipos de agentes (de aprendizaje y *meta-learning*) que van a ser usados. Todas estas operaciones pueden ser monitorizadas y coordinadas desde la herramienta gráfica central que puede representar de forma visual y mediante una animación los procesos de intercambio de clasificadores entre los nodos.

Una de las principales ventajas que presenta esta arquitectura es la posibilidad de extender los componentes de análisis de forma flexible. Las capacidades de análisis están recogidas en los agentes de aprendizaje y *meta-learning*, que están diseñados en base a una jerarquía de objetos que permite la derivación de nuevas clases que implementen el mismo conjunto de funcionalidades definidas por el interfaz de la superclase.

Dentro de las limitaciones de esta arquitectura se resalta su focalización sobre el problema de clasificación únicamente, no soportando distintas consultas ni ninguna otra operación del proceso de KDD. La extensión a otro tipo de consultas afectaría muy notablemente al diseño del sistema pues éste se fundamenta en la teoría de combinación de modelos en base a *meta-learning*, que es aplicable únicamente a problemas de clasificación.

1.4.3.2 PADMA

El sistema *PADMA* (*PArallel Data Mining Agents*) [KHS97a, KHS97b] es fruto del proyecto de investigación con el mismo nombre realizado en el Laboratorio Nacional de Los Álamos y financiado por el Centro Nacional de Aplicaciones de Supercomputación del gobierno norteamericano. El sistema *PADMA* surge inicialmente como una aplicación de análisis de documentos (*text mining*) sobre un entorno distribuido. Los componentes fundamentales del sistema son agentes, responsables del acceso local a los datos, el análisis cooperativo de los mismos y la visualización de los resultados.

Los tres componentes funcionales de la arquitectura *PADMA* representados en la figura 1.12 son:

- Los agentes de *Data Mining*.
- El *facilitador* que coordina las actividades de los agentes.
- El interfaz de usuario.

'ARQUITECTURA DISTRIBUIDA DE CONTROL PARA SISTEMAS CON CAPACIDADES DE DATA MINING'

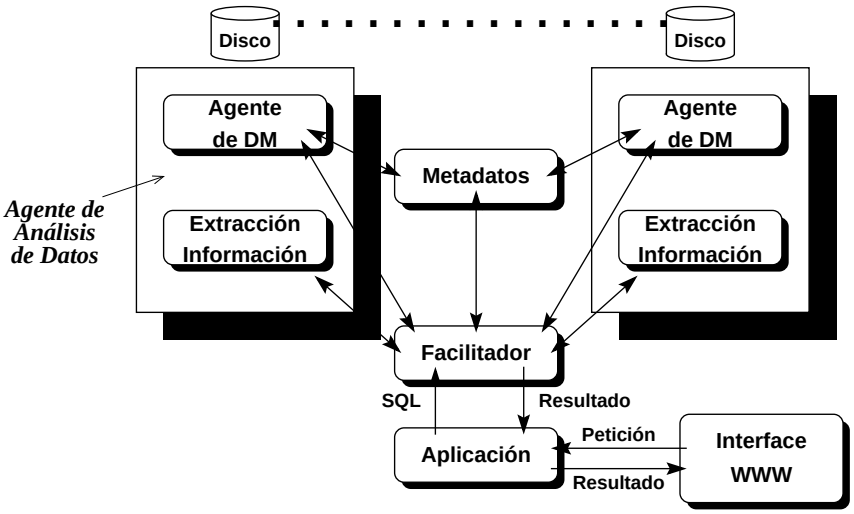


Figura 1.12: Arquitectura de *PADMA*

Los agentes de *Data Mining* son responsables de acceder a los datos en paralelo, así como de extraer de ellos información de alto nivel. El acceso a los datos se realiza de forma directa, sin la mediación de un SGBD. Son los propios agentes de *Data Mining* los que proporcionan además de las funcionalidades de análisis cierto número de operaciones relacionales, tales como ordenación, *joins* (*nested-joins*, *sort/merge-joins* y *hash-joins*) y operaciones de selección y proyección. Las funcionalidades relacionales y de análisis son accedidas mediante un subconjunto del lenguaje de interrogación SQL.

Las tareas del facilitador es la coordinación de los agentes entre sí y la interacción entre el paquete de agentes y el usuario. El interfaz de usuario es vía *web* y basado en sentencias restringidas de SQL.

El desarrollo del sistema *PADMA* ha usado las librerías de usuario PPFS (*Parallel Portable File System*) desarrolladas en la Universidad de Illinois (Urbana-Champaing) [Hub95] como soporte de almacenamiento. Los agentes se han desarrollado en C++ usando MPI (*Message Passing Interface*) como substrato de intercomunicación entre agentes. Por último, el interfaz gráfico se ha desarrollado en Java.

La principal aportación del sistema *PADMA* es su completa integración con la tecnología de bases de datos. Son los propios componentes del sistema los que implementan las funcionalidades de gestión de datos asociadas a un SGBD. Esto permite que el sistema funcione como un SGBD restringido, capaz de crear/destruir tablas y de ciertas operaciones relacionales. Esta aproximación

permite que el proceso de análisis utilice estructuras internas del proceso de gestión relacional de los datos tales como *caches* y memorias intermedias gestionadas por el sistema que almacenan los datos más comúnmente utilizados.

1.4.3.3 Osiris/Papyrus

Papyrus [GKM+98, Ram98, GBST99] es un sistema de *Data Mining* basado en agentes que es capaz de analizar conjuntos de datos distribuidos sobre una red de área extensa (WAN). El proyecto *Papyrus* se ha desarrollado en la Universidad de Chicago Illinois en conjunto con el Centro Nacional de *Data Mining*. Dicho proyecto pertenece al programa M3D2 (*Management and Mining Massive Distributed Data*).

Al igual que el sistema *JAM* se basa en una fundamentación formal denominada *Meta-Clusters*, sobre la cual se asienta el diseño del sistema. En base a dicha teoría, el sistema está compuesto por una serie de agentes cooperativos que generan modelos locales para cada fuente de datos, intercambiando los modelos entre para la obtención del modelo global. Aun basándose en una idea similar a *JAM*, *Papyrus* es un sistema mucho más evolucionado en otros aspectos. Las diferencias entre ambos sistemas son:

- El formato de representación de los modelos generados localmente esta estandarizado en base al lenguaje PMML (*Predictive Modeling Markup Language*) [GBR+99]. Dicho lenguaje es una DTD descrita en XML y representa un formato unificado de intercambio de modelos de clasificación.
- El mecanismo de acceso a los datos es uno de los elementos fundamentales del sistema *Papyrus*. Dicha tarea se realiza por medio del sistema *Osiris*, evolución de las librerías de de objetos persistentes ligeros *PTool* [GQ94]. *Osiris* es una sistema de *Data Warehousing* sobre datos orientados a objetos con una jerarquía de soportes de almacenamiento que van desde memoria física hasta dispositivos terciarios de almacenamiento (cintas). Es un SGBD optimizado para las funcionalidades de análisis y *Data Mining*.

La arquitectura *Papyrus* está compuesta por una serie de entidades denominadas *clusters*. Estos *clusters* contienen los siguientes elementos:

- Un punto de entrada al *cluster* o *Cluster Point of Entry* (CPOE).
- Un *Data Warehouse* local (sobre *Osiris*).
- Un repositorio local de información de recursos.
- Un servidor de agentes.

- ❒ Un motor de *Data Mining* con capacidad para generar modelos en formato PMML.

El diseño del sistema *Papyrus* esta conformado por cuatro módulos diferentes:

- ❒ Interfaz de Usuario: Encarnado en **Puntos de Presencia de *Data Mining*** o *Data Mining Points of Presence* (DMPOP). Estos elementos identifican a los *clusters* disponibles por medio de sus puntos de entrada o CPOEs. El protocolo de comunicación entre los interfaces de usuario y los *clusters* es usado para identificar los servicios de *Data Mining* ofertados por cada uno de dichos *clusters*.

- ❒ Un sistema de Agentes denominado *Bast*. *Bast* proporciona un mecanismo de comunicación que permite que los interfaces de usuario y los *clusters* intercambien información. *Bast* se encuentra implementado en Agent-Tcl [Gra95, Gra97].

- ❒ Un motor de *Data Mining*, encargado de negociar con los interfaces de usuario los servicios de *Data Mining* ofrecidos. Dicha negociación no alcanza a pedir consultas diferentes a las de clasificación, sino que determina únicamente el tipo de algoritmo a usar (inducción, redes neuronales, . . .).

- ❒ Herramientas de Visualización que interpretan los resultados y los representan en formato VRML.

Al igual que el sistema *JAM* la principal desventaja de este sistema es que su fundamentación teórica esta orientada a problemas de clasificación, no proporcionando soporte a otro tipo de consultas ni otras fases del proceso de *Data Mining*. La principal aportación de esta arquitectura, se centra en la inclusión de un SGBD especializado y optimizado para tareas de *Data Mining*. *Osiris* es una herramienta muy eficiente en tareas de recuperación de datos, cargado masivo de información, etc. Sin embargo, funcionalidades como el control de transacciones, la integridad referencial y la optimización de las inserciones individuales en la base de datos no se encuentran soportadas por el gestor.

1.4.3.4 Kensington

El sistema *Kensington* [Ken99b, Ken99a, CGS99, GS99] surgió a partir de un proyecto de investigación del Imperial College de Londres dentro de su centro de Computación Paralela. Posteriormente, el sistema desarrollado ha sido explotado por la empresa del mismo nombre. Este sistema ha llegado a alcanzar un relevante grado de éxito al conseguir el premio dentro del *Terabyte Challenge* [Gro96] en su edición celebrada a raíz del congreso SuperComputing'98 [ACM98].

Kensington es un sistema distribuido basado en una tecnología de componentes desplegados sobre una red de comunicaciones estándar como *Internet*. Uno de los principios fundamentales

del diseño del sistema ha sido la definición de una arquitectura abierta dentro de la cual se puedan acoger nuevos componentes según surjan nuevas necesidades analíticas. Una de las claves del éxito de *Kensington* ha sido su escalabilidad. Esta escalabilidad ha sido definida en base a tres dimensiones:

- El tamaño de los conjuntos de datos.
- El número de fuentes de datos (localizaciones de los datos).
- El número de usuarios que acceden al sistema.

La arquitectura del sistema *Kensington* puede analizarse en base a dos cortes sobre la estructura del sistema. Verticalmente, el sistema esta compuesto por tres niveles, de forma similar a las arquitecturas cliente/servidor del mismo tipo. Esta división reconoce:

- **Funcionalidades Cliente**: Acceso remoto, interacción para definir las consultas, visualización de resultados, ...
- **Servidor de Aplicaciones**: Gestión de sesiones, almacenamiento persistente de los objetos del sistema (usuarios, modelos, tareas, ...), ejecución de tareas y coordinación y gestión de datos y pre-procesamiento.
- **Servidores de Tercer Nivel**: SGBD o sistemas de computación de altas prestaciones (HPCs).

El desglose horizontal, por su parte, está basado en el concepto de *arquitectura de componentes software*. Dichas arquitecturas están compuestas por elementos modulares que pueden ser combinados para la definición de sistemas de forma flexible. Las tecnologías usadas en el diseño de *Kensington* son *Java Enterprise Architecture* y CORBA, por proporcionar diseños abiertos, en lugar de soluciones propietarias (DCOM).

La arquitectura *Kensington*, representada en la figura 1.13, está compuesta por componentes EJB (*Enterprise JavaBeans*) para las funcionalidades de gestión distribuida. La ejecución de consultas sin embargo es realizada por nodos de computación de altas prestaciones (HPC) mediante algoritmos escritos en C y MPI (*Message Passing Inteface*) encapsulados en interfaces CORBA. Las funcionalidades de gestión se encuentran centradas en un servidor de aplicaciones *EJBServer* que tiene los siguientes elementos:

- **UserSpace EJB**: Gestiona los objetos persistentes del sistema, tales como los usuarios, modelos, tareas, etc. La gestión de estos objetos permite un tratamiento completo de operaciones y resultados dependiente de usuarios y permisos.

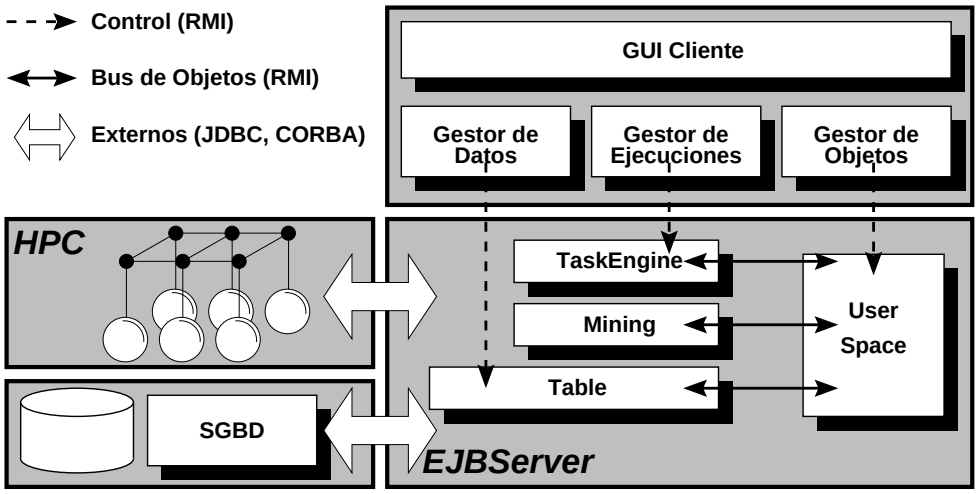


Figura 1.13: Arquitectura de *Kensington*

- **TaskEngine EJB**: Es el punto de contacto entre los interfaces de usuario y la ejecución de las consultas. Proporciona funcionalidades de preprocesamiento, generación y evaluación de modelos y recuperación de resultados para el cliente.
- **Mining EJB**: Maneja los interfaces de los algoritmos y componentes de proceso del sistema. Interactúa con nodos de computación de altas prestaciones mediante CORBA para solicitar la ejecución de los mismos.
- **Table EJB**: Responsable de las operaciones de importación y exportación de resultados vía JDBC.

Una instalación de *Kensington* puede definir varios servidores de aplicaciones (*EJBServers*) para interactuar con diferentes localizaciones de fuentes de datos o distintos nodos de computación de altas prestaciones. En conjunto el sistema completo puede estar distribuido a nivel de funcionalidades en muchas diferentes localizaciones.

La especial atención que el diseño del sistema confiere al número de usuarios del sistema (y por tanto de procesos de consulta diferentes) como factor de carga y la flexibilidad en la incorporación de componentes son las principales aportaciones de esta arquitectura. La implementación de la misma es también innovadora, pues hace uso de tecnologías de vanguardia (*Enterprise JavaBeans* y CORBA) que permiten al sistema la interacción con otras herramientas (paquetes estadísticos como S-plus) que comparta dichos interfaces. Como principal carencia, la arquitectura de *Kensington* resalta las funcionalidades de gestión muy por encima de las funcionalidades de ejecución de consultas las cuales descansan sobre los nodos de computación de altas prestaciones para los cuales *Kensington* no define ninguna organización.

1.4.4 Taxonomía de los Sistemas de *Data Mining* Distribuido

Recapitulando el análisis realizado en esta sección sobre los sistemas de *Data Mining* distribuido, éstos se pueden clasificar de acuerdo a varios factores:

① La aplicación de la distribución al problema, pudiendo darse:

- ❒ *Data Mining* Colectivo: Análisis de datos originalmente dispersos de los que se desea extraer un modelo común.
- ❒ *Data Mining* Paralelo: Utilización de técnicas de cálculo masivo para resolver un problema de *Data Mining* por medio de procesamiento distribuido.

② Integración con las fuentes de datos, al igual que en el caso de los sistemas de *Data Mining* centralizado:

- ❒ Sistemas Integrados: Proporcionan las funcionalidades de *Data Mining* al mismo nivel que las operaciones relacionales o por medio de sistemas gestores especializados para el sistema.
- ❒ Sistemas No Integrados: Descansan sobre un SGBD que gestiona los datos. Sobre dicho SGBD se definen las operaciones del sistema.

③ Consultas de *Data Mining* soportadas. Muchos sistemas de *Data Mining* distribuido sólo dan soporte para consultas de clasificación. Sin embargo, otros proporcionan casi todas las consultas (asociación, *clustering*, etc.) así como el resto de operaciones del proceso de KDD, como son la preparación y la limpieza de los datos.

En base a estas características se pueden representar los sistemas estudiados dentro de una tabla (Tabla 1.2) indexada por sus funcionalidades. Ciertos espacios de esta representación, dentro de la faceta de *Data Mining* Paralelo, pueden ser completados por medio de aportaciones dentro del campo de algoritmos y no de sistemas propiamente dichos. Los ya citados algoritmos de asociación [AS96a, CNFF96, HKK97, Zea97], patrones secuenciales [SK98, Zak00] o clasificación [SAM96a, JKK98, ZHA99], son un ejemplo. La clasificación del sistema *Papyrus* como perteneciente a la rama de *Data Mining* Colectivo se debe más a su aplicación (dentro del campo de datos procedentes de experimentos físicos o datos médicos) que por la propia arquitectura del mismo. De la misma forma *Kensington* puede ser usado para *Data Mining* Colectivo bajo ciertas configuraciones.

Un factor relevante en el desarrollo de los sistemas de *Data Mining* distribuido presentados en este trabajo son las tecnologías usadas en su implementación:

- ❒ *JAM*: Diseñado mediante un modelo de agentes.

	Data Mining Paralelo		*Data Mining* Colectivo	
	Integrado	**No Integrado**	**Integrado**	**No Integrado**
Sólo Clasificación	Papyrus*		Papyrus	JAM
Varias Consultas		Kensington	PADMA	Kensignton*

Tabla 1.2: Características de los sistemas de *Data Mining* Distribuido estudiados.

* Las caracteristicas de Papyrus y Kensington permite realizar tanto tareas de *Data Mining* colectivo como *Data Mining* paralelo.

- *PADMA*: Diseñado también por medio de agentes. Desarrollado sobre el sistema de fichero PPFS, interfaz MPI y tecnologías Java.
- *Papyrus*: Diseñado sobre el sistema de agentes *Bast*. Usa el soporte de datos *Osiris*, un lenguaje de intercambio de modelos (PMML) y se encuentra implementado en *Agent-Tcl*.
- *Kensignton*: Desarrollado como un sistema distribuido sobre las tecnologías *Entreprise Java Beans* y *CORBA*. Los *clusters* de procesamiento masivo ejecutan algoritmos implementados sobre MPI.

1.5 Conclusiones

Como se ha podido comprobar, los sistemas actuales de *Data Mining* son aplicaciones complejas completamente integradas dentro de los sistemas de información de las organizaciones. Los sistemas centralizados se corresponden con aplicaciones comerciales de amplia utilización. La creciente demanda en este sector ha hecho que las principales casas comerciales dentro del sector de bases de datos desarrollasen o adquiriesen productos de *Data Mining* que ofertar a sus clientes.

Por su parte, los sistemas distribuidos de *Data Mining* están aun a un nivel experimental, en el cual centros de investigación y universidades son los encargados de su desarrollo. De todas formas, se espera su explotación comercial tan pronto como la tecnología se asiente y los sistemas en desarrollo maduren.

1.5.1 Características Alcanzadas

Las características más relevantes que se han estudiado dentro de los sistemas centralizados de *Data Mining* han sido:

- **Integración con los SGBD**: Existiendo desde sistemas débilmente integrados hasta sistemas

que ejecutan parte de los procesos de análisis dentro del SGBD (ciertas funciones de *IMiner* de IBM).

- **Extensibilidad**: La gran mayoría de sistemas comerciales incorporan APIs de desarrollo de nuevos algoritmos u otros mecanismos similares para ampliar sus funcionalidades. Estos mecanismos son más o menos restrictivos dependiendo de filosofía de proceso de *Data Mining* que soporte la herramienta.

1.5.2 Retos Pendientes

Si bien los nuevos sistemas a desarrollar deben conservar las funcionalidades alcanzadas por las soluciones actuales, quedan aun retos a conseguir en las futuras generaciones de sistemas.

Es importante resaltar que la problemática de la **flexibilidad de control**, tal y cómo se enunció en la introducción de este trabajo, no se encuentra actualmente soportada por ningún sistema de *Data Mining* ni comercial ni experimental. Este punto representa la principal aportación de este trabajo de investigación. Los sistemas vistos representan soluciones estáticas a la resolución de consultas de *Data Mining*. Su comportamiento está definido en la fase de diseño del mismo y sólo se puede cambiar rediseñando el sistema.

Dependiendo de ciertas consideraciones de control un mismo sistema puede resultar especialmente eficiente para ciertos tipos de operaciones y para otros no. La posibilidad de modificar los parámetros de control permitiría ajustar los sistemas a los requisitos de rendimiento y entornos de aplicación necesarios. Esta faceta de los sistemas de *Data Mining* es la que se ha juzgado como eje central de este trabajo de investigación. La consecución de un diseño capaz de permitir la modificación de sus criterios de control sin rediseñar el sistema o incluso sin detenerlo es el objetivo de esta aportación.

1.5.3 Sistemas de *Data Mining* Distribuidos

Los sistemas de *Data Mining* distribuidos incorporan a las características anteriores la **distribución**, bien orientada al uso eficiente de *clusters* de computadores para ejecución de los algoritmos, o para proporcionar soluciones a entornos de datos originalmente distribuidos. Las aportaciones de estos sistemas de *Data Mining* se basan principalmente en dos tecnologías distribuidas:

- *Tecnologías de Componentes Distribuidos*: Como sustrato de comunicación entre los elementos distribuidos del sistema. Dentro de este enfoque destacan las tecnologías de *middleware* CORBA y DCOM.

- *Sistemas Multiagente*: Que define las pautas de diseño de los componentes autónomos y cooperativos que participan en la resolución del problema distribuido.

El estudio de ambas técnicas se presenta en los dos siguientes capítulos, con el fin de exponer todos los elementos que conforman el planteamiento y posterior solución del problema.

Capítulo 2

ARQUITECTURAS DE COMPONENTES DISTRIBUIDOS

Índice General

2.1 Introducción

La aplicación de soluciones distribuidas a diferentes problemas computacionales ha originado la aparición de multitud de técnicas. Estas técnicas plantean diferentes estrategias para la tarea de división del problema en varios subproblemas cuya resolución pueda realizarse paralelamente, en

diferentes nodos de computación y que garantice la interoperabilidad entre los diferentes elementos del sistema.

La necesidad de desarrollar sistemas distribuidos de forma eficiente y con una alta productividad ha dado lugar a la aparición de lo que se denomina entornos de desarrollo distribuidos. Dichos entornos facilitan la implementación de aplicaciones distribuidas, proporcionan una serie de librerías de desarrollo, elementos prediseñados de los sistemas y funcionalidades varias. En conjunto, todas estas herramientas permiten que el desarrollo de sistemas distribuidos se realice sobre un nivel inferior que garantiza una serie de premisas. Tales premisas, incluyen la ocultación de la complejidad de comunicación sobre una red de conexión, los mecanismos de conversión adecuados para que arquitecturas de computadoras diferentes se comuniquen (representaciones canónicas de tipos de datos) y los mecanismos de nombrado y de localización de componentes.

Los entornos de desarrollo distribuido han abarcado aportaciones dentro de un amplio rango de herramientas. Desde las soluciones simples que proporcionan únicamente mecanismos de comunicación y funciones explícitas de conversión de tipos, como es el caso de las librerías de comunicación BSD *sockets* de Berkeley [CS93b], pasando por mecanismos transparentes de invocación de procedimientos remotos como RPCs o de métodos como RMI de Java, hasta entornos completos basados en los anteriores que adicionalmente incorporan elementos del sistema distribuido que proporcionan servicios útiles para cualquier desarrollo, como puede ser el entorno DCE de OSF (actuálmente Open Group) [OSF94, OSF95]. Sin embargo, en la actualidad se ha producido una importante revolución en lo que se denominan entornos de componentes distribuidos.

2.1.1 ¿Qué son los Componentes Distribuidos?

El concepto de componente se deriva de la noción original de objeto. El desarrollo de la tecnología de orientación a objetos se ha centrado prioritariamente en dos de los conceptos originarios de la teoría de objetos. Estos conceptos son la encapsulación y la reutilización de código. Estos dos objetivos han sido alcanzados tanto por las metodologías como por los lenguajes orientados a objetos desarrollados. Sin embargo, el concepto de componente recupera una de las ideas originales que propone la teoría de objetos, que es la interoperabilidad (*pluggability* en inglés). Esta característica se centra en la fácil integración de un elemento (componente) dentro de un sistema, exigiendo de estos elementos el mayor número de facilidades en ese sentido.

De los componentes, se exige que:

- Las características de implementación sean irrelevantes. La implementación de un compo-

nente en un lenguaje u otro no debe afectar a su interacción con el sistema.

- La interfaz de interacción con un componente debe descubrirse de forma dinámica. Esto permite que durante la ejecución del sistema éste conozca las funcionalidades proporcionadas por cada nuevo componente.
- La localización física del componente debe ser transparente al sistema. En este caso se habla de componentes distribuidos, los cuales pueden interactuar con otros componentes del sistema, dentro del mismo espacio de direcciones (el mismo programa), desde otro programa dentro de la misma computadora, o incluso desde otras computadoras.

El concepto de componente ha supuesto una pieza fundamental en el desarrollo de sistemas distribuidos. Todas las características de un componente son de extrema utilidad en el desarrollo de aplicaciones formadas por elementos distribuidos que deben cooperar en la resolución de un problema.

Las tecnologías más relevantes dentro del campo de componentes distribuidos son, CORBA de OMG, DCOM de Microsoft y *Enterprise JavaBeans* de Sun. Dichas tecnologías se revisan a continuación, puesto que la aportación práctica de esta tesis parte del desarrollo de una arquitectura de un sistema distribuido, basada en alguna de estas tecnología de componentes, o cualquier futura tecnología de similares prestaciones.

2.2 Arquitectura OMA de OMG

En Mayo de 1989, ocho empresas: 3Com Corporation, American Airlines, Canon Inc., Data General, Hewlett-Packard, Phillips Telecomunications N.V., Sun Microsystems y Unisis Corporation fundaron el consorcio OMG (*Object Management Group*) [SS95]. En la actualidad OMG está compuesto por varios centenares de miembros (800+), entre las compañías más importantes del sector y los principales centros de investigación, salvo la notable ausencia de Microsoft. El consorcio OMG se planteó como un grupo de trabajo de varios sectores empresariales para promover la teoría y práctica de la tecnología orientada a objetos dentro del desarrollo software, estos objetivos incluyen el establecimiento de unas directrices para la industria así como la confección de especificaciones que permitan definir un marco para el desarrollo de aplicaciones. Una de las principales metas de OMG es la reusabilidad, portabilidad e interoperabilidad del software orientado a objetos sobre entornos distribuidos heterogéneos.

Los esfuerzos de OMG en dirección hacia el desarrollo de una tecnología para el desarrollo de aplicaciones distribuidas se encuentran plasmadas en la arquitectura OMA (*Object Management*

Architecture) [OMG00c], que proporciona un marco conceptual sobre el cual se basan el resto de especificaciones.

2.2.1 Componentes de la Arquitectura OMA

La arquitectura OMA está compuesta por un *Modelo de Objetos* [OMG99h], que especifica cómo los objetos están definidos y por un *Modelo de Referencia* que indica cómo estos objetos interaccionan entre si.

2.2.1.1 Modelo de Objetos (*Object Model*)

Este modelo define cómo deben estar descritos los objetos que se encuentran distribuidos sobre un entorno heterogéneo. El Modelo de Objetos indica cómo debe encapsularse la implementación (independientemente del lenguaje) para poder invocarse por otras aplicaciones (denominadas *clientes*). Estos objetos encapsulados se encuentran disponibles para el resto de elementos del entorno por medio de lo que se denominan *interfaces*. Estos *interfaces* están clara y formalmente expresados en el lenguaje IDL (*Interface Definition Language* [OMG99g]) y representan la información necesaria para que los clientes soliciten servicios al objeto.

Los elementos más importantes del modelo de objetos de OMA, son:

- *Tipos*: OMA define *tipo* como la entidad que asociada a un predicado (una función o una variable) determina el conjunto de valores posibles que dicho predicado puede tomar. El modelo de objetos clasifica todos los posibles tipos en tres grupos, *tipos básicos*, *tipos compuestos* y *referencias a objetos*.

- *Interfaces*: Un interfaz dentro de este modelo de objetos se define como una descripción de todas las operaciones que es posible realizar sobre un determinado objeto. Por extensión, dentro de la arquitectura OMA, los objetos se consideran representados por sus interfaces, por lo tanto cualquier implementación que satisfaga dicho interfaz (soporte todas las operaciones descritas en el interfaz) puede ser adaptada como un objeto.

  Los interfaces se encuentran definidos por medio del lenguaje IDL (*Interface Definition Language*) [OMG99h] que es un lenguaje orientado a la descripción de operaciones (y no a la implementación de las mismas). El lenguaje IDL proporciona diversas características orientadas a objetos, entre ellas mecanismos de herencia múltiple, por medio de los cuales se pueden diseñar objetos que satisfagan varios interfaces. Debido a que IDL no está orientado a la implementación de objetos y sólo a su descripción, así como por motivos de flexibilidad, existen diversas traducciones (*mappings*) de IDL a diferentes lenguajes de programación: IDL–Java [OMG99p, OMG99n] IDL–Ada [OMG99a], IDL–C [OMG99b], IDL–C++ [OMG99c],

IDL–Smaltalk [OMG99q] o IDL–COBOL [OMG99d], encontrándose en fase de definición otros muchos más (Lisp [OMG00h] o PL/I [OMG99o]).

- *Operaciones*: Una *operación* dentro del contexto de OMA, es una especificación que describe un servicio que puede ser solicitado a un objeto. Una operación está asociada a un identificador (único dentro de un mismo interfaz), así como a una serie de parámetros y a un valor de retorno; de forma similar a como se define una función en un lenguaje procedimental. Adicionalmente, las operaciones (descritas en IDL) pueden incluir información relativa a: (i) excepciones que pueden ser levantadas bajo ciertas circunstancias anómalas, así como sus respectivos argumentos; (ii) información de contexto que puede afectar a la ejecución de la operación y (iii) semántica de la ejecución, que indica si se asegura que la operación se realiza (o se levanta una excepción, modo *at-most-once*[1]), o si por el contrario, el objeto no garantiza poder realizarla y no notificará al cliente sobre el éxito o fracaso de la misma (modo *best-effort*). Todos estos componentes (argumentos, valor de retorno, excepciones, información de contexto y semántica de la operación) componen lo que se denomina firma (*signature*) de la operación. La sintaxis de una firma completa es:

SIGNATURE FORMAT:

```
[oneway]   type   identif    ([in|out|inout]param_1 , ...
                             [in|out|inout]param_L)
                  [raises    (except_1 , ...
                             except_M)]
                  [context   (name_1 , ...
                  name_N)]
```

2.2.1.2 Modelo de Referencia (*Reference Model*)

Como complemento al Modelo de Objetos, OMG define un segundo grupo de especificaciones denominadas Modelo de Referencia. Este modelo describe los diferentes tipos de elementos que componen la arquitectura así como las distintas clases en las que se dividen los interfaces de los objetos que componen la misma. Todos estos elementos pueden verse en la Figura 2.1:

- **ORB** (*Object Request Broker*): Este elemento hace las funciones de un bus de interconexión entre los diferentes objetos que se comunican en la arquitectura, proporcionando los mecanismos de adaptación y conversión de llamadas a operaciones entre clientes y objetos, así como otras funcionalidades básicas para el funcionamiento de la arquitectura.

[1] El modo *at-most-once* puede ser invocada en modo síncrono, bloqueando al cliente hasta que la operación finaliza (con éxito o excepción) o en modo diferido, que implica que cuando la operación concluye, el cliente es notificado por medio de un mensaje asíncrono.

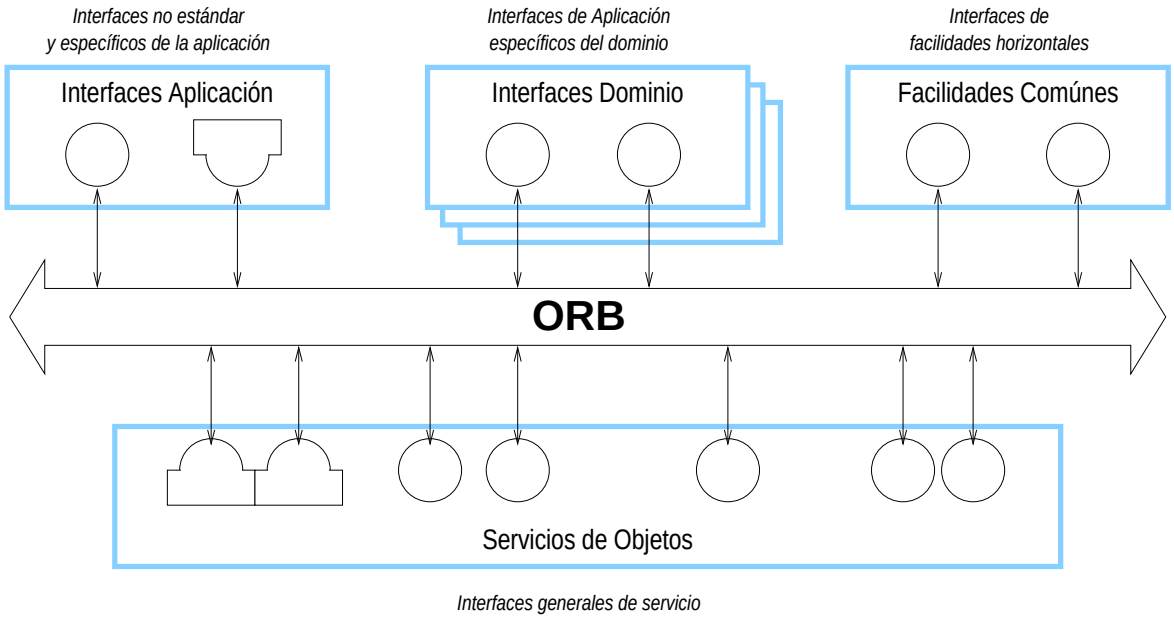


Figura 2.1: Elementos de la arquitectura OMA

Este elemento se puede encontrar implementado como:

- ❍ Un API de desarrollo o librerías enlazadas por cada cliente u objeto que pertenezca al entorno.
- ❍ Un servicio del sistema, compartido por todos los procesos de la computadora.
- ❍ Un nivel determinado dentro de la pila de comunicaciones del sistema, incluida dentro del núcleo del sistema operativo de la computadora.

Todos los clientes, así como los objetos de servicio, utilizarán la implementación del ORB para registrarse y hacer uso del bus de comunicación. Tanto las implementaciones de este elemento, como las de cualquiera de los componentes de OMA no se realizan por OMG (OMG sólo define especificaciones).

- ❒ **Servicios de Objetos** (*Object Services*): Estos servicios son un conjunto de interfaces, claramente especificados que proporcionan funcionalidades elementales de propósito general y especialmente orientadas al desarrollo de otros objetos dentro del entorno. Independientemente de lo específico de las funciones de estos servicios la combinación de sus funcionalidades permite el desarrollo de aplicaciones basadas en componentes distribuidos altamente complejas.

Los interfaces de este grupo se encuentran especificados en el documento [OMG98a]. Los

servicios actualmente especificados por OMG son: Servicio de Nombres, Servicio de Gestión de Eventos, Servicio de Notificación, Servicio de Objetos Persistentes, Servicio de Ciclo de Vida, Servicio de Externalización, Servicio de Relación, Servicio de Consulta, Servicio de Transacción, Servicio de Consulta, Servicio de Licencia, Servicio de Propiedad, Servicio de Tiempo, Servicio de Seguridad, Servicio de Negociación y Servicio de Colección.

- **Facilidades Comunes** (*Common Facilities*): Estos interfaces conforman el segundo grupo que está definido como herramientas orientadas al usuario y aplicables a varios dominios o entornos de aplicación (utilidades horizontales). Estos interfaces no están planteados como funcionalidades dirigidas al desarrollo de aplicaciones sino como herramientas directamente utilizables por usuarios, el ejemplo más claro es el entorno DDCF [OMG95] (*Distributed Document Component Facility*) para la gestión distribuida de documentos, basado en *OpenDoc* de Apple Computer Inc.

- **Interfaces de Dominio** (*Domain Interfaces*): Este grupo de interfaces proporciona servicios similares a los Servicios o Facilidades antes citados pero dentro de un campo de aplicación muy concreto, como puede ser: Aplicaciones Médicas (CORBAMed [OMG99l, OMG99m, OMG99k, OMG98i, OMG98j]), Telecomunicaciones (CORBATelecoms, [OMG98g, OMG98f, OMG98b]), Sistemas de Información Geográficos (CORBAGis [OMG97t]) o Comercio Electrónico [OMG99j], . . .

- **Interfaces de Aplicación** (*Application Interfaces*): Este último grupo de interfaces se desarrollan para cada aplicación en concreto. Dentro de este grupo OMG no ha publicado (ni se cree que pueda publicar) ninguna especificación, por tratarse de interfaces específicas para cada aplicación o sistema que se desarrolla siguiendo la arquitectura OMA.

Tanto los nuevos objetos desarrollados para aplicaciones concretas como los servicios y el resto de interfaces de la arquitectura OMA, representan fragmentos de código ejecutable cuyas funcionalidades se encuentran definidas por medio de un interfaz, expresado en IDL. Dicho interfaz, representa las funcionalidades que dicho objeto proporciona y, por lo tanto es la información que el cliente desea conocer, junto con la referencia al objeto para solicitar dichas funcionalidades.

2.2.2 ORB

Como componente clave de la arquitectura OMA se encuentra el *Object Request Broker*(ORB). Este elemento se encarga de hacer llegar las peticiones y retornar las respuestas a los clientes que las solicitaron. La especificación principal de OMG sobre este componente es CORBA (CORBA 2.0 [OMG97a], CORBA 2.1 [OMG97b], CORBA 2.2 [OMG98d] y CORBA 2.3 [OMG98e]). El ORB se encuentra a su vez dividido en una serie de elementos (que pueden verse en la Figura 2.2) que se

agrupan en:

- ❒ Núcleo ORB.
- ❒ Adaptador de Objetos.
- ❒ Repositorio de Interfaces.
- ❒ Repositorio de Implementaciones.
- ❒ *Stub* y *Skeleton* IDL.
- ❒ Invocación Dinámica.

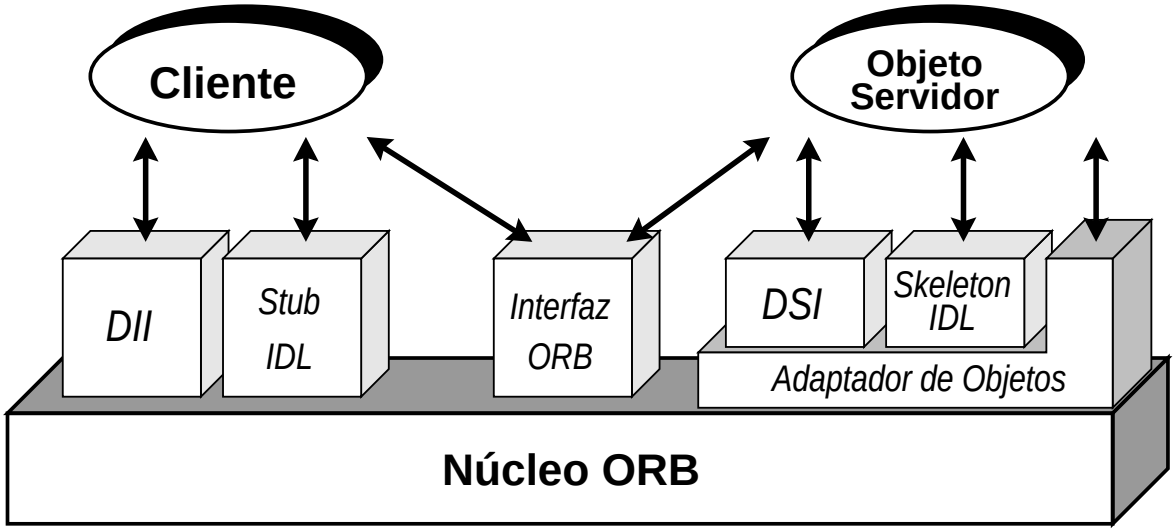


Figura 2.2: Componentes de un ORB

2.2.2.1 Núcleo ORB

Este nivel inferior del ORB se encarga de construir un "*bus* o canal de objetos" sobre el cual funcionan el resto de componentes. El principal objetivo del núcleo es proporcionar de forma transparente una serie de funcionalidades de comunicación que deben ocultar:

① *La localización de los objetos*: El cliente no tiene la necesidad de conocer dónde reside el objeto que le proporciona el servicio. Este puede estar dentro del mismo proceso, en la misma máquina o en otra computadora de la red.

② *La implementación del objeto*: El cliente no sabe el lenguaje o características de implementación del objeto que le sirve. El ORB oculta factores dependientes del hardware, sistema operativo, lenguaje de programación, . . .

③ *El estado del objeto*: Cuando el cliente realiza una petición a un objeto de servicio no debe preocuparse de si el objeto está activado o no, siendo el ORB el encargado de arrancar dicho servicio si no se encontrase activo en ese momento.

④ *Los mecanismos de comunicación*: El diseño y desarrollo de objetos que se comunican por medio del ORB es completamente independiente de la tecnología y protocolos de comunicación subyacentes. El núcleo del ORB se encarga de hacer uso de redes basadas en TCP/IP, mecanismos de comunicación en base a memoria compartida interproceso o llamadas a métodos locales indistintamente.

Todas estas características permiten que el núcleo del ORB sea capaz de comunicar dos objetos (cliente y servidor) aislándose de todas las peculiaridades indicadas anteriormente. La comunicación se basa en la invocación de determinados métodos del objeto servidor por parte del cliente. Para poder realizar dichas operaciones es necesario que el cliente disponga de una *referencia de objeto* que representa una notación (opaca para el cliente) que permite localizar al objeto servidor a lo largo de la red. Esta representación define a los objetos de forma única y su valor no puede ser alterado, teniendo únicamente significado para el ORB. Por necesidades de interoperabilidad, el formato de representación de las *referencias de objetos* se ha estandarizado para determinados mecanismos de comunicación, existiendo de esta forma especificaciones del formato para redes TCP o entornos DCE.

En relación al problema de la interoperabilidad, OMG ha definido un protocolo denominado UNO GIOP (*Universal Network Objects, General Inter-ORB Protocol* [OMG97c]) para especificar cómo han de comunicarse dos ORBs de diferentes fabricantes por medio de un red genérica. Esta ampliación acordada con posterioridad a la primera especificación de CORBA (CORBA 1.0 en 1.991) está dirigida a superar las limitaciones de dicha especificación a la hora de interconectar objetos situados sobre ORBs de distintos fabricantes. Derivado de GIOP, existen dos protocolos que especifican de forma concreta la forma de interconectar dos ORBs, sobre tecnologías de comunicación concretas:

- IIOP (*Internet Inter-ORB Protocol*): Para redes TCP/IP. Versión 2.0 [OMG97a], versión 2.1 [OMG97e] y versión 2.2 [OMG98h].
- DCE CIOP (*DCE Common Inter-ORB Protocol* [OMG97s]): Sobre el entorno DCE de OSF [OSF94, OSF95].

2.2.2.2 Adaptador de Objetos

Este elemento del ORB, representa el nexo de unión entre los objetos (tanto específicos de una aplicación , como los servicios y facilidades estándar) y el núcleo de ORB. Este componen-

te sigue el modelo de funcionamiento del patrón Adaptador, tal y como se encuentra definido en [GHJV95, GHJV93] y actúa como mediador y traductor entre los clientes y los objetos, de forma que los primeros encuentren el interfaz deseado en sus llamadas.

La existencia de este componente hace posible que en el entorno existan de forma organizada miles de objetos, realizando una serie de tareas de vital importancia:

- *Registrar Objetos*: Un adaptador de objetos debe permitir (por medio de una serie de funciones), que ciertos fragmentos de código sean registrados ('*dados de alta*') como implementaciones de objetos.
- *Generar Referencias de Objetos*: Como resultado de registrar un objeto, el Adaptador proporciona una referencia única de este objeto usada por los clientes para realizar sus peticiones al objeto.
- *Activar Procesos de Servicios*: Si es necesario, el Adaptador de Objetos debe de ser capaz de arrancar un proceso de servicio dentro del cual se cobijarán los objetos y que permitirá la recepción de peticiones por parte de los clientes.
- *Activar Objetos*: Las implementaciones registradas como objetos deben de ser activadas (si no estaban activadas ya), cuando una petición llegue dirigida a ellas.
- *Demultiplexar las Peticiones*: El Adaptador debe cooperar con el núcleo del ORB para asegurarse que las peticiones pueden recibirse por múltiples conexiones sin bloquear de forma indefinida una única conexión.
- *Propagar Peticiones*: Debido a que, por lo general, existe más de un objeto dentro de un mismo Adaptador, éste debe ser capaz de localizar para cada petición el objeto de los registrados hacia el que va dirigida, así como el método del mismo al que se solicita. Una vez determinada, debe realizar la llamada a la función asociada a la operación del interfaz solicitada.

Los Adaptadores, además de las funciones que proporcionan a los objetos situados sobre él, deben estar *escuchando* de forma permanente las peticiones que los clientes puedan realizar sobre los objetos en él registrados, redirigiendo dichas peticiones a la función del objeto adecuada.

OMG, no restringe la posibilidad de definir diferentes tipos de adaptadores, siempre y cuando respondan a las funcionalidades antes citadas, pero para evitar la excesiva proliferación de diferentes modelos de adaptadores, especificó el Adaptador BOA (*Basic Object Adaptor*) [OMG97d]. El objetivo principal de BOA es que se tratase de una especificación lo suficientemente libre como para poder ser realizada para diferentes lenguajes; pues hay que tener en cuenta que la forma de

acceder a las funcionalidades del Adaptador es por medio de un conjunto de funciones (una API) y dichas funciones deben ser invocadas por la implementación del objeto para, por ejemplo registrarse, esto implica que para desarrollar objetos en C++, por ejemplo, se necesita disponer de un Adaptador con una API, para C++. Esta característica de la especificación de BOA, plantea, sin embargo una desventaja, pues las implementaciones que de ella se han realizado tienden a ser muy dependientes del lenguaje, por lo tanto, los objetos diseñados para un determinado adaptador BOA, son difícilmente portables a otro Adaptador.

Esta problemática, queda resuelta con la especificación de POA (*Portable Object Adaptor*) [OMG99f], que está enfocada para solucionar, entre otros, los siguientes problemas:

- *Portabilidad*: Las especificaciones de POA, son mucho más concretas que las de BOA y permiten la portabilidad de objetos entre diferentes implementaciones de Adaptadores.
- *Persistencia*: Permitiendo que los objetos registrados en el Adaptador mantengan la consistencia (desde el punto de vista de un cliente que posee una referencia al objeto) entre múltiples ejecuciones del servidor.
- *Transitoriedad*: Haciendo posible la definición de objetos transitorios de una forma simple y eficiente.
- *Activación Transparente*: Arrancando los objetos de forma transparente.
- *Multiplicidad*: Dejando la posibilidad de disponer de varias instancias del adaptador dentro de un mismo servidor, organizados por medio de jerarquías de Adaptadores.
- *Control del Comportamiento*: Permitiendo a las implementaciones de los objetos definir cuál debe ser el comportamiento del Adaptador a la hora de tratarlos. Esta funcionalidad se proporciona por medio de una serie de políticas del adaptador que pueden ser redefinidas por el objeto, permitiendo, por ejemplo, la realización de peticiones al objeto de forma bloqueante o concurrente.

2.2.2.3 Repositorio de Interfaces

El Repositorio de Interfaces, es el componente del ORB encargado de proporcionar un almacenamiento persistente de las definiciones de interfaces previamente expresados en IDL. Este componente es clave dentro del funcionamiento del ORB, pues todos los objetos existentes, se acceden en virtud de la información que de ellos se dispone, es decir del conocimiento de las operaciones que proporcionan.

Las funciones que proporciona el Repositorio de Interfaces, son de almacenamiento, gestión y consulta de forma distribuida de los interfaces en él recogidos. La inserción de nueva infor-

mación dentro del Repositorio de Interfaces se realiza por parte de los propios objetos, bien de forma estática (*Skeleton*) o de forma dinámica, por medio de una serie de funciones del propio Repositorio. La consulta de los interfaces almacenados en este Repositorio es pública, aunque los principales elementos que la consultan son:

- ❒ Los clientes cuando usan los procedimientos de invocación dinámica (ver 2.2.2.6).
- ❒ El propio ORB, cuando verifica que todas las operaciones del interfaz están implementadas en el objeto, de forma que pueda atender todas operaciones, satisfaga los grafos de herencia de otros interfaces y puedan ser compartidos cuando varios ORBs se interconectan.
- ❒ Clientes específicamente diseñados para explorar el Repositorio de Interfaces, como pueden ser herramientas gráficas de interacción con el entorno, que representen de forma visual los elementos del entorno en un *toolbox* o caja de herramientas. Estas aplicaciones pueden considerarse navegadores de la información de los Repositorios de Interfaces.

La información almacenada en el Repositorio de Interfaces es, el nombre de los interfaces y para las operaciones de dichos interfaces, los tipos de los valores retorno y de los argumentos, así como los identificadores y nombres de los mismos. Adicionalmente, pueden contener cualquier otro tipo de información que resulte útil y necesaria para las funciones del mismo. Esto indica que un mismo ORB puede disponer de más de un Repositorio de este tipo, esto está justificado cuando existen dos tipos de necesidades de los objetos del ORB o cuando un grupo de objetos contiene otro tipo de información (puede ser el caso de Bases de Datos Orientadas a Objetos). OMG no define cómo ha de permanecer coherente la información de varios Repositorios (evitando la definición del mismo interfaz en varios de ellos con información contradictoria), por el contrario si define las verificaciones que han de realizarse dentro de un mismo Repositorio para asegurarse de que, por ejemplo no existen dos interfaces con el mismo nombre.

El Repositorio de Interfaces, al igual que cualquier otro componente dentro del entorno, dispone de un interfaz propio [OMG99e], también expresable por medio de IDL. Este interfaz, sin embargo, no se encuentra almacenado dentro de este mismo Repositorio, sino que se conoce por parte de todos los elementos del entorno (objetos y clientes) que han de consultarlo.

2.2.2.4 Repositorio de Implementaciones

Junto con el Repositorio de Interfaces, la especificación de CORBA, menciona la existencia de otro Repositorio, denominado *Repositorio de Implementaciones*, aunque no proporciona una especificación clara del mismo. Teóricamente este Repositorio mantiene información referente a las implementaciones de objetos registradas en el adaptador, tales como mecanismos de activación o el propio identificador de objeto.

2.2.2.5 *Stub* y *Skeleton* IDL

La forma habitual de establecer una interacción entre un cliente y un objeto por medio de CORBA es usando los mecanismos de invocación estática. Para ello, se define por medio de IDL el interfaz del objeto a consultar, por ejemplo un *buffer*:

EJEMPLO BUFFER.IDL:

```
interface Buffer {
        void    insertar   (in string elemento);
        string  recuperar  ();
        short   tamano     ();
};
```

Partiendo de esta definición, por medio de una serie de herramientas (compiladores IDL), se obtiene la parte cliente y la parte servidor asociada a dicho interfaz:

- La parte cliente se denomina *stub* y está compuesta por un conjunto de funciones (las mismas definidas en el interfaz) encargadas de transformar los argumentos de la operación a un formato de comunicación (*marshalling*) y realizar la petición al ORB. Si dicha petición debe retornar algún valor dicho valor se transforma en la dirección opuesta (del formato de comunicación al original).

- La parte servidora se denomina *skeleton*. Ésta representa la función que se ejecuta en el objeto una vez que se recibe una petición. Asimismo, se encarga de transforman de nuevo del formato de comunicación a la representación nativa (*demarshalling*) los argumentos. Si la operación devuelve algún valor éste se transforma al formato de comunicación (*marshalling*). El código de servicio de estas funciones (el que finalmente realiza la operación especificada en el interfaz) no se genera por el compilador IDL sino que está vacío. El desarrollo habitual del servidor suele realizarse, derivando una sub-clase del *skeleton* (denominada habitualmente *implementation*) que contenga el código de dichas funciones.

Tanto el *stub* como el *skeleton* se generan por medio del compilador IDL para un determinado lenguaje de programación y su código ha de ser enlazado para obtener los códigos finales de tanto cliente como servidor (objeto). La conversión de IDL a los distintos lenguajes de programación se encuentra recogida por OMG por medio de una serie de especificaciones (*IDL-Mappings*: Ada [OMG99a], C [OMG99b], C++ [OMG99c], COBOL [OMG99d], IDL a Java [OMG99n], Java a IDL [OMG99p] y Smalltalk [OMG99q]).

Todas las funciones de comunicación se encuentran encapsuladas dentro de estos elementos y su complejidad queda oculta desde la perspectiva del desarrollador, el cual sólo ha de especificar

el interfaz en IDL, compilar dicho interfaz e implementar las operaciones del interfaz para el caso del servidor.

2.2.2.6 Invocación Dinámica

Como alternativa al método estático de interacción entre cliente y servidor (*stub* y *skeleton*), CORBA proporciona un mecanismo dinámico de invocación de operaciones. Esta alternativa tiene como principal ventaja que:

- El cliente descubre de forma dinámica cuáles son las operaciones de un determinado interfaz, construyendo en tiempo de ejecución la peticiones. Para realizar esta búsqueda el cliente usa el Interfaz de Invocación Dinámico (DII o *Dynamic Invocation Interface*).

- El servidor procesa las peticiones de forma dinámica, analizando los argumentos de la petición y determinando cuál es el método al que ha de llamarse para resolverla. Esta opción permite publicar nuevas operaciones o alterar las ya existentes durante la ejecución del servidor. Esta gestión de peticiones se realiza por medio del Interfaz de *Skeleton* Dinámico (DSI o *Dynamic Skeleton Interface*).

La principal ventaja de este mecanismo es que la relación entre el interfaz (tal y como este ha de ser definido en IDL en el modo estático) y cliente/servidor no se fija en la fase compilación, sino que entre varias ejecuciones o incluso dentro de una misma ejecución, tanto cliente como servidor son capaces de usar/definir nuevos interfaces. Este mecanismo, mucho más flexible que el uso de interfaces enlazados en compilación en *stubs* y *skeletons* es, por el contrario, más complejo de utilizar y en la gran mayoría de implementaciones bastante menos eficiente y por lo general su aplicación está enfocada a situaciones en las cuales, por ejemplo, en el momento de diseñar el cliente se desconoce el interfaz del objeto o cuando un servidor debe de proporcionar una serie de operaciones que varían en el tiempo, el ejemplo más paradigmático al respecto es el desarrollo de puentes DCOM/CORBA.

2.2.3 CORBA: *Common Services*

Como ya se ha visto, los diferentes objetos (o interfaces) de la arquitectura OMA están divididos distintos grupos. Por su parte, los servicios representan los interfaces estándares para el desarrollo de aplicaciones en un entorno distribuido CORBA.

Las especificaciones de los servicios se recogen en diferentes grupos de documentos, denominados series COSS (*Common Object Services Specifications*) [OMG97r], ordenadas cronológicamente

(COSS1, COSS2, ...). Estas especificaciones son modulares, lo cual permite desarrollar aplicaciones que usen únicamente los servicios que requieran, aunque de todas maneras existen servicios que se requieren por otros servicios, aunque este tipo de dependencias no son muy frecuentes. Otra característica de éstas especificaciones es que en algunos servicios se definen diferentes conjuntos de funcionalidades ordenadas por medio de una graduación de forma que pueden desarrollarse implementaciones de los mismos que sólo cumplan hasta un determinado grado de funcionalidades. Todas las especificaciones de servicios están descritas en IDL.

2.2.3.1 Servicio de Nombres (*Naming Service*)

El Servicio de Nombres [OMG97h, OMG00i] proporciona las funcionalidades para asociar un nombre a un objeto, de acuerdo a un determinado contexto de nombrado (*naming context* [OMG98m]). Este nombre es único dentro de dicho contexto, pero puede repetirse en otros contextos. Dichos contextos pueden anidarse de forma jerárquica conformando una estructura similar a un árbol de directorios dentro de un sistema de ficheros.

Este servicio está definido para poder incluir dentro de él diferentes modelos de nombrado estándar existentes dentro de distintos dominios y plataformas. Otra característica de este servicio es que los servidores de nombres pueden encontrase *federados* [OMG98n], conformando una estructura distribuida sobre diferentes implementaciones para la gestión de nombres y contextos para un entorno complejo.

Este servicio es uno de los más importantes pues una vez localizado un servidor de nombres, se puede acceder al resto de servicios, consultando a dicho servidor. Para resolver este primer servidor, se necesita conocer la referencia del objeto de forma previa, esto es posible por medio de ciertos mecanismos de resolución de objetos que el propio ORB puede tener (`resolve_initial_references`) para estos servicios elementales.

2.2.3.2 Servicio de Eventos (*Event Service*)

El Servicio de Eventos [OMG97h, OMG00d] permite construir canales de eventos a los cuales pueden suscribirse tanto objetos que generen dichos eventos (*event suppliers* [OMG98l]) como receptores de los mismos (*event consumers* [OMG98l]). Por medio de este servicio es posible que dos objetos intercambien mensajes de forma asíncrona sin que tengan que conocer las referencias del otro objeto, de la misma forma se pueden tener varios productores y varios consumidores de eventos escuchando sobre el mismo canal.

La especificación de este servicio no requiere la existencia de ningún servidor centralizado que

reciba y reenvíe los eventos, permitiendo la escalabilidad del servicio. Las implementaciones que existen de este servicio utilizan operaciones de *mapping* sobre el repositorio de interfaces para gestionar los canales existentes, así como los productores y consumidores suscritos. El uso de este servicio permite establecer relaciones entre objetos que son capaces de intercambiarse mensajes, pero en las que ninguno de los mismos tiene porque conocer el objeto con el que se está comunicando.

El Servicio de Eventos proporciona dos modos de funcionamiento que pueden ser combinados para un mismo canal, estos modos son: (i) *push model* en el cual el productor de eventos decide cuando emite un mensaje a todos los consumidores y (ii) el modo *pull model* en el cual los consumidores solicitan en un momento determinado que los productores de eventos generen un mensaje. Además de los modelos de funcionamiento, el Servicio de Eventos proporciona otras funcionalidades adicionales, como son la especificación de políticas, el tipado de los canales de eventos, la negociación de calidades de servicio o la definición de mecanismos de filtrado de canales o de seguridad (aunque este último caso está aun en desarrollo).

2.2.3.3 Servicio de Objetos Persistentes (*Persistent Object Service*)

El Servicio de Objetos Persistentes [OMG97k, OMG00k] (habitualmente referido como POS) proporciona unos interfaces y mecanismos para poder almacenar el estado de un objeto de forma permanente, con la intención de reconstruirlo exactamente a lo largo de varias ejecuciones.

Una de las principales ventajas de este servicio es su flexibilidad, la cual permite que sea el propio objeto el que, en ultimo caso, defina cómo se ha de almacenar el estado y cómo éste se reconstruye, pudiendo delegar dichas operaciones en las funciones por omisión del servicio. El Servicio de Objetos Persistentes también se caracteriza por encontrarse abierto a diferentes tipos de implementaciones, que optimicen el funcionamiento de acuerdo a las características de los objetos y el entorno donde se apliquen.

2.2.3.4 Servicio de Ciclo de Vida (*Life Cycle Service*)

El Servicio de Ciclo de Vida [OMG97j, OMG00g] define un marco de funciones para crear, eliminar y mover objetos entre diferentes localizaciones. Este servicio se basa en el uso de Fábricas (el patrón *Factory* de Gamma [GHJV93]) para dar soporte a dichas funciones de gestión de objetos.

El Servicio de Ciclo de Vida define el interfaz de fábrica abstracta que se extiende por las diferentes fábricas para los distintos objetos. Estas fábricas concretas son otros objetos CORBA definidos por sus respectivos interfaces IDL, derivados de la citada fábrica abstracta.

Un escenario del funcionamiento de este servicio comienza cuando un cliente solicita a una fábrica la creación de un objeto de un tipo determinado y con unos argumentos concretos. Estas fábricas se localizan por medio de otros objetos denominados buscadores de fábricas (*Factory Finders*) que en base a una descripción del objeto a crear localizan las fábricas capaces de producir dicho objeto (estos buscadores pueden combinarse con otros mecanismos de localización de objetos, como pueden ser el Servicio de Nombres o el Servicio de Negociación). Un objeto creado de esta forma es posible destruirlo o moverlo/copiarlo a otra localización donde exista una fábrica que permita dicha migración.

Como funcionalidades extendidas, este servicio permite realizar las funciones anteriores sobre grupos de objetos interconectados por medio del Servicio de Relación, de forma que todos los objetos relacionados conforman un grafo que será, copiado, movido y/o destruido como si se tratase de un único objeto.

2.2.3.5 Servicio de Control de Concurrencia (*Concurrency Control Service*)

El Servicio de Control de Concurrencia [OMG97g, OMG00b] define mecanismos para gobernar el acceso por parte de varios clientes de forma concurrente a un mismo recurso. Estos accesos a recursos compartidos se coordinan de forma que sea posible resolver conflictos cuando dos clientes desean realizar algún tipo de operación sobre el recurso.

Los elementos que se usan en este servicio para realizar dichas funciones son los cierres (*locks*), que están asociados a los recursos, resultando necesarios para acceder a los mismos y que no pueden encontrarse en posesión de varios clientes de forma simultánea. El servicio proporciona diferentes modelos de cierres, con distintas semánticas para regular el acceso a los recursos en diferentes modalidades (lectura, escritura, ...) que a su vez pueden ser configurados para asignar tiempos máximos de bloqueo, cierres anidados, etc.

2.2.3.6 Servicio de Externalización (*Externalization Service*)

El Servicio de Externalización [OMG98k, OMG00e] proporciona los mecanismos necesarios para convertir un objeto en un flujo de datos que puede ser almacenado en un fichero, en memoria o transmitido por una red de comunicaciones. Asimismo, define las funciones necesarias para recuperar dicho formato de flujo de datos y reconstruir el objeto en el mismo u otro ORB.

Los mecanismos utilizados para transportar dicho flujo de datos están fuera del ámbito de responsabilidad del ORB y son por lo tanto independientes del servicio. El Servicio de Externalización

especifica un formato concreto para almacenar el flujo de datos en un fichero, por cuestiones de portabilidad.

Este servicio está relacionado con los servicios de Ciclo de Vida, para la reactivación de objetos y de Relación, para la gestión de grafos de objetos relacionados.

2.2.3.7 Servicio de Relación (*Relationship Service*)

El Servicio de Relación [OMG97n, OMG00n] define una serie de mecanismos para representar relaciones entre objetos, basándose en dos conceptos, por un lado el concepto de Rol (como entidad que participa en una relación) y por otro lado el concepto de Relación en sí. Por medio de este servicio se definen una serie de roles, que serán interpretados por diferentes objetos, y sobre dichos roles se definen las relaciones, las cuales especifican la cardinalidad, el orden y el tipo de los roles que las forman. Estas características de la relación se verifican por el propio servicio que asegura, por ejemplo, que las restricciones de cardinalidad no sean incumplidas, y si es así, levantará una excepción.

El Servicio de Relación juega un papel muy importante como complemento a los servicios de Ciclo de Vida o Externalización, por ejemplo, permitiendo el tratamiento como un conjunto de una serie de objetos interrelacionados. Las relaciones establecidas entre objetos, dibujan lo que se denominan grafos de relación. Estos grafos pueden ser navegados por medio de una serie de funcionalidades de este servicio, lo que implica no tener que implementar dichas funciones en cada uno de los objetos del grafo. Una de las ventajas de estas relaciones, a diferencia de la relación que se puede establecer cuándo un objeto intercambia con él peticiones, conociendo la referencia de otro, es que por medio de este servicio, la relación entre dos objetos puede ser alterada de forma externa sin que ninguno de los participantes de la relación intervenga en el cambio (de forma transparente a dichos objetos). Por ejemplo, si el objeto A solicita una petición al objeto B, con el que mantiene una relación definida mediante este servicio, un tercer objeto C, que conozca la existencia de dicha relación puede sustituir el objeto B, por el objeto D, sin que A intervenga (siempre cuando, B y D puedan asumir el mismo Rol en la relación).

2.2.3.8 Servicio de Transacción (*Transaction Service*)

El Servicio de Transacción [OMG97q, OMG00r] permite encapsular en una entidad la realización de una o varias operaciones sobre uno o varios objetos, de forma que se garantice que todas las operaciones agrupadas en una transacción, se realizan. Este servicio garantiza la atomicidad (si hay un fallo todas las operaciones de la transacción quedan anuladas), consistencia (el estado de los objetos que intervienen en una transacción no queda en ningún momento inconsistente,

después de ser realizada), aislamiento (los estados intermedios de una transacción no son visibles fuera de la transacción) y durabilidad (los efectos de una transacción completada con éxito son persistentes) de este tipo de operaciones múltiples.

El modelo básico de este servicio admite transacciones simples, y el modelo extendido, además soporta transacciones anidadas. Adicionalmente, el Servicio de Transacción permite dos modos de propagación: implícito (gestionado por el sistema) y explícito (controlado por la aplicación).

2.2.3.9 Servicio de Consulta (*Query Service*)

El Servicio de Consulta [OMG97m, OMG00m] permite a los usuarios y objetos la definición de consultas sobre colecciones de objetos. El término *consulta* dentro de este contexto no sólo se asocia a procesos de interrogación, sino también a operaciones de creación, inserción o borrado de elementos dentro de estas colecciones. Este servicio se basa en diferentes estándares de interrogación existentes, tales como SQL-92 [ANS93], OQL-93 o OQL-93 Basic [Cat94]. Este servicio ha sido definido con la intención de integrar aplicaciones tales como bases de datos (relacionales, orientadas a objetos o de cualquier otro tipo) dentro de un entorno distribuido CORBA.

La filosofía de este servicio permite que cualquier cliente pueda realizar una consulta sobre cualquier colección arbitraria de objetos, en base a una serie de predicados y en la cual se solicitan una serie de operaciones. Dichas operaciones pueden estar relacionadas con otros servicios, tales como el de Ciclo de Vida, el de Objetos Persistentes o el de Relación. Asimismo, el servicio permite la definición de mecanismos de mejora de rendimiento, tales como índices.

2.2.3.10 Servicio de Licencias (*Licensing Service*)

El Servicio de Licencias [OMG97i, OMG00f] permite a los fabricantes de software/hardware controlar el uso de sus productos por parte de los clientes, sujetos a una serie de términos de contrato o instalación. Este servicio va orientado hacia la aceptación de la tecnología CORBA por parte de los desarrolladores de productos software. Las políticas de control de definición de licencias especificadas por este servicio son muy flexibles y se basan en tres tipos de atributos:

- *Tiempo*: Usado para definir, entre otras, la fechas de comienzo, finalización y duración de licencias.
- *Mapas de Valores*: Usados para cuantificar el uso de la licencia, en base a valores *asignados* (por ejemplo, número de puestos disponibles) o valores *consumidos* (por ejemplo, número de usos).
- *Clientes*: Utilizados para asignar o reservar el uso de la licencia a un determinado cliente (por

ejemplo, una máquina concreta) que se haya registrado.

2.2.3.11 Servicio de Propiedades (*Property Service*)

El Servicio de Propiedad [OMG97l, OMG00l] permite la definición de atributos asociados a los diferentes objetos del entorno. Estos atributos son definidos, eliminados y manipulados dinámicamente, a diferencia de los atributos descritos por el interfaz IDL del objeto. Su uso se centra en la definición de información asociada a un objeto pero que no es parte de la información de dicho tipo de objeto. Las propiedades son pares nombre-valor o tuplas nombre-valor-modo, donde el valor puede ser cualquier tipo de datos. La utilización de las propiedades se puede manipular por medio de operaciones *batch* sobre conjuntos de propiedades u objetos y admite la definición de restricciones y control de acceso sobre las propiedades.

2.2.3.12 Servicio de Tiempo (*Time Service*)

El Servicio de Tiempo [OMG97o, OMG00p] permite definir una relación entre diferentes eventos que indique el orden de dichos acontecimientos y el cálculo del intervalo entre dichos eventos, así como la generación automática de eventos relacionados con el tiempo (alarmas o temporizadores). Para realizar estas funciones, este servicio se basa en la especificación de X/Open del Servicio de Tiempo en DCE [X/O94] y en el Protocolo de Tiempo de Red (RFC1119 [Mil89b]).

2.2.3.13 Servicio de Seguridad (*Security Service*)

El Servicio de Seguridad [OMG98o, OMG00o] proporciona las siguientes funcionalidades:

- *Identificación* y *autentificación* tanto a nivel de usuarios humanos como de objetos que participan en el entorno.
- *Autorización* y *control de acceso* en base a atributos del solicitante (privilegios, roles o grupos) y a cualidades de objeto (permisos).
- *Auditoría de seguridad* sobre las acciones de un usuario a lo largo de la secuencia de peticiones que realice. Esta función gestiona la delegación de privilegios.
- *Seguridad en la comunicación* entre dos objetos, proporcionando autenticación (del cliente frente al objeto y viceversa), integridad de los datos y opcionalmente confidencialidad.
- *No repudio* que garantiza que tanto cliente como objeto tienen prueba de que la otra parte ha enviado o recibido los datos.
- *Administración* de la información de seguridad, por ejemplo las políticas.

Estas funcionalidades del servicio se basan en otras funcionalidades internas, tales como el cifrado. Los mecanismos proporcionados por este servicio deben de estar soportados por todos los componentes del entorno, no sólo los objetos o clientes sino, el ORB y los elementos que lo componen, de esta forma, este servicio extiende el protocolo de interconexión entre ORBs con unas funciones de seguridad.

2.2.3.14 Servicio de Negociación de Objetos (*Object Trader Service*)

Como alternativa al Servicio de Nombres, el Servicio de Negociación [OMG97p, OMG00q] proporciona otro mecanismo para localizar objetos dentro del entorno. Este servicio utiliza la definición de una serie de atributos por parte de los objetos, estos atributos determinan la información relativa al tipo de servicio que proporcionan. Esta información, junto con la referencia del objeto es transmitida al Negociador o *Trader*. Éste es interrogado por los clientes en base a predicados de consulta sobre los atributos definidos por los objetos para localizar a los objetos. La operación de registro de un objeto se denomina *exportación* y la consulta de los objetos por parte de los clientes que cumple unas determinadas características se denomina *importación*.

Este servicio proporciona un mecanismo de localización de objetos mucho más flexible y potente que el Servicio de Nombres. Por medio de las consultas de los clientes se puede seleccionar un objeto que proporcione unas determinadas funcionalidades, restringiendo la búsqueda a implementaciones sobre un determinado sistema operativo u otros factores definidos por los atributos. Estos atributos con los que se registran los objetos en el servidor son definibles de forma dinámica y no están restringidos por la especificación.

Al igual que en el caso del Servicio de Nombres, y para facilitar la escalabilidad y administración de este servicio, se puede distribuir entre diferentes objetos *Trader* federados que cooperen para proporcionar el servicio en sistemas complejos.

2.2.3.15 Servicio de Colecciones de Objetos (*Object Services Service*)

El Servicio de Colecciones de Objetos [OMG97f, OMG00a] es más una utilidad que un servicio propiamente dicho. Por medio de las funciones que proporciona se pueden agrupar objetos en estructuras, tales como conjuntos, pilas, colas, listas y árboles. Dichas estructuras soportan operaciones de manipulación, consulta y navegación sobre los objetos que las componen. La definición de este servicio está orientada hacia la especificación unificada de las estructuras de datos más comúnmente utilizadas.

2.2.3.16 Servicio de Notificación (*Notification Service*)

El Servicio de Notificación [OMG00j] surgió como una extensión del Servicio de Eventos. Las nuevas funcionalidades proporcionadas por este servicio son:

- La posibilidad de transmitir datos en estructuras predefinidas, junto con eventos tipados y datos de tipo *Any*.
- Permite a los consumidores de eventos la definición de funciones de filtrado de eventos sobre los canales, para seleccionar la recepción de sólo un determinado tipo de eventos de los transmitidos por el canal.
- Permite a los productores de eventos, descubrir el tipo de eventos demandados por los receptores del canal, de forma que dichos eventos sean producidos bajo demanda, evitando la generación de mensajes que no son recepcionados por ningún objeto.
- Permite, de forma análoga, a los receptores del canal descubrir los tipos de eventos ofertados por los productores de forma dinámica, para su posterior suscripción.
- Proporciona mecanismos para la definición de calidades de servicio por canal, evento u objeto.
- La definición opcional de un repositorio de tipos de eventos, que los objetos puedan consultar para dirigir las suscripciones o accesos al canal de servicios.

2.3 Arquitectura DCOM de Microsoft

DCOM [HK97] es la extensión distribuida del modelo componentes COM [COM95] de Microsoft. DCOM (*Distributed Component Object Model*) es una tecnología integrada dentro de los sistemas operativos Windows NT y Windows 2000 de forma nativa. El principal factor a favor de esta tecnología se basa en el respaldo del fabricante líder en sistemas operativos de equipos de sobremesa Microsoft.

La tecnología COM permite la integración de componentes (denominados controles *Active/X*) dentro de aplicaciones. COM proporciona los mecanismos para integrar componentes dentro del mismo proceso o en diferentes procesos dentro de la misma computadora. DCOM extiende el modelo de interacción de COM por medio de mecanismos de conexión con componentes remotos. Estos mecanismos de conexión descansan sobre el entorno de comunicación DCE RPC [OSF95].

A diferencia de CORBA, DCOM no es una especificación general para guiar diferentes implementaciones de distintos fabricantes, en su lugar DCOM propone una tecnología definida desde su

diseño hasta su implementación y soporte. DCOM es un modelo menos ambicioso y más práctico para una arquitectura de objetos distribuidos.

2.3.1 Elementos de la Arquitectura DCOM

La Figura 2.3 muestra los elementos que componen la arquitectura de comunicaciones DCOM. Los elementos usados tanto por el cliente como por el objeto de servicio (denominado componente en la terminología COM) se denominan:

- **Proxy**: Elementos análogo al *stub* de CORBA, que representa el resguardo generado en base a la descripción IDL del interfaz del componente para el cliente.
- **Stub**: Elemento análogo al *skeleton* en CORBA. Este elemento es generado en base al fichero IDL que define el interfaz del componente. Es usado para activar el componente en la máquina servidora.

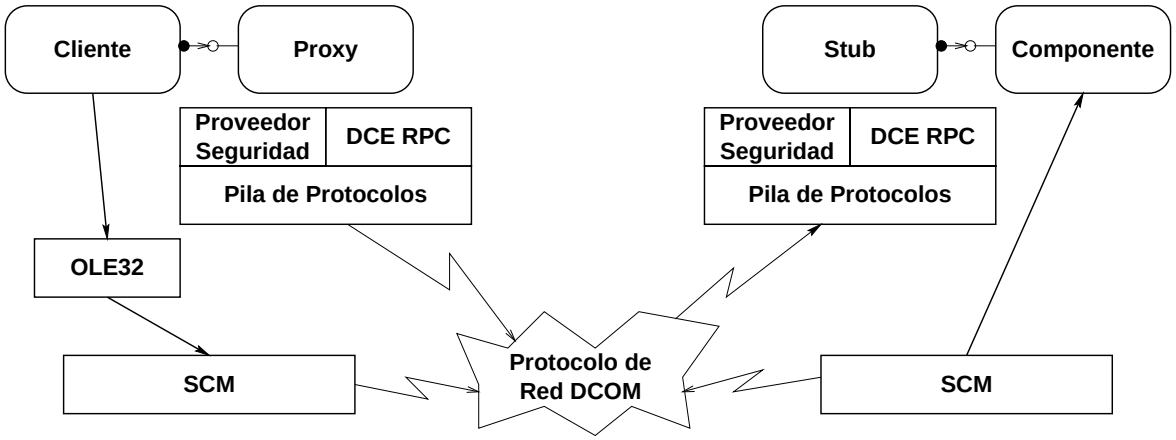


Figura 2.3: Elementos de la arquitectura DCOM

Estos componentes son generados desde la definición del interfaz del componente. Al igual que en CORBA la definición de los interfaces se realiza en IDL. La sintaxis de IDL usada por DCOM es sustancialmente diferente de la utilizada por CORBA, por ejemplo, DCOM-IDL no soporta herencia múltiple ni herencia de implementaciones, sin embargo DCOM permite que un mismo componente implemente varios interfaces mientras que CORBA no.

2.3.1.1 Niveles Inferiores de Comunicación

La pila de protocolos por debajo del interfaz DCE RPC substituye al mecanismo de llamada a procedimientos locales LPC (*Local Procedure Call*) usado por la arquitectura COM [Box97a, Box97b] cuando se interconectan dos componentes dentro de la misma computadora. El nivel

inferior de la arquitectura descansa en el SCM (*Shared Connection Manager*) que proporciona soporte a un mecanismo de recolección de basura (*garbage collection*) basado en un protocolo de *pinging*. Este protocolo manda mensajes periódicos desde el cliente hacia el servidor para indicar que dicho cliente mantiene una referencia al citado objeto. La pérdida de varios de estos mensajes (usualmente tres) de forma consecutiva implica la terminación anómala del cliente y por lo tanto la posible eliminación del componente si éste no es referenciado por otros clientes. Los mensajes de *ping* son agrupados para cada pareja de máquinas cliente/servidor (*delta pinging*), de forma que independientemente del número de referencias entre clientes y componentes entre dos computadores, sólo se manda un mensaje de *ping*. Estos mensajes pueden también ser adjuntados a mensajes intercambiados entre las dos computadoras, de forma que se reduce la intensidad de este tipo de tráfico en la red. Este mecanismo de detección de objetos no referenciados no encuentra especificación análoga en CORBA, que carece de técnicas de *garbage collection*.

2.3.1.2 Repositorios de Información

Al igual que CORBA, DCOM define dos repositorios donde se mantiene la información sobre los componentes del sistema. Estos elementos son necesarios para hacer posible la localización e interacción de los componentes con los clientes. Estos dos elementos son:

- **El Registro**: Usado para mantener información del software instalado en una computadora, es utilizado por DCOM para mantener la relación entre el identificador de clase (CLSID) y la ruta donde se encuentra el ejecutable asociado al componente. El CLSID se usa como identificador de cada objeto dentro del entorno DCOM.
  En DCOM tanto el *proxy* como el *stub*, es decir los resguardos de cliente y servidor respectivamente, son objetos DCOM. Por lo tanto, la librería dinámica (DLL) que los contiene también aparece registrada en este elemento.

- **La Biblioteca de Tipos**: Los interfaces de los métodos generados por el compilador IDL son almacenados en este elemento. A cada entrada de la biblioteca se asocia un *uuid*. Esta información es usada para la realización de los procesos de invocación dinámica de componentes, al igual que en CORBA.

2.3.1.3 Seguridad en DCOM

El soporte de seguridad en DCOM recae sobre los mecanismos de seguridad proporcionados por el entorno Windows NT. Los clientes, tanto usuarios como otras aplicaciones se agrupan en grupos a los que se asocian permisos. La información de cada usuario es almacenada en el directorio de usuarios, donde se encuentran el identificador, *password* y los privilegios. Las implementaciones de DCOM en otras plataformas no Windows (como es el caso de la versión para ciertas

plataformas Unix desarrollada por Software-AG) incluyen un *proveedor de seguridad* compatible con el esquema de seguridad de Windows NT.

2.3.2 Servicios de DCOM

El desarrollo de aplicaciones complejas requiere la disponibilidad de funcionalidades del entorno de objetos distribuidos que faciliten el desarrollo de las mismas. Estas funcionalidades se plasman en servicios estándar utilizables por varias aplicaciones. En DCOM existen los siguientes servicios:

- **Servicio de Transacciones** (MTS *Microsoft Transaction Service*)
  Que define el formato de los componentes asociados a operaciones atómicas de consulta–modificación de datos. Este servicio puede interactuar directamente contra SBGD, como SQL Server, Oracle e IBM DB2. MTS proporciona múltiples opciones de seguridad, eficiencia y facilidad de uso. Puede encontrarse más información sobre este servicio en [Mic98, Mic00c].

- **Servicio de Colas de Mensajes** (MSMQ *Microsoft Message Queue*)
  Proporciona comunicación entre componentes por medio de colas de mensajes asíncronos. Esta funcionalidad este proporcionada por un servidor centralizado donde son almacenados los mensajes enviados en una cola hasta que el componente receptor los retira. El servicio proporciona mecanismos de *log* de los mensajes retransmitidos, integridad y privacidad de los datos enviados y diferentes niveles de prioridad. Los documentos [Mic98, Mic00b] incluyen más información sobre este servicio.

- **Servidor de Información *Internet*** (IIS *Internet Information Server*)
  Este servicio está orientado a la difusión de información orientada al entorno *Web*. La tecnología de páginas ASP (*active Server Pages*) y VBScripts integrada con el servidor IIS permite el desarrollo de aplicaciones de *Internet* que interaccionen con una serie de componentes DCOM tanto en el servidor como en el cliente. Las referencias [Mic98, Mic99b] describen con más detalle este servicio.

- **Servicio de Directorio Activo** (ADS *Active Directory Service*)
  Este servicio se ha incorporado como parte de la arquitectura de red del entorno Windows 2000. Este servicio está orientado a la compartición y gestión eficiente de información relativa a los recursos, usuarios y componentes dentro de un entorno distribuido, manteniendo información de seguridad y características de estos elementos. ADS proporciona una organización jerárquica de servidores para dichas funcionalidades, permitiendo la escalabilidad del servicio. El objetivo del servicio ADS es homogeneizar e integrar la administración de un entorno distribuido de computadoras y recursos Windows. En [Mic99a, Mic00a] se puede en-

contrar más información sobre este servicio. El servicio ADS se encuentra desarrollado sobre el protocolo de servicio de directorio X.500 [WR92, WRH92].

2.4 Otras Aportaciones

Junto CORBA y DCOM, existen otras arquitecturas relevantes dentro de las herramientas de los entornos de desarrollo distribuido, por ejemplo, las tecnologías RMI, *Enterprise JavaBeans* y *Jini* de Sun o DCE de OSF.

2.4.1 Arquitecturas Distribuidas Java

La empresa Sun Microsystems en 1995 sacó al mercado el lenguaje y entorno de programación Java [GM95]. Una de las principales características de este lenguaje es la independencia de plataforma a la hora de ejecución. Esta característica representa una gran ventaja dentro de un marco de computadoras heterogéneas y para aprovechar dicha característica se han desarrollado diferentes entornos distribuidos centrados en la tecnología Java [Far98], tales como:

① *Remote Method Invocation* (RMI).

② *Enterprise JavaBeans*.

③ *Jini*.

La única desventaja clara de estas aportaciones es que suelen centrarse en sistemas completamente desarrollados en lenguaje Java, lo cual restringe la integración de código heredado (*legacy applications*) y de componentes desarrollados en otros lenguajes.

2.4.1.1 RMI

RMI [Sun98c] en sí no es un modelo de objetos distribuido, sino una técnica para comunicar objetos entre sí por medio de la invocación remota de métodos de forma transparente. RMI representa una versión mucho más flexible, pero similar a RPC [BN84]. La tecnología RMI es aplicable al desarrollo de pequeñas aplicaciones distribuidas, debido a su facilidad de manejo que facilita un rápido desarrollo. Sin embargo, RMI no proporciona ningún servicio extendido (salvo un esquema de localización y nombrado de objetos) que posibilite el desarrollo de grandes aplicaciones distribuidas.

Las últimas versiones RMI se encuentran implementadas sobre IIOP [Sun99c], el protocolo de interoperabilidad de diferentes ORBs CORBA.

2.4.1.2 Enterprise JavaBeans

La tecnología de componentes de Java, denominada *Enterprise JavaBeans* o EJB, proporciona un entorno de desarrollo completo para sistemas distribuidos. EJB se basa en tecnologías como el lenguaje Java y el mecanismo de invocación de métodos remotos RMI, añadiendo un modelo de componente flexible y ciertos servicios. El modelo de componente permite la definición de elementos que engloban ciertas operaciones sobre datos del sistema mediante un servicio sin estado. Estos componentes se instalan dentro de un contenedor o *container* que media en la interacción del cliente con el componente. Los componentes pueden ser distribuidos sin necesidad de proporcionar el código fuente y si sólo usan los servicios estándar de EJB pueden ser instalados en cualquier contenedor compatible.

Los servicios proporcionados por el entorno EJB son:

- JNDI (*Java Naming and Directory Interface*) [Sun99b]
  Proporciona un modelo homogéneo de definir esquemas de nombrado y organización de componentes. JNDI integra otros servicios de nombrado como DNS o DSML entre otros.

- EJB QL (*EJB Query Language*) [Sun99a]
  Permite localizar a componentes en base a una gramática reducida de SQL92. Una consulta EJB QL es interpretada por todos los contenedores y permite incluso consultar componentes cuyo estado este almacenado mediante mecanismos de persistencia de los contenedores. Este método de interrogación debe estar soportado por todos los contenedores y para todos los mecanismos de persistencia.

- JMS (*Java Messaging Service*) [Sun98b]
  Representa el mecanismo de mensajes asíncronos emitidos por los clientes o componentes y recogidos por los contenedores, los cuales se encargan de notificarlos al componente.

- JTS (*Java Transation Service*) [Sun99d]
  Proporciona el servicio de definición, control y gestión de operaciones transaccionales. JTS está relacionado con el Servicio de Objetos Persistentes de CORBA [OMG00r].

El modelo EJB se basa en la definición de un ciclo de vida en el desarrollo de componentes en el cual se identifican siete Roles: Desarrolador, Ensamblador, Implantador, Proveedor de Servicio, Proveedor de Contenedores, Proveedor de Gestores de Persistencia y Administrador. Estos roles pueden ser asignados a distintas empresas o grupos dentro del desarrollo de cada proyecto.

Las referencias [SG99, MH99] y las especificaciones EJB 1.1 [Sun98a] y EJB 2.0 [Sun99a] proporcionan información detallada sobre la tecnología *Enterprise JavaBeans*.

2.4.1.3 Jini

Jini [Wal99, Sun99e] es una tecnología para la gestión de recursos (software o hardware) y para facilitar su utilización a los usuarios. La tecnología *Jini* va dirigida hacia uno de los principios de la filosofía de Sun Microsystems: "*The network is the computer*". Bajo este lema, *Jini* proporciona a los usuarios la visión de la red como una única computadora en la cual los recursos son compartidos dinámicamente por diferentes usuarios del entorno.

Los elementos básicos de la filosofía *Jini* son los componentes que conforman lo que se denominan servicios federados. Dichos servicios son proporcionados al resto de componentes del sistema. El objetivo de esta tecnología se plasma en un modelo de programación más que en un conjunto de servicios bien definido. Mediante este modelo se promueve el desarrollo de componentes que cooperen con facilidad.

Jini se basa en otras tecnologías Java, como son RMI o el modelo de seguridad de la máquina virtual java (*sandboxing*). Los servicios elementales de *Jini* son similares a los usados por *Enterprise JavaBeans*, como JNDI, JMS o JTI, entre otros. Como servicio propio *Jini* presenta el Servicio de Búsqueda (*discovery/join service*), mediante el cual se descubren y registran los componentes y servicios *Jini*. *Jini* define un servicio propio para identificar, caídas o desconexiones de los componentes por medio de tiempos de *leasing*.

Jini es una tecnología centrada en Java, aunque de éste interesa principalmente las características de Java como entorno, como la portabilidad de código, carga dinámica y otras funcionalidades de los binarios Java. *Jini* admite cualquier lenguaje de programación si el resultado de la compilación fuese un *bytecode* Java, ejecutable en una máquina virtual java (JVM) estándar.

2.4.2 DCE de OSF

DCE [OSF94, OSF95] (*Distributed Computing Environment*) fue desarrollado por el consorcio OSF (*Open Software Foundation*) compuesto por IBM, DEC y Hewlett–Packard. DCE está compuesto por una serie de librerías y unos servicios estándar, orientadas al desarrollo de aplicaciones distribuidas. La arquitectura de funcionalidades de DCE se encuentra recogida en la Figura 2.4.

DCE utiliza como base de comunicaciones el mecanismo RPC [BN84] (*Remote Procedure Call*) para la invocación remota de procedimientos. La definición de estos procedimientos se describe en IDL. Junto con el substrato de comunicación, DCE proporciona un paquete de hilos de ejecución (*threads*) para los sistemas operativos que no soportan *threads* de forma nativa.

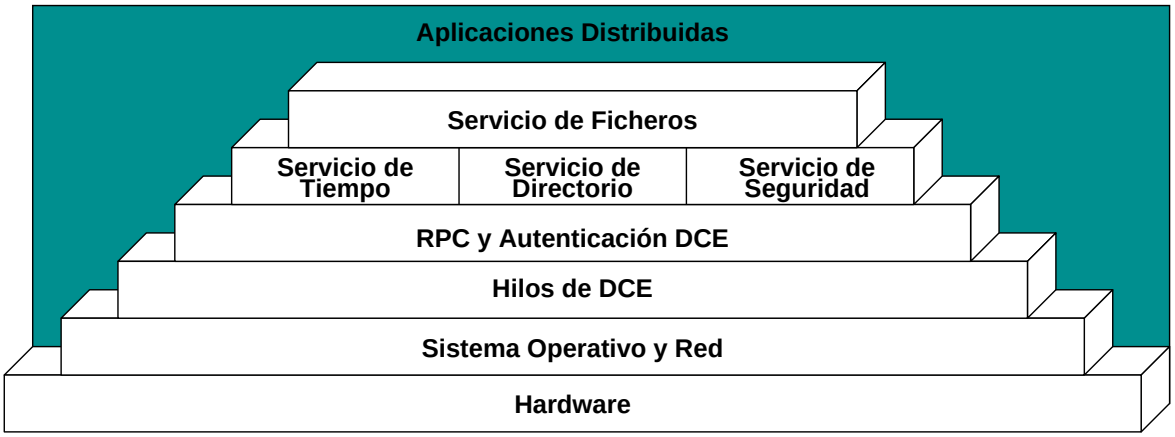


Figura 2.4: Elementos de la arquitectura DCE

Por encima del nivel de comunicación se encuentran definidos tres servicios básicos de DCE:

- **Servicio de Tiempo**: Este servicio se utiliza para sincronizar los relojes de las diferentes computadoras del entorno. Los eventos y demás sucesos del sistema pueden ser ordenados y secuenciados en base al instante de tiempo en el que ocurren, independientemente de la máquina en la que sucedan.

- **Servicio de Directorio**: Este servicio mantiene en una estructura todos los elementos del sistema (computadoras, servidores, recursos). El Servicio de Directorio en DCE se basa en el Servicio de Directorio Distribuido X.500 [WR92, WRH92] el cual es capaz de integrar diferentes esquemas de nombrado, tales como el Servicio de Nombres de Dominio DNS. Todos los elementos del sistema se registran en este servicio.

- **Servicio de Seguridad**: DCE implementa un servicio de autenticación basado en certificados y claves asimétricas para las peticiones de los clientes, así como listas de control de acceso para los recursos. El esquema de seguridad de DCE se basa en el sistema *Kerberos* [SNS88, Koh91].

Por encima de estos tres servicios básicos DCE define un **Servicio de Ficheros Distribuido** que permite que todos los elementos del entorno DCE tengan acceso común a los datos para los cuales tengan privilegios. Este servicio está compuesto por dos elementos para cada uno de las computadoras del sistema, un sistema de ficheros local, denominado *Episode* y unos mecanismos para compartir dicho sistema de ficheros local entre todas las computadoras de la red.

2.5 Conclusiones

Las tecnologías de componentes u objetos distribuidos representan en la actualidad las funcionalidades base para el desarrollo de sistemas distribuidos. Estas tecnologías garantizan funcionalidades de comunicación sobre entornos heterogéneos, transparencia y facilidad en la realización de peticiones remotas, ocultándolas como si se tratase de llamadas a métodos o procedimientos locales.

2.5.1 Comparativa de las Distintas Tecnologías

Dentro de las distintas tecnologías presentadas en este capítulo, se pueden identificar diferentes características propias de cada una de ellas. Estas características diferenciadoras son las que se han de analizar a la hora de seleccionar una u otra en el diseño e implementación de un sistema concreto.

2.5.1.1 CORBA

CORBA representa una tecnología de integración y consenso entre diferentes fabricantes. El arsenal de servicios y funcionalidades especificadas por los socios de OMG representa una de sus mayores ventajas. Sin embargo, al tratarse de un conjunto de especificaciones, su aplicación práctica en forma de herramientas e implementaciones que las soporten lleva un retraso en relación a las especificaciones. De todas formas existen multitud de implementaciones comerciales (Orbix, OAK, Visibroker) y de libre distribución (ORBit, MICO, OmniORB2, TAO) de ORBs que ya incorporan un representativo subconjunto de estos servicios y funcionalidades.

La falta de herramientas o entornos de desarrollo rápido sobra la tecnología CORBA es ahora una de las barreras. La carencia de un modelo de componente debidamente especificado en CORBA es otra de sus desventajas. Esta carencia se subsanará tras la definición de Modelo de Componentes CORBA [OMG98c] integrado en la siguiente versión de *CORBA Component* [OMG99i] como parte de la especificación de CORBA 2.3. Un modelo de componente soportado por todos los fabricantes de ORBs CORBA permitirá el desarrollo y distribución de componentes por parte de distintas compañías.

2.5.1.2 COM/DCOM

La baza más importante de la tecnología COM/DCOM se llama Microsoft. La multinacional norteamericana líder en el software de los equipos de sobremesa, distribuye el soporte de la tecnología DCOM integrado dentro de la familia de sistemas operativos Windows de la última gene-

ración (WinNT, Windows 2000). Su implantación en el mercado y el conjunto de herramientas de desarrollo rápido asociadas (por ejemplo, MS VisualStudio) permiten el desarrollo de aplicaciones DCOM potencialmente muy productivo. La otra cara de la hegemonía de Microsoft es que implica una dependencia de los desarrollos con un único fabricante. Esta situación liga a los usuarios con Microsoft, sin un mecanismo de migración aparentemente sencillo a entornos de otros fabricantes.

2.5.1.3 RMI/EJB/Jini

En base a la tecnología Java, Sun Microsystems ha desarrollado toda una familia de entornos de desarrollo y funcionalidades para sistemas distribuidos. RMI es la pieza más sencilla dentro de esta familia, en sí no es un entorno distribuido completo sino una extensión sencilla y cómoda del lenguaje Java para invocación de métodos remotos. EJB por su parte es una tecnología de componentes completa, con servicios claramente definidos y con un soporte de herramientas de desarrollo en continuo crecimiento. *Jini* por su parte es una idea más ambiciosa, la cual incluye no sólo los elementos tecnológicos de los anteriores (como invocación de métodos vía RMI o los servicios de EJB) sino servicios propios y una metodología de desarrollo con roles detalladamente definidos.

Estas tres tecnologías, sin embargo, centran su potencial en el entorno del lenguaje Java, lo cual restringe su aplicación a problemas que requieran la integración de código heredado (*legacy*) o en las que consideraciones de otro tipo (por ejemplo de eficiencia) requieran el desarrollo de componentes en otros lenguajes.

2.5.2 Limitaciones de la Tecnología de Componentes Distribuidos

El uso de modelos de referencia como los aquí vistos (especialmente CORBA) permite realizar diseños de sistemas distribuidos en los que se mantiene un cierto nivel de abstracción. Esta abstracción se centra en las características de más bajo nivel del desarrollo (lenguaje, sistema operativo, mecanismos de comunicación, ...) y permite, desde el punto de vista formal, definir soluciones a entornos distribuidos de una forma general y sin atarse a implementaciones o mecanismos de comunicación concretos, conservando la posibilidad de implantar soluciones aplicables.

No obstante, el uso de cualquiera de estas tecnologías de componentes distribuidos soluciona sólo en parte el problema derivado de la complejidad de los sistemas distribuidos actuales. Más concretamente, CORBA o DCOM proporcionan herramientas que a nivel tecnológico permiten solventar problemas de comunicación, concurrencia, movilidad, etc. Pero, aun así no dan ninguna directriz en relación a cómo organizar los componentes del sistema, es decir cómo dividir las tareas, cómo realizarlas; en resumen, no dan ninguna pauta de como definir el comportamiento de

los componentes del entorno. Esto se debe a que, evidentemente, estas consideraciones formales se encuentra fuera de su ámbito de aplicación y son problemas que deben ser resueltos por medio de otras aportaciones, por ejemplo la filosofía de *agentes*.

Capítulo 3

AGENTES SOFTWARE

Índice General

3.1 Introducción

En la actualidad el paradigma y, en general, toda la filosofía de agentes constituye uno de los campos de mayor desarrollo. Esto se debe a que su aparición en escena permite el desarrollo de aplicaciones que anteriormente era imposible desarrollar o por que bien su complejidad o la eficiencia de las posibles soluciones las hacia inabordables.

El empuje de agentes software se debe a las aportaciones de múltiples disciplinas que convergen en esta línea. Estas disciplinas provienen de campos tales como la Inteligencia Artificial (Inteligencia Artificial Distribuida, Sistemas Expertos, ...), Comunicaciones (Componentes Distribuidos y Componentes Móviles) o la Ingeniería del Software (Paradigmas de Programación, Interacción Hombre-Máquina, ...).

3.1.1 ¿Qué es un Agente?

Debido a las aportaciones procedentes de todas las áreas que han participado en esta nueva línea, se ha conformado una situación en la cual el término *agente*, pivote de toda esta nueva tecnología, no mantiene un significado igual para todos los investigadores del campo. El término original data de los trabajos de Hewitt [Hew76, Hew77], donde el concepto aparece frecuentemente referenciado también como *actor*. Sin embargo, este término ha evolucionado y se ha adaptado a las diferentes aportaciones que este nuevo campo proporciona a cada una de las distintas áreas de la computación que lo adoptan. Esto ha llevado a que cada investigador vea como un agente a componentes software de distinta naturaleza. Con la intención de arrojar un poco de luz sobre el concepto y con la intención de no ser en ningún momento categórico, a continuación se presentan tres enfoques diferentes de los agentes formulados por grupos de investigadores, considerados de los más relevantes en la materia. Estos enfoque son los de: Bradshaw, Jennings/Wooldridge y Nwana.

3.1.1.1 Bradshaw: Atribuciones y Descripciones

Bradshaw [Bra97a, Bra97b] aporta una definición al término agente basada en la combinación de dos enfoques diferentes:

- Por un lado una visión de los agentes basada en sus atribuciones, mediante la cual un agente software es aquel elemento del cual se espera que actúe en representación de otro elemento o persona con el objetivo de realizar las responsabilidades que en él se han delegado [ML94]; evidentemente un agente no puede realizarlo si desconoce el contexto de la solicitud.

- Por otro lado un agente puede ser definido de forma descriptiva, indicando que es una entidad software que opera, de forma continua y autónoma, en un entorno particular, habitualmente habitado por otros agentes [Soh97]. Esta segunda definición aporta al término agente el concepto de cooperación.

Además de estas características, diferentes investigadores han ido añadiendo nuevas características atribuibles a los agentes que conforman el abanico de atributos que se pueden asociar a dichos elementos, tales como: Reactividad, Movilidad, Adaptabilidad o Capacidad de Inferencia,

por nombrar algunas. Naturalmente, no todos los agentes cumplen todas ellas, sino que cada uno de los tipos de agentes incorpora, en mayor o menor medida, una o varias de estas características.

3.1.1.2 Jennings/Wooldridge: Características de un Agente

Jennings y Wooldridge [JW98a, JW98b] dan una definición preliminar de agente en la cual lo describen como '*el sistema que situado dentro de un entorno es capaz de actuar de forma autónoma para alcanzar los objetivos para los que se ha diseñado*'. Esta definición resalta como principal característica de los agentes el concepto de *autonomía*, sin embargo existen multitud de componentes actuales que se comportan de dicha forma y no pueden ser clasificados como agentes. Los autores ponen como ejemplos de agentes, para el caso de sistemas de control, desde un termostato hasta un sistema de monitorización de una central nuclear. Dentro del campo concreto de la computación, los *demonios* o servicios en UNIX o los componentes de control de los dispositivos de comunicación y conmutación (*routers*, puentes, ...). Sin embargo, no todos estos sistemas pueden ser considerados agentes, o por lo menos lo que Jennings y Wooldridge definen como *agentes inteligentes*, los cuales, además de esta característica cumplen las siguientes:

- *Reactivos*: Un agente debe ser capaz de percibir su entorno (que puede ser el mundo físico, otros agentes, ...) y responder a tiempo a los cambios producidos en este entorno.
- *Proactivos*: Un agente no sólo debe actuar en respuesta a los estímulos del entorno sino, que debe exhibir un cierto oportunismo, un comportamiento dirigido hacia su objetivo que le permita tomar la iniciativa cuando lo considere apropiado.
- *Sociales*: Los agentes deben ser capaces de interactuar con otros agentes o seres humanos, con el objeto de resolver su propio objetivo y colaborar en las actividades de los demás.

Además de estas características, un agente puede presentar otras cualidades, como movilidad, adaptabilidad o capacidad de aprendizaje; pero esas, en general no se consideran características fundamentales de un agente, sino funcionalidades particulares de los mismos para determinados entornos.

3.1.1.3 Nwana: Tipos de Agentes

Nwana [Nwa96, NN98] considera que el termino *agente* es un paraguas bajo el cual se cobijan todos los componentes software y/o hardware que presentan un comportamiento activo que les permite realizar las tareas que les han sido encomendadas. Partiendo de este punto, el autor muestra los diferentes tipos de elementos que él considera agentes. Esta clasificación representa una definición en extensión del termino general agente como la unión de todos los tipos de agentes. Según dicha clasificación, mostrada en la figura 3.1, los agentes se alinean en base a tres

atributos ideales, que son *autonomía, capacidad de aprendizaje* y *cooperación.*

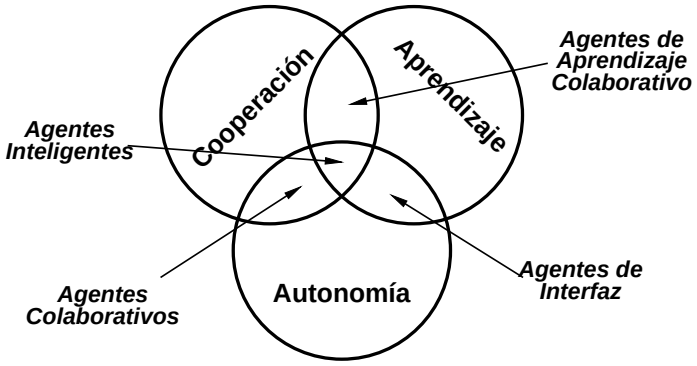


Figura 3.1: Tipología de agentes según Nwana

Autonomía representa la capacidad de actuar sin una guia directa del usuario, lo cual adicionalmente implica un cierto grado de proactividad para alcanzar los objetivos finales del agente. *Cooperación* implica la habilidad de los agentes para comunicarse entre sí por medio de un lenguaje común. Esta comunicación tiene como objetivo la colaboración para resolver problemas complejos. La *capacidad de aprendizaje* permite al agente aprender de su entorno con el objeto de proporcionar una mejor respuesta a la hora de interactuar con el mismo para alcanzar sus objetivos. Una consideración muy importante proporcionada por Nwana es que esta división no es estricta y que un gran número de los agentes presentan las tres cualidad, pero que en la mayoría de casos como máximo dos de ellas se encuentran mucho más marcadas que el resto. De la misma forma, Nwana afirma que hasta el momento no existen ningún agente que se pueda considerar un Agente Inteligente (*Smart Agent*).

Junto con los tipos de agentes extraídos de las combinaciones de estas características, aparecen otra serie de clases de agentes identificadas por otra serie de cualidades (movilidad, uso de información, . . .). En conjunto los agentes se clasifican en las siguientes divisiones:

- Agentes Colaborativos (*Collaborative Agents*): Este tipo de agentes ponen un mayor énfasis en los aspectos de autonomía y cooperación. Aunque, por supuesto, aprender no es su faceta principal. Estos agentes disponen de los mecanismos necesarios para *negociar* unas condiciones de cooperación satisfactorias que les permitan resolver en conjunto un problema general. Un ejemplo de este tipo de agentes son los que componen el sistema *Pleiades* [SZ96].

- Agentes de Interfaz (*Interface Agents*): En este caso las cualidades de aprendizaje y auto-

nomía caracterizan a estos agentes. El prototipo de agente de este tipo es el *asistente personal* que colabora directamente con el usuario en su propio entorno de trabajo; su aplicación más habitual es como guias en procesos de aprendizaje de un nuevo entorno (por ejemplo, una aplicación o un sistema operativo) o como herramientas que aprenden de las acciones del usuario con la intención de conseguir un mayor aprovechamiento del entorno.

- Agentes Móviles (*Mobile Agents*): Este grupo de agentes se podrían definir como las aplicaciones Software capaces de recorrer grandes redes (como Internet), interactuando con cada uno de los servidores, realizando una serie de tareas en nombre de su propietario y regresando al punto de origen. Este tipo de agentes tiene un factor muy importante de autonomía y disponen de otras cualidades cooperativas, pero sin duda su principal características es la capacidad que tienen de ejecutarse en diferentes ordenadores, moviéndose, incluso durante sus ejecución. La propuesta mas ambiciosa en este sentido es el lenguaje *TeleScript* [Whi96] aunque en la actualidad, y debido a su gran empuje en los últimos años, se usa el lenguaje *Java* [GM95].

- Agentes de Internet/Información (*Internet/Information Agents*): Este tipo de agentes se centran en gestionar, manipular y registrar fuentes de información que se encuentran distribuidas. Su principal aplicación ha sido en la exploración de la información contenida en los documentos WWW (*World Wide Web*). Dicha información se caracteriza por estar, en términos generales, muy desorganizada, contener numerosas referencias nulas o desactualizadas, pero, aun así se trata de una fuente de información increíblemente útil debido a la gran información que contiene. Los agentes desarrollados con la intención de obtener esta información se suelen denominar *softbots* (*software robot*). Estos agentes hacen uso de los protocolos y servicios de Internet, como `telnet`, `ftp` o `mail` como mecanismos de actuación, así como fuentes de información tales como `archie`, `gopher`, etc. Contrariamente de lo que pueda parecer, este tipo de agentes no suelen ser agentes móviles, más bien suele tratarse de sistemas integrados dentro de la herramienta de navegación (*browser*).

- Agentes Reactivos (*Reactive Agents*): Los agentes de esta clase se caracterizan por no contener ninguna representación simbólica del entorno. Esto implica que al no disponer de un modelo del mundo en el que se encuentran su funcionamiento está regulado por unos mecanismos estímulo-respuesta. A pesar de su simplicidad, Maes [Mae91b] recoge varios ejemplos en los cuales el comportamiento de dichos sistemas simples da lugar a unos patrones de comportamiento complejos.

- Agentes Híbridos (*Hybrid Agents*): Estrictamente, este elemento de clasificación no se corresponde con un tipo concreto de agente, sino con los casos en los cuales se define una combinación de las características de dos tipos de agentes. Una de las hibridaciones más

habitual (InteRRaP de Muller et al. [MP93, MPT95, FMP96]) es la combinación de los paradigmas reactivo y deliberativo. Fruto de esta unión es posible definir agentes en los cuales sólo parte de las acciones son analizadas por un modelo que planifica acciones a largo plazo (*goal-oriented*), siendo el resto de las acciones resueltas por medio de mecanismos reactivos (estímulo-respuesta).

Nwana [Nwa96] resalta que a la hora de construir sistemas multiagente, por lo general no se parte de un sólo tipo de estos agentes sino que se combinan las características de varios ellos en lo que se denominan **sistemas de agentes heterogéneos** (*heterogeneous agent systems*). En estos sistemas un elemento muy importante es la existencia de un lenguaje común de comunicación entre agentes. En el ámbito de estos sistemas, un agente es tal "*si y sólo si es capaz de comunicarse correctamente en un lenguaje de comunicación de agentes (ACL)*" Genesereth y Ketchpel [GK94].

3.1.2 Un Objeto no es un Agente

La revolución en el campo de desarrollo de aplicaciones fomentada por los agentes se asemeja a la sucedida con la aparición del paradigma de orientación a objetos. Este paradigma comparte con la filosofía de agentes el concepto de *abstracción*. Esta abstracción permite que un objeto oculte su estado interno (representado por sus atributos) dejando que este sólo sea accesible por medio del control de los métodos definidos. La diferencia es que los agentes encapsulan junto con su estado el *comportamiento*. Un objeto, a diferencia de un agente no tiene ningún control sobre le ejecución de los métodos. Un objeto depende de elementos externos (usuarios u otros objetos) que invoquen sus métodos. Desde este punto de vista un objeto como tal carece de una de las características que Jennings resalta de los agentes, la *autonomía*. Sin embargo, un agente dispone exactamente de esta capacidad, de forma que de un agente no se invoca un método sino que se realiza una petición que el agente realizará o no dependiendo de múltiples factores.

Franklin y Graesser en [FG96] intentan extraer cuales son las características esenciales de un agente y recalcan que una de las características fundamentales que distinguen los agentes de otras abstracciones software, como programas, es la *continuidad temporal*. Esta cualidad hace que un agente se encuentre activo de una forma continua a la espera de estímulos del exterior y como respuesta a ellos realice acciones. Un objeto o un programa tiene que ser invocado por una entidad activa (un ser humano, otro programa/objeto), no por una situación puntual del entorno.

3.2 Facetas del Diseño de un Agente

Tras una perspectiva general de cuales son las posibles definiciones del término agente, el siguiente punto a desarrollar son las diferentes facetas que intervienen en el proceso de diseño de un agente. Considerando como facetas, los diferentes niveles de análisis en dicho proceso de desarrollo, recorriendo el espectro que va desde las herramientas formales de especificación del comportamiento del agente hasta posibles opciones a la hora de elegir el lenguaje en el que dicho agente ha de ser programado. Se va a explorar el desarrollo de un agente como elemento aislado, dejando para más adelante su interacción con otros agentes. Esto implica que, si bien algunas de las aproximaciones que se detallan a lo largo de esta sección preven la integración del agente dentro de un entorno habitado por otros agentes con los que ha de comunicarse y colaborar, la línea general de la sección intentará centrarse, en la medida de lo posible, en un agente y su entorno más inmediato.

Según el análisis realizado por Wooldridge y Jennings en [WJ95] es posible identificar tres facetas a considerar a la hora abordar el diseño de un agente software:

① **Teorías de Agentes**: Generalmente expresadas como especificaciones que aportan respuestas a cómo ha de ser conceptualizado un agente, o cuáles son las propiedades que debe tener un agente. Un elemento muy importante de estas aproximaciones formales al diseño de un agente es cómo se van a expresar formalmente y como se van a manejar las propiedades para definir las pautas de comportamiento del agente.

② **Arquitecturas de Agentes**: Dirigidas a progresar desde la especificación hacia la implementación del agente. Las consideraciones planteadas a este nivel atajan los problemas derivados de construir sistemas y aplicaciones que satisfagan dichas propiedades antes citadas. Las soluciones encontradas indican cuáles han de ser las estructuras (software o hardware) adecuadas para el desarrollo apropiado del agente.

③ **Lenguajes de Agentes**: El último paso en el desarrollo del agente es la elección del lenguaje de programación en el que será implementado. Esta decisión debe encontrarse justificada por ciertas características del lenguaje que lo hagan especialmente idóneo para el desarrollo del agente, tal y como ha sido concebido en las fases anteriores. Dentro de las opciones disponibles se encuentran tanto los lenguajes de programación de propósito general como los especialmente diseñados para el desarrollo de agentes con unas determinadas características, por ejemplo la movilidad.

3.2.1 Teorías de Agentes

Las teorías de agentes son una serie de formalismos que ayudan a definir una especificación de lo que se desea que haga el agente. Esta especificación contiene ciertas propiedades necesarias para poder modelizar el comportamiento del agente en base a unos modelos determinados. Es decir, que si se desea que el agente sea capaz de recordar cosas, debemos dotarle de memoria, así como de una representación determinada de los hechos y estados pasados que sea útil para hacer uso de dicha memoria de una forma provechosa.

Un agente es una entidad autónoma que debe tomar decisiones dentro de un entorno del cual tiene una relativa información. Esto implica que el modelo de funcionamiento de un agente debe representar tanto la información parcial del entorno como los planes de acciones autónomas del agente. En el campo de desarrollo de los agentes, se ha optado por utilizar esta aproximación para representar el comportamiento de sistemas complejos y que trabajan con información incompleta. En concreto, para los agentes se han estudiado [WJ95] las actitudes posibles como formas de describir su comportamiento, agrupándolas en dos clases: *actitudes de información* y *pro-actitudes*.

$$\text{Actitudes de Información} \begin{cases} \text{Creencias} \\ \text{Conocimientos} \end{cases} \qquad \text{Pro-Actitudes} \begin{cases} \text{Deseos} \\ \text{Intenciones} \\ \text{Obligaciones} \\ \text{Opciones} \\ \dots \end{cases}$$

Las *actitudes de información* están relacionadas con la información del entorno que conoce el agente, mientras que las *pro-actitudes* guía las acciones a realizar por el agente. A priori, un agente puede ser definido usando como mínimo una actitud de cada uno de los grupos, aunque habitualmente se usan varias. En conjunto todas estas actitudes constituyen el estado del agente.

Una teoría de agentes debe describir de forma concreta una serie de aspectos del comportamiento del agente, como son:

① Cómo afecta la información disponible por el agente a las pro-actitudes que planifica.

② Cómo cambia el estado del agente a lo largo del tiempo.

③ Cómo afecta el entorno al estado del componente.

④ Cómo conducen la información y las pro-actitudes a que el agente realice acciones.

3.2.1.1 Teoría de Conocimiento y Acción

Esta aproximación al problema de la modelización de agentes fue propuesta por Moore [Moo90]. Ésta plantea como hipótesis de partida en concepto de *pre-condiciones de conocimiento para ac-*

ciones, es decir qué necesita saber un agente para poder realizar una acción. Para ello proporciona un modelo para presentar las *habilidades* por medio de la lógica que contiene una modelización del conocimiento contenido en el agente y un aparato formal, basado en la lógica para modelizar las acciones. Este formalismo, permite que un agente sea capaz de tener información incompleta de como alcanzar un objetivo y de todas formas realizar acciones con el objeto de encontrar como llegar a dicha meta.

3.2.1.2 Teoría de Intención

Una teoría que representa un punto de referencia en la teoría de agentes actuales es la formulada por Cohen y Levesque [CL90a]. Este formalismo fue usado para desarrollar la teoría de la intención, que es considerada por los autores como un prerrequisito para la teoría de actos del habla [CL90b]. Posteriormente se comprobó que la lógica formalizada con esa intención resultaba útil para expresar modelos de razonamiento de los agentes. Unos de sus mayores campos de aplicación es en la modelización de sistemas multi-agente, como herramienta para resolver conflictos de cooperación y diálogo.

3.2.1.3 Teoría de Creencia, Deseo, Intención

El modelo de Creencia, Deseo, Intención (*Belief, Desire, Intention* o BDI) formulado por Rao y Georgeff [RG91a, RG91b, RG92a] selecciona estas tres actitudes de entre las clasificadas como actitudes de información y pro-actitudes como primitivas modales. Esta elección la diferencia del modelo de Cohen y Levesque que, junto con creencias y metas, define posteriormente las intenciones en función de estas dos. Este formalismo está basado en una exploración de las intenciones por medio de ramas construidas a lo largo del tiempo (*time-branching model*). Este modelo actualmente se está usando para enfocar la representación de cómo las creencias de un agente en relación al futuro afectan a sus deseos y creencias. Esto permite añadir planes sociales al formalismo [RG92b]. En [GPP+99] se recoge un análisis detallado de la situación actual de este modelo de cara a plantear soluciones a sus carencias. Probablemente sea uno de los formalismos más comúnmente aceptado, encontrándose asociado a las arquitecturas IRMA (*Intelligent Resource-Bounded Machine* [BIP88]) y PRS (*Procedural Reasoning System* [GL87]).

3.2.1.4 Teoría de Singh

Una aproximación ligeramente diferente es aportada por Singh [Sin94]. El trabajo de Singh se plasma en el desarrollo de una familia de definiciones lógicas para representar intenciones, creencias, conocimientos, comunicación, etc. El formalismo en conjunto es extremadamente rico, pero relativamente complejo. Existen diferentes autores trabajando en el establecimiento de propieda-

des para este formalismo, así como de se aplicación práctica.

3.2.1.5 Teoría de Werner

Werner [Wer88, Wer89, Wer90, Wer91] define los fundamentos de un modelo general de agentes, tomando conceptos de trabajos en los campos de la economía, teoría de juegos, teoría de situación de autómatas, semánticas de situación y filosofía. Actualmente, las capacidades aplicadas de este modelo no han sido desarrolladas lo suficiente como para argumentar por medio de resultados sus ventajas y/o problemas.

3.2.1.6 Teoría de Modelización de Sistemas Multi-Agente

Como tema de investigación de su tesis doctoral, Wooldridge [Woo92a, Woo92b], desarrolló una serie de modelos lógicos para representar las propiedades de un sistema multiagente. Se diferencia, sin embargo de las teorías vistas anteriormente en no intentar proporcionar un marco general para los modelos de agentes. En su lugar, Wooldridge busca la definición de unos formalismos que permitan especificar y verificar sistemas multiagente. El modelo expresado en la tesis representa de una forma hasta cierto punto general el comportamiento de estos sistemas y muestra cómo la traza de la ejecución de los mismos puede ser usada para modelizar las creencias. Wooldridge también aplica esta teoría a la especificación y verificación de protocolos de cooperación entre agentes.

3.2.2 Arquitecturas de Agentes

Una vez asentados los formalismos que representan la especificación del comportamiento del agente, el siguiente paso es aproximar esta especificación del campo teórico a la práctica.

Según Maes [Mae91a] una arquitectura de agentes se define como una metodología particular para la construcción de agentes. La autora recalca que esta arquitectura debe especificar cómo se debe descomponer la construcción del agente en una serie de componentes, cómo se deben hacer los módulos y cómo han de comunicarse entre sí. De esta forma, en conjunto todos los módulos deben interactuar para proporcionar una respuesta a la situación detectada por los sensores, encontrándose el agente en un determinado estado interno, planteando esta respuesta en términos de órdenes sobre los actuadores, así como en un nuevo estado del agente. Finalmente Maes resalta que una arquitectura comprende técnicas y algoritmos que den soporte a dichas metodologías.

Existen en la literatura de agentes numerosas propuestas arquitectónicas a la hora de construir un agente. Estas investigaciones pueden clasificarse básicamente en tres grupos: **arquitecturas**

deliberativas usadas por agentes que representan un modelo interno del entorno donde se desenvuelven, **arquitecturas reactivas** que implementan agentes guiados por reglas del tipo estímulo-respuesta y **arquitecturas híbridas** que combinan ambos enfoques.

3.2.2.1 Arquitecturas Deliberativas

Existe un punto de partida común para varios modelos de arquitecturas dentro del campo de la inteligencia artificial, que data de los trabajos de Newell y Simon [NS76]. Este punto de referencia se basa en el paradigma de la hipótesis del sistema físico-simbólico (*phisical-symbol system hypothesis*). Una arquitectura de este tipo está compuesta por entidades físicas (símbolos) organizados en estructuras que puede operar y procesar dicho sistema. Bajo esta hipótesis, un sistema de este tipo es capaz de realizar acciones inteligentes. Partiendo de esta idea original, surge el concepto de los agentes deliberativos como aquellos que representar mediante unos modelos simbólicos el entorno donde operar, de forma que las decisiones (por ejemplo, las acciones a realizar) son tomadas por medio de razonamiento lógico (comparación de patrones o manipulación de símbolos).

Dentro de este enfoque, la primera arquitectura que se puede encuadrar en él es STRIPS [FN71]. Esta arquitectura parte de una representación simbólica del mundo y de los objetivos y usa razonamiento basado en análisis medios–fines para construir planes comparando pre- y postcondiciones asociadas a varias acciones. El algoritmo de STRIPS se demostró que no era válido para problemas por encima de una complejidad moderada, aportándose mejoras basadas en planificación jerárquica y no lineal. A pesar de sus limitaciones, variaciones de esta arquitectura se han aplicado en entornos de relativa complejidad: simulación de tráfico (AUTODRIVE [Woo93]), sistemas operativos UNIX (Softbots [ELS94]) o simulación de incendios forestales (PHEONIX [CGHH89]).

Basados en los modelos BDI (*belief, desires, intentions*) de Rao y Georgeff vistos anteriormente, Bratman desarrolló la arquitectura genérica IRMA (*Intelligent Resource–bounded Machine Architecture*) [BIP88]. Esta arquitectura define cuatro conjuntos de elementos simbólicos: una librería de planes, y las representaciones de creencias, deseos e intenciones. Como elemento principal de la arquitectura se encuentra un motor de razonamiento, así como un analizador de oportunidades (que monitoriza el entorno para determinar las futuras acciones del agente), un proceso de filtrado (que reduce el conjunto de acciones que agente puede realizar en base a unas situaciones determinadas) y un proceso de deliberación (encargado de seleccionar entre las diferentes alternativas de acción). La arquitectura IRMA ha sido evaluada con éxito en un escenario experimental denominado *Tileworld* [PR90].

Vere y Bickmore [VB90] desarrollaron un prototipo de agente capaz de ser controlada por me-

dio de ordenes expresadas en base a un reducido conjunto de palabras (800 términos) y capaz de establecer y realizar planes para alcanzar los objetivos formulados en dichas sentencias. HOMER, simulaba un robot submarino sobre un mundo submarino bidimensional, denominado *Seaworld* del cual el robot posee un conocimiento parcial. El tipo de acciones que HOMER era capaz de realizar eran, típicamente, buscar y recoger objetos.

La arquitectura GRATE* propuesta por Jennings [Jen93] divide en dos diferentes niveles las funciones del agente, por debajo se encuentra un sistema dependiente del dominio (control industrial, finanzas o transporte) y por encima un nivel de coordinación entre el agente y los agentes de los otros dominios. Este nivel se encuentra, a su vez compuesto por un interfaz con el nivel inferior así como por un modulo de cooperación y un módulo de valoración de situaciones.

3.2.2.2 Agentes Reactivos

La idea de agentes deliberativos basados por completo en mecanismos lógicos de razonamiento es increíblemente tentadora, pero la experiencia ha demostrado que existen grandes problemas a la hora de representar un universo moderadamente complejo por medio de modelos simbólicos, así como de definir algoritmos de razonamiento eficaces para tales modelos [Kae86, RW91]. Además de este tipo de arquitecturas, y sobre todo para determinados campos de aplicación de los agentes, se utiliza como alternativa, lo que se denominan modelos reactivos, que funcionan siguiendo un paradigma del tipo *estímulo-respuesta*.

Este tipo de agentes, a diferencia de los anteriores no disponen de un modelo del mundo sobre el cual construir sus planes sino que disponen de una serie de patrones activables bajo ciertas condiciones de los sensores (estímulos) y que activan de forma directa los actuadores (respuesta). Los modelos de funcionamiento de este tipo de agentes van desde redes (activadas en base a condiciones de los sensores), autómatas de estados (habitualmente organizados en varios niveles), coste asociado a tareas (seleccionando entre varias opciones la de mayor/menor coste) o redes neuronales.

Brooks [Bro86] fue el primero en criticar el paradigma simbólico como modelo de construcción de agentes. Brooks afirma que un comportamiento inteligente puede realizarse sin una representación simbólica como las antes citadas, así como sin necesidad de un razonamiento abstracto y que en realidad la inteligencia aparece como consecuencia lógica de ciertos sistemas complejos. Como alternativa a las arquitecturas deliberativas, el autor propone la arquitectura de subordinación (*subsumption architecture*). Esta arquitectura está compuesta por una jerarquía de comportamientos enfocados hacia la resolución de ciertas tareas que compiten por el control de las acciones

del robot. Los niveles superiores determinan las acciones más generales del comportamiento del robot, mientras que los niveles inferiores (con mayor precedencia a la hora de la evaluación) representan los comportamientos mas básicos, por ejemplo evitar la colisión con obstáculos. Este tipo de arquitecturas han proporcionado muy buenos resultados: *Mars Explorer* [Ste90].

Agre y Chapman [CA86, AC87] plantearon que la gran mayoría de las actividades diarias, hasta cierto punto se realizan de forma rutinaria, en el sentido que requieren poco o ningún razonamiento abstracto. Partiendo de esta idea, Arge y Chapman proponen que la gran mayoría de tareas, una vez aprendidas se pueden aplicar de forma rutinaria con mínimas variaciones. La aplicación de estas afirmaciones se plasma en el desarrollo del sistema PENGI que se basa en un circuito lógico capaz de determinar las acciones de forma muy rápida que es revisado de forma periódica para adaptarlo a nuevos problemas o situaciones.

Kaelbling y Rosenchein [KR91, Kae91] por su parte plantean una solución similar. Partiendo de una especificación en términos declarativos del agente, mediante un proceso de compilación obtienen una máquina digital capaz de aplicar el razonamiento en un rango limitado de tiempo. Esta aproximación toma de las arquitecturas declarativas los modelos de razonamiento simbólico (y por lo tanto ciertas de sus limitaciones), pero resuelve el problema de la eficiencia, puesto que durante la ejecución del agente no existe ningún razonamiento simbólico y únicamente la aplicación de la máquina digital a las entradas (sensores) del agente.

Pattie Maes aplica dentro de la arquitectura de red de agentes (*agent network architecture*) [Mae91a] las ideas de varios predecesores. La arquitectura propuesta está dividida en módulos activables de forma independiente (similar a la arquitectura de Brooks). Estos módulos están definidos en base a términos relacionados con pre- y postcondiciones (como los operadores de STRIPS), así como asociados a un nivel de activación (asociado a la importancia del módulo en determinadas situaciones, cuanto mayor sea dicho valor, más probable es que dicho módulo afecte al comportamiento del agente). Estos módulos se encuentran enlazados en base a sus pre- y postcondiciones, de forma similar a los enlaces establecidos en una red neuronal.

3.2.2.3 Arquitecturas Híbridas

Para varios autores ni el paradigma simbólico de las arquitecturas deliberativas ni las alternativas reactivas proporcionan soluciones enteramente adecuadas para la construcción de agentes. Como punto de convergencia de ambos enfoques surgen las *arquitecturas híbridas*. La perspectiva general de estas soluciones parte del desarrollo de dos módulos, uno deliberativo y otro reactivo que colaboran en el control del agente. Por lo general, el módulo reactivo se le da prioridad con el

fin de resolver problemas de forma rápida en situaciones en las cuales se requiere una reacción en un tiempo limitado.

El sistema PRS (*Procedural Reasoning System* [GL87]) de Georgeff y Lansky es un ejemplo de este tipo de arquitecturas. Al igual que IRMA se basan en modelos de agentes BDI (*belief–desire–intention*), pero se diferencia de este último principalmente en que la librería de planes contiene planes parcialmente elaborados llamados áreas de conocimiento (KA o *knowledge areas*) asociadas con condiciones de invocación. Estas KAs pueden ser activadas por las metas (*goal-driven KA*) o por los datos (*data-driven KA*) o presentar un funcionamiento reactivo, consiguiendo de esta forma una reacción rápida ante cambios en el entorno. El conjunto de KAs activas en un momento determinado representa las intenciones del agente.

Ferguson en su tesis doctoral [Fer92] desarrolló la arquitectura TOURINGMACHINES. La arquitectura está compuesta por dos subsistemas de percepción y actuación interconectados por medio de tres niveles de control. Estos niveles son: (i) el nivel reactivo que está implementado en base a reglas del tipo de situación–acción y que está pensado para proporcionar una respuesta rápida a eventos (similar a la propuesta de Brooks), (ii) un nivel de planificación que usa una librería de planes parciales y una representación del mundo para construir planes que permitan alcanzar los objetivos del agente; este nivel dispone de un mecanismo para enfocar la atención del nivel con la intención de reducir el volumen de información que ha de combinar para construir el plan y (iii) un nivel de modelización que contiene una representación simbólica del estado de otras entidades del entorno del agente; estos modelos se usan para identificar y resolver los conflictos de objetivos, de forma que el agente sea capaz de identificar las metas que no son alcanzables. El trabajo sobre esta arquitectura se encuentra recogido con mayor detalle en [Fer95a, Fer95b].

Burmeister *et al.* [BS92, Had94] definieron la arquitectura COSY partiendo de las ideas extraídas de IRMA y PRS. COSY es una arquitectura BDI desarrollada sobre un entorno de desarrollo y prueba de sistemas multiagente denominado DASEDIS. Esta arquitectura se compone de cinco elementos: sensores, actuadores, comunicadores, intención y cognición. COSY guarda varias similitudes con las dos arquitecturas en las que se apoya, pero presenta características innovadoras en lo referente a la cooperación entre agentes de la misma naturaleza por medio de protocolos, definidos como estereotipos de diálogos dentro del entorno cooperativo de los agentes.

INTERRAP [MP93, MPT95, FMP96, Mül96] es otra arquitectura organizada en niveles (como TOURINGMACHINES de Ferguson) en la cual cada nivel representa un grado mayor de abstracción a la hora de determinar las acciones del agente. La principal diferencia es que en INTERRAP la

arquitectura del agente también se encuentra dividida de forma vertical, de forma que cada nivel dispone de una base de conocimiento por un lado, y de unos componentes de control que usan dicha base de conocimiento. El nivel más bajo se denomina interfaz con el mundo (*world interface layer*) cuya base de conocimiento mantiene patrones de comportamiento o PoBs (*Patterns of Behaviour*) que definen el comportamiento reactivo del agente. Por encima de este nivel se encuentra el nivel de planificación encargado de confeccionar planes para alcanzar los objetivos del agente. Y por último se encuentra el nivel de cooperación que construye planes de consenso entre los objetivos de varios agentes.

3.2.3 Lenguajes de Agentes

Si bien un agente puede ser implementado en cualquier lenguaje de programación, existen ciertas opciones especialmente adecuadas para esta tarea. De forma análoga que el desarrollo de un compilador puede hacerse en ensamblador, el uso de lenguajes de más alto nivel como C o Pascal es más adecuado a la complejidad de dicho problema; en el caso del desarrollo de agentes existen a su vez ciertos lenguajes especialmente adecuados para abordar dicho problema. Las cualidades de un lenguaje que le hacen más adecuado que otro para el desarrollo de un agente dependen de las características deseadas para el agente que se encuentren respaldadas por funcionalidades y capacidades integradas y propias del lenguaje a usar.

En sí, el concepto de lenguaje propiamente diseñado para agentes se podría definir como el conjunto de herramientas que permiten desarrollar un sistema (hardware o software) en base a los conceptos teóricos de la teoría de agentes.

3.2.3.1 Programación Orientada a Agentes

La programación orientada a agentes (AOP o *Agent Oriented Programming*), definida por Shoham [Sho90, Sho93, Soh97], utiliza los mismos términos usados para definir los agentes, tales como creencias, intenciones etc. como vehículo para programar también dichos agentes, mediante una sintaxis determinada.

Shoham propone que para desarrollar completamente AOP es necesario resolver tres puntos:

- Un sistema lógico para definir el estado mental de los agentes.
- Un lenguaje de programación interpretado para programar los agentes.
- Un proceso de '*agentificación*' (*agentification*) mediante el cual compilar los programas y conseguir ejecutables sobre arquitecturas determinadas.

Sobre estas premisas, Shoham desarrolló el primer lenguaje orientado a agentes, denominado *Agent0*. Un ejemplo de una sentencia de este lenguaje es el presentado en *AGENT0–1* [Sho97], este ejemplo refleja la expresión: *'Si te llega una solicitud para liberar la impresora en los próximos 5 minutos y no te comprometes a terminar el trabajo actual en los próximos 10 minutos y si crees que el solicitante es amistoso, entonces acepta la petición y dile que lo hiciste'*.

EJEMPLO AGENT0–1:

```
IF
 MSG COND:     (?msgId ?someone REQUEST
               ((FREE-PRINTER 5min) ?time))
 MENTAL        ((NOT (CMT ?other (PRINT ?doc (?time+10min)))))
 COND:
               (B (FRIENDLY ?someone))
 THEN          ((?someone (FREE-PRINTER 5min) ?time))
 COMMIT
               (myself (INFORM ?someone (ACCEPT ?msgId)) now)
```

En relación a la lógica de los programas en *Agent0*, hay que resaltar que los elementos manejados son las creencias que el agente tiene en relación a la información externa y los compromisos (*commitments*) que tiene contraídos. Estos se expresan por medio de reglas de compromisos como la antes vista en la cual se indican condiciones de mensaje (`MSG COND`), condiciones del estado del agente (`MENTAL COND`) y una acción. Estas condiciones son comparadas con los mensajes recibidos por el agente y por las creencias y compromisos que el agente tiene, respectivamente. Por otro lado, las acciones puede representar variaciones en el estado del agente (acciones privadas) o el envío de mensajes a otros agentes (acciones de comunicación), estas últimas se basan la teoría de actos del habla (*speech acts*) [Aus62, Sea69], usando los tipos de mensaje `request`, `unrequest` e `inform`.

Como extensión al prototipo *Agent0* desarrollado por Shoham, en [Tho93] Thomas desarrolla PLACA (*Planning Communication Agents*). La autora plantea una solución a la principal desventaja de *Agent0* que es que los agentes no puede expresar la construcción de planes que dirijan sus acciones hacia un objetivo final. La programación en PLACA es bastante similar a nivel de sintaxis a la de *Agent0* pero incluye operadores de planificación para realizar acciones y alcanzar objetivos. Al igual que en el caso anterior se trata de un lenguaje experimental y no de desarrollo general.

3.2.3.2 Lenguajes de Validación de Agentes

Fisher [Fis94] argumenta que tanto *Agent0* como PLACA son lenguajes de programación de agentes en los cuales la relación entre la lógica y el lenguaje de programación interpretado es muy

débil, remarcando que en realidad no se puede asegurar que el programa esté ejecutando la verdadera lógica asociada a las intenciones del diseñador del agente. Como solución propone el lenguaje *Concurrent MetateM*. Un sistema desarrollado por medio de este lenguaje contiene múltiples agentes ejecutándose concurrentemente y comunicándose entre ellos por medio de mensajes asíncronos enviados en modo multidifusión (*broadcast*). Cada uno de estos agentes está programado en base a una especificación en lógica temporal del comportamiento que se desea que tenga dicho agente. La ventaja respecto a las implementaciones de AOP es que dicha especificación lógica es la que es ejecutada de forma directa, de esta forma es posible probar que el procedimiento usado para ejecutar el agente es correcto, por lo tanto satisface la especificación.

EJEMPLO DESIRE–1:

```
task structure AAA
     subcomponents      own_process_control,
                        update_world_state_info,
                        diagnose_fault,
                        manage_communication;
     links              incomming_info,
                        incomming_world_state_info,
                        info_on_current_world_state,
                        boa_info, request_info,
                        required_boa_info,
                        fault_results, info_to_output,
                        boa_requests, faults;
end task structure AAA
```

La semántica de *Concurrent MetateM* está estrechamente relacionada con la semántica de la propia lógica temporal, lo cual implica que la especificación y validación de sistemas desarrollados por medio de este lenguaje es verificable [FW93].

Al mismo tiempo de *Concurrent MetateM*, Brazier *et al.* han diseñado el entorno de desarrollo para sistemas multi-agente DESIRE (*DEsign and Specification of Interacting REasoning components*) [BDKJT97]. Este entorno permite la especificación e implementación de un diseño conceptual de un sistema de agentes, mediante el cual se pueden expresar de forma explícita el conocimiento, la interacción y la coordinación de las diferentes tareas y capacidades de razonamiento de sistemas de agentes. La aplicación directa de DESIRE es la implementación del sistema ARCHON [WJM94, JCL+95, CJ96] para control de aplicaciones industriales, actualmente usado para el transporte de energía eléctrica en España.

DESIRE está pensado para poder especificar de forma clara tanto las funcionalidades internas de los agentes como las funcionalidades de cooperación entre agentes. La especificación mane-

jada por DESIRE define tanto agentes como la arquitectura completa en base a tareas, las cuales están caracterizadas por un conjunto de entradas, un conjunto de salidas así como sus relaciones con otras tareas. Asimismo, la comunicación entre agentes y con elementos externos se especifica en términos de intercambios de información. Uno de los objetivos de DESIRE es proporcionar construcciones con las cuales los patrones de razonamiento puedan ser modelizados de forma explícita. En el ejemplo *DESIRE–1*, extraído de [BDKJT97], puede verse cual es la plantilla de una tarea especificada en DESIRE y en el ejemplo *DESIRE–2* la especificación de un enlace.

EJEMPLO DESIRE–2:

```
link info_on_current_world_state: object-object
     domain             update_world_state_information
     output             world_state_obs
     codomain           diagnose_fault
     input              world_state_obs
     sort links         (World_state_info, World_state_info)
     object links       identity
     term links         identity
     atom links         (obs_in_current_world_state
                        (I:World_state_info),
                        obs_in_current_world_state
                        (I:World_state_info)):
                        <<true,true>,<false,false>>
end link
```

Un estudio más detallado de DESIRE y *Concurrent MeteteM* y su aplicación a la modelización de agentes se puede encontrar en el informe realizado por Mulder, Treur y Fisher [MTF98].

3.2.3.3 Lenguajes para Agentes Móviles

Dentro de este tipo de lenguajes, el entorno de desarrollo de agentes móviles más conocido es *TeleScript* [Whi96], comercializado por la empresa General Magic, Inc. Probablemente sea el primer lenguaje para desarrollo de agentes que se distribuye de forma comercial [GMI95a, GMI96a, GMI95b, GMI96b].

En realidad *TeleScript* es un epígrafe bajo el que se encuadra una tecnología formada por una familia de conceptos y técnicas desarrollados por dicha compañía. Los dos conceptos centrales de esta tecnología son los *places* o lugares y los agentes. Los *places* son localizaciones virtuales que pueden ser ocupadas por los agentes. Por su parte, los agentes son proveedores y consumidores de bienes en las aplicaciones de mercado electrónico (*electronic marketplace*) para las que está dirigido *TeleScript*. Los agentes desarrollados por medio de estas tecnologías son procesos móviles que por medio del comando `GO` son capaces de congelar y codificar su estado (incluido pila y registros),

para reactivarse de nuevo y proseguir con su ejecución en la nueva localización. Otra característica de los agentes desarrollados por medio de *TeleScript* es su capacidad para comunicarse entre si, por medio de una red de comunicaciones si no se encuentran en la misma localización o por medio de otros mecanismos en caso contrario.

La tecnología *TeleScript* está compuesta por cuatro piezas fundamentales:

- El propio lenguaje *TeleScript*, diseñado para realizar funciones de navegación, transporte, autenticación y control de acceso entre otras.
- El motor *TeleScript*, que actúa como interprete del lenguaje, manteniendo los *places* y planificando la ejecución, comunicación y transporte de los agentes, así como proporcionando una interfaz de comunicación con otro tipo de aplicaciones.
- El conjunto de protocolos *TeleScript* que permiten la codificación y decodificación de los agentes así como el transporte entre diferentes *places*.
- Por último, se encuentran una serie de aplicaciones y herramientas pensadas para asistir en el desarrollo de agentes por medio de esta tecnología.

Como propuesta alternativa a *TeleScript*, Gray [Gra95, Gra97], desarrolló el lenguaje *Agent Tcl*, dirigido a solventar una serie de problemas encontrados dentro de los lenguajes de programación para agentes móviles. Estos problemas son:

- El proceso de migración no puede ocurrir en puntos arbitrarios de la ejecución del agente o requiere la captura explícita de la información de estado a nivel del agente.
- La comunicación entre agentes o no existe o bien es costosa y difícil.
- No existen mecanismos de seguridad.
- Los agentes han de ser escritos en un lenguaje específico, en algunos casos bastante complejo.
- Sólo existen implementaciones sobre hardware no estándar (como es el caso de PDAs *Personal Digital Assistants*).
- Otra parte de las implementaciones sólo son ejecutables sobre determinadas plataformas UNIX, muy concretas.
- El código fuente no está disponible para la comunidad de investigadores en el campo.

Como solución a estos problemas propone el lenguaje *Agent Tcl*, como una variación del lenguaje de *scripts* Tcl, con las siguientes características:

- ❒ Reducir el proceso de migración a una sola instrucción, como es el caso del comando `GO` de *TeleScript*, pero que puede invocarse en cualquier momento de la ejecución del agente.
- ❒ Proporcionar una comunicación transparente entre agentes.
- ❒ Soportar múltiples lenguajes y mecanismos de comunicación para permitir añadir estos fácilmente.
- ❒ Ser ejecutable en cualquier plataforma UNIX y de fácil adaptación a plataformas no-UNIX.
- ❒ Proporcionar seguridad y tolerancia a fallos de forma efectiva.
- ❒ Estar disponible para dominio público.

La arquitectura de *Agent Tcl* está dividida en cuatro niveles diferentes, el nivel inferior está conformado por los diferentes protocolos y mecanismos de comunicación (TCP/IP, protocolos de correo electrónico, ...). Por encima, se encuentra el motor o servidor, encargado de aceptar nuevos agentes (bajo un proceso de autentificación), mantener un espacio de nombres jerárquico (para identificar los agentes) y gestionar los procesos de comunicación. Encima del motor se encuentran los diferentes intérpretes para los distintos lenguajes que pueden usar *Agent Tcl*, la inclusión de un nuevo lenguaje únicamente implica añadir un nuevo intérprete a este nivel. El nivel superior de la arquitectura es donde se encuentran situados los agentes.

Además de *TeleScript* y *Agent Tcl*, existen otros lenguajes para desarrollo de agentes móviles, como son *Tacoma* [JvRS95a, JvRS95b, JvRS96], *Aglets* [LC96, Lan97, LOKK97], $M\emptyset$ [TDMH94, DMTH95, Tsc97] *IBM Itinerant Agents* [CGH+95], entre otros.

3.2.3.4 Lenguajes de Propósito General

Un agente, igual que cualquier sistema o aplicación software, puede desarrollarse por medio de múltiples lenguajes de propósito general. Si bien, estos lenguajes carecen del soporte proporcionado por los anteriores para traducir de una forma automática la especificación a la implementación del agente (como DESIRE y *Concurrent MetateM*), de proporcional movilidad (*Telescript*) o de presentar un marco formal similar a los lenguajes derivados de la AOP. Sin embargo, el uso de lenguajes tales como C, Pascal, Prolog, etc. posee la ventaja de la experiencia de los equipos de desarrollo en estos lenguajes, así como la existencia de entornos de desarrollo (compiladores, depuradores y otras herramientas) ampliamente probadas y capaces de generar código muy eficiente.

Una de las mayores desventajas derivadas de la utilización de estos lenguajes en el desarrollo de agentes, es que implica el diseño del agente partiendo desde cero. Una ayuda menor en este aspecto es el uso de lenguajes orientados a objetos, pues como ya se comentó al comienzo de

este capítulo, si bien un agente es algo más que un objeto, no cabe duda que ambos conceptos comparten una serie de características comunes.

3.3 Comunicación entre Agentes

Los lenguajes de comunicación de agentes o ACL (*Agent Comunication Languages*) representan uno de los campos de estudio más activos dentro de la comunidad de investigadores de la tecnología de agentes. El objetivo de estos lenguajes es proporcionar un mecanismo mediante el cual diferentes agentes puedan intercambiar información (modelos de conocimiento, datos o peticiones). Debido a que uno de los principales problemas que pretende atajar la filosofía de agentes es la integración de herramientas heterogéneas representadas como agentes. Esta integración requiere a menudo el diálogo entre dos agentes para negociar ciertos parámetros que han de relacionar ambas herramientas. Con el objeto de construir entornos en los cuales este tipo de interacciones sean posibles es necesario definir lenguajes de comunicación entre agentes. El principal problema presente en la definición de estos lenguajes es que estos han de ser suficientemente abiertos como para no restringir el tipo de comunicación que por medio de ellos se puede hacer; por otro lado deben de estar suficientemente bien detallados como para poder ser operativos y útiles.

Los problemas que aparecen a la hora de intercomunicar dos elementos diferentes, incluso cuando ya se ha concertado un lenguaje común de intercomunicación, se deben a problemas sintácticos o de vocabulario. Estos problemas se agrupan, por un lado en *inconsistencias*, al usar la misma expresión o término para denotar dos conceptos diferentes y por otro lado aparecen problemas de *incompatibilidades* al usar cada uno de los componentes un mismo término o expresión diferente para hacer referencia al mismo concepto. La solución a estos problemas pasa por la utilización de un lenguaje estándar para realizar la comunicación entre agentes. Esto no implica que grupos restringidos de agentes puedan utilizar dentro de colectivos reducidos de agentes otros lenguajes no estándar. Pero, lo más adecuado es que todos los agentes fuesen capaces de comunicarse (como mínimo) en lenguaje estándar.

3.3.1 Tipos de Lenguajes

Genesereth et al. [GGH+95] como parte de su estudio para el programa I^3 realiza una clasificación de los tipos de lenguajes de comunicación entre agentes, distinguiendo:

① Lenguajes Procedimentales.

② Lenguajes Declarativos.

3.3.1.1 Lenguajes Procedimentales

Estos lenguajes están compuestos por directivas mediante las cuales se describe un procedimiento que el receptor del mensaje ejecuta. La ejecución de este procedimiento puede contener:

- una secuencia de peticiones a realizar e incluso una serie de estructuras condicionales de control que indiquen en que orden realizar dichas peticiones,
- sentencias que alteren su información sobre el entorno o sobre el estado de otro agente o
- sentencias que contienen información resultado de mensajes anteriores a otros agentes.

El tipo de lenguajes utilizados en esta línea son lenguajes interpretados como Perl, Python o Tcl o variaciones sobre los mismos como Agent Tcl [Gra95, Gra97] o TeleScript [Whi96]. Estos dos últimos lenguajes también son usados como lenguajes para desarrollo de agentes, especialmente agentes móviles.

Este tipo de lenguajes proporcionan una gran potencia al transmitir programas enteros en cada petición permitiendo una gran variedad de posibilidades a la hora de especificar los objetivos de una petición. Adicionalmente, estos lenguajes suelen ser ejecutados directamente tras el proceso de interpretación, y por lo general de una forma muy eficiente.

Como desventajas se encuentran todas las características de ser lenguajes puramente procedimentales. Los emisores de peticiones no disponen de información relativa a los receptores que puede ser necesaria para construir la petición. Tal información puede implicar conocer que tipo de funcionalidades proporciona dicho receptor, para construir programas correctamente ejecutables en el destino. Por otro lado, se trata de mecanismos de comunicación básicamente unidireccionales. Por último, este tipo de formato de mensajes es difícilmente combinable, en el sentido en que varias peticiones no pueden ser unificadas en una sola solicitud que negocie todas ellas. La combinación o ejecución en paralelo de varios programas puede implicar la interferencia entre peticiones. Por lo general esto no ocurre porque los agentes que funcionan con este tipo de lenguajes suelen proporcionar esquemas de comunicación uno-a-uno.

3.3.1.2 Lenguajes Declarativos

En contraste con los anteriores, estos lenguajes hacen uso de sentencias declarativas para hacer definiciones, afirmaciones, peticiones, respuestas, etc. Para ser completamente útiles, los lenguajes declarativos deben proporcionar mecanismos para transmitir cualquier tipo de información incluidos procedimientos.

Actualmente los sistemas implantados tienden a implementar soluciones basadas en lenguajes procedimentales (básicamente debido a tratarse de lenguajes con los que los desarrolladores ya están familiarizados), pero se considera que una apuesta más a largo plazo es el uso de lenguajes declarativos. Dentro de estos últimos, el más representativo es KQML, como parte del proyecto KSE (*Knowledge Sharing Effort*) de ARPA.

El planteamiento teórico de los lenguajes declarativos se basa en lo que se denomina *teoría del acto del habla* (*speech act theory*) [Aus62, Sea69]. Esta teoría desarrollada por diversos lingüistas presenta una explicación al modelo de comunicación humano. Dentro de esta teoría las declaraciones realizadas por un sujeto se describen en base a *acciones*, en el mismo sentido que las acciones en el mundo real. Dentro de estas acciones se distinguen tres aspectos, por un lado la declaración propiamente dicha (en inglés: *locution*), por otro el tipo de acción declarada (*illocution*) y por último el modo en el cual la declaración afecta al receptor (*perlocution*). Centrándose en el segundo grupo (*illocutions*) se puede indicar que está compuesto por los matices que indican si la acción es, por ejemplo, afirmada, consultada, solicitada, etc. En el contexto de los lenguajes de comunicación entre agentes los mecanismos utilizados se han dirigido hacia el campo de los elementos de este segundo grupo, los que se denominan tradicionalmente como "*performative messages*".

3.3.2 KSE

KSE (*Knowlegde Sharing Effort*) [PFPS+92, Nec94] es una iniciativa de la agencia americana ARPA para facilitar la reutilización e intercambio de bases de conocimiento entre sistemas. Actualmente la coordinación del grupo recae en un consorcio de varias universidades y centros de investigación. El objetivo final de este grupo de investigadores es definir, desarrollar y probar una infraestructura y una tecnología que lo soporte que permita a los grupos que participan construir sistemas mayores y con más capacidades de las que podrían desarrollar por separado. Los resultados de las investigaciones realizadas por el consorcio de KSE se plasman en:

- Unas especificaciones de dominio público, así como de las implementaciones asociadas a dicha tecnología.
- Informes, artículos y propuestas en el campo de la representación e intercambio de conocimiento entre sistemas.
- Una librería de uso público de demostraciones.

Los investigadores de KSE argumentan que en la actualidad la definición y diseño de sistemas basados en el conocimiento (como pueden ser los agentes) por lo general implica la planificación

y construcción completa de una nueva base de conocimiento. Evidentemente, este planteamiento no permite abordar problemas de gran escala, los cuales se deben basar en la reutilización de componentes previamente desarrollados y probados en otros proyectos, permitiendo de este modo que el desarrollo del nuevo sistema implique únicamente la definición de las funcionalidades que son incorporadas a los componentes que ya existían. En resumen, se reduciría al desarrollo de los nuevos modelos de conocimiento y razonamiento propios del nuevo sistema. Adicionalmente, la existencia de un mecanismo de representación común del conocimiento permitiría la cooperación de sistemas basados en el conocimiento que serían capaces de intercambiarse información.

El proyecto de KSE se encuentra dividido en cuatro diferentes grupos de trabajo encargados de cada una de las cuatro áreas de trabajo definidas en el primer *workshop* que abordo el tema. Estos grupos son:

① **Interlingua**: Grupo centrado en la traducción entre diferentes lenguajes de representación (tanto a nivel de diseño como durante la ejecución del sistema). Este grupo está liderado por Richard Fikes y Mike Genesereth de la Universidad de Stanford.

② **KRSS** (*Knowledge Representation System Specification*): Grupo encargado de la definición de construcciones comunes a varias familias de lenguajes de representación. Este grupo está representado por Bill Swartout de la Universidad de Southern California y Peter Patel-Schneider de los Laboratorios Bell.

③ **Interfaces Externos** (*External Interfaces*): El grupo liderado por Tim Finin y Jay Weber de la Universidad de Maryland Baltimore County es el encargado de la definición de mecanismos de interacción entre bases de conocimiento y otros sistemas. Un punto de principal interés de este grupo son los protocolos de comunicación entre sistemas basados en el conocimiento.

④ **Bases de Conocimiento Compartidas/Reusables** (*Shared/Reusable Knowledge Bases*) Este grupo se encarga de establecer un consenso en lo referente al contenido de bases de conocimiento compartidas. Los encargados de este grupo son los investigadores de la Universidad de Maryland Baltimore County, Tom Gruber and Marty Tenenbaum.

3.3.3 KQML

KQML, es el producto más claro del trabajo de investigación realizado por el grupo de interfaces externos. Su aplicación como lenguaje común de comunicación entre agentes software [MLF96] han hecho de el uno de los elementos fundamentales para la integración de los sistemas desarrollados por medio de la tecnología de agentes.

La sintaxis de KQML (*Knowledge Query and Manipulation Language*) fue propuesta por el grupo de Tim Finin [ARP93, LF97] como parte de la propuesta KSE de ARPA. Se podría definir de una forma general KQML como un lenguaje mediante el cual los componentes de un sistema basado en el conocimiento (particularizando el caso de los agentes) intercambian mensajes. Los mensajes KQML representan solicitudes de actuación (*performatives*) que un agente envía a otro para requerir una acción, el intercambio de información o la coordinación en cualquiera de estas tareas. Los mensajes del lenguaje son suficientemente simples para hacerlo fácil de generar y analizar y pueden contener como argumento de información sentencias o procedimiento descritos en otros lenguajes (C, Lisp, Prolog, Perl, . . .).

El formato de los mensajes KQML está compuesto por una palabra clave, denominada *performative*, que representa el tipo de solicitud transmitida. Existen una serie de *performatives* estándar que de ser implementadas han de serlo de la forma descrita en la especificación del lenguaje. El significado de la *performative* representa el tipo de acción que se solicita del receptor, en el sentido de añadir nuevos elementos en su base de conocimiento, eliminarlos, consultarlos o solicitar que el agente actúe sobre el entorno donde encuentra. La información objeto de dicha petición (información a borrar, insertar o consultar) se encuentra definida en uno de los campos del mensaje denominado `:content`. Este campo es el que puede estar descrito en cualquier otro lenguaje. La forma en la que se especifica qué lenguaje se utiliza para el campo `:content` es por medio del campo `:language`.

EJEMPLO KQML–1:

```
(ask-all
         :sender       Ag1
         :receiver     Ag2
         :in-reply-to  id0
         :reply-with   id1
         :language     SQL
         :ontology     DataMining
         :content      "SELECT dbid,name
                        FROM DATABASES
                        WHERE owner=DMUSER")
```

En este ejemplo *KQML–1* se puede ver como, por medio de un mensaje KQML, el agente `Ag1` solicita al agente `Ag2` toda la información referente a una consulta. Esta consulta está expresada en SQL, como indica el parámetro `:language`. La consulta en sí está asociada al argumento `:content`.

Categoría	*Performatives* Reservadas
Discurso	`ask-if`, `ask-all`, `ask-one`, `stream-all`, `eos`, `tell`, `untell`, `deny`, `insert`, `uninsert`, `delete-one`, `delete-all`, `undelete`, `achieve`, `unachieve`, `advertise`, `unadvertise`, `subscribe`
Intervención	`error`, `sorry`, `standby`, `ready`, `next`, `rest`, `discard`
Red y Acceso a Recursos	`register`, `unregister`, `forward`, `broadcast`, `transport-address`, `broker-one`, `broker-all`, `recommended-one`, `recommended-all`, `recruit-one`, `recruite-all`

Tabla 3.1: *Performatives* de KQML (extraídas de [LF97])

EJEMPLO KQML–2:

```
(tell
         :sender       Ag2
         :receiver     Ag1
         :in-reply-to  id1
         :reply-with   id2
         :language     SQL
         :ontology     DataMining
         :content      "0113 MEDICAL
                        0116 PATIENTS
                        0210 X-RAYS")
```

Esta consulta puede ser respondida por el agente `Ag2` de la forma indicada en el ejemplo *KQML–2*. En este caso hace uso también de SQL para presentar los resultados.

El estándar de KQML contempla una docena de posibles *performatives* diferentes cuyo significado se encuentra perfectamente definido dentro de las especificaciones del lenguaje. Estas *performatives* se encuentran clasificadas en diferentes grupos, como recoge la Tabla 3.1.

En conjunto un mensaje KQML, puede verse dividido en tres niveles diferentes:

① **El nivel de Contenido**: Representado por el argumento `:content` que conforma la información interna del mensaje cuya representación está fuera del alcance de la especificación del lenguaje. Puede ser cualquier sentencia expresable en cualquier lenguaje (C, Prolog, lenguaje natural).

② **El nivel de Mensaje**: Descrito por el uso de las *performatives* reservadas y que representa el objetivo principal de KQML.

③ **El nivel de Comunicación**: Que agrupa los protocolos de comunicación (se podría decir que los 5 niveles inferiores del modelo OSI). En relación a este nivel, KQML asume las siguientes características relativas a los mecanismos de transporte:

- ❒ Los agentes están conectados por medio de enlaces unidireccionales capaces de transportar mensajes de tamaño limitado.
- ❒ Estos enlaces pueden llevar asociado un retardo de transmisión determinado.
- ❒ Cuando un agente recibe un mensaje, este conoce en enlace de entrada por donde ha sido recibido.
- ❒ Cuando un agente envía un mensaje, puede especificar en enlace de salida que desea utilizar.
- ❒ Los mensajes llegan al destino en el mismo orden en el que fueron enviados.
- ❒ El envío de mensajes es fiable. Esta restricción sólo es relevante si se desea proporcionar una red de agentes fiable.

Desde el punto de vista práctico, este nivel puede estar soportado por redes TCP/IP, usando protocolos particulares o servicios estándar (correo electrónico), pueden usarse arquitecturas de objetos distribuidos como CORBA (Benech et al. [BDR98]) o pueden estar soportados por hardware de comunicaciones específico de maquinas multiprocesadoras o masivamente paralelas (e.g. Hypercube).

En contra de KQML, Cohen y Levesque [CL97] describen tres principales fallos dentro del concepto de este lenguaje, por un lado, consideran que en ciertas *performatives* la definición es ambigua y vaga y en ese sentido la semántica en ciertas definiciones no es suficientemente precisa; en segundo lugar, y como consecuencia de lo anterior, no es posible determinar cuando una aplicación/agente es capaz de comunicarse conforme al estándar por lo tanto dicho estándar en si no representa un punto de consenso ni referencia adecuado. Por último, Cohen y Levesque recalcan la falta de ciertas *performatives* que hagan completo el lenguaje, un ejemplo son las sentencias de *commit* para un bloque de mensajes. A pesar de estos problemas, KQML representa el más claro punto de referencia dentro de los mecanismos de comunicación entre agentes. Esto hace que tanto en Estados Unidos como en Europa existan numerosos proyectos y herramientas que hacen uso de KQML como lenguaje de comunicación principal.

3.3.4 Ontologías

En la base de los lenguajes de comunicación entre agentes se encuentran las ontologías que representan el conocimiento compartido entre emisor y receptor de un mensaje para hacer que la interpretación del mismo sea idéntica para ambos agentes.

En los ejemplos de KQML vistos, se puede observar la presencia del campo `:ontology`. Este argumento expresa el conjunto de términos que representan el vocabulario en el cual está expresada la consulta. Esto diferencia los aspectos sintácticos de la consulta, indicados por el lenguaje

de consulta, de los aspectos semánticos. Las connotaciones semánticas están ligadas al campo de aplicación del sistema, es decir al entorno en el cual están ubicados los agentes. Por ejemplo, dentro del campo de sistemas de *Data Mining*, la ontología utilizada hará referencia a los componentes, información y, en general, términos usados dentro de dicho dominio. Estos elementos deben estar claramente definidos y expresados para evitar problemas de *inconsistencia* e *incompatibilidad*.

En el campo de los agentes software, las ontologías contienen los términos usados en la comunicación así como las restricciones elementales asociadas a los mismos. El reconocimiento de una ontología por parte de un agente implica la definición, atributos y relaciones entre los términos usados así como entre dichos términos y las restricciones. La existencia de una ontología común es una condición indispensable para poder definir un lenguaje de comunicación entre agentes válido (como puede ser el caso de KQML), esto se debe a que la gran mayoría de la carga semántica de las declaraciones recae en los términos utilizados, mientras que los lenguajes utilizados únicamente proporcionan la sintaxis mediante la cual las expresiones son construidas en base a dichos términos.

El desarrollo de ontologías apropiadas para los diferentes campos de aplicación de los sistemas basados en agentes es un tema muy amplio. Actualmente los trabajos de investigación realizados al respecto se agrupan en tres diferentes categorías:

3.3.4.1 Ontologías *Ad Hoc*

Esta aproximación es la más utilizada de hasta el momento, en especial en el desarrollo de prototipos de sistemas multi-agente. Este tipo de ontologías se encuentran embebidas de forma implícita en los agentes del sistema y es definida por los desarrolladores y diseñadores del mismo. Es decir que una aplicación dentro de un dominio determinado, por ejemplo para el tratamiento de datos médicos, construye un vocabulario para definir los términos del dominio que utiliza. El significado de dichos términos y de las restricciones de los mismos se definen en fase de diseño del sistema y se encuentra directamente ligado al propósito y tareas concretas que el sistema ha de desempeñar.

Esta aproximación plantea dos problemas principalmente:

① Si cada sistema define sus ontologías, restringidas al estricto dominio de aplicación de dicho sistema en concreto, entonces las posibilidades de intercomunicar dos agentes, aunque se encuentren trabajando sobre el mismo dominio son muy reducidas o incluso nulas. Esta situación se da aunque dichos agentes usen un lenguaje de comunicación común (e.g. KQML). Por ejemplo, dentro del dominio de *Data Mining*, si un agente realiza la petición:

EJEMPLO KQML–1:

```
(ask-all
         :sender       Ag1
         :receiver     Ag2
         :in-reply-to  id0
         :reply-with   id1
         :language     SQL
         :ontology     DataMining
         :content      "SELECT dbid,name
                        FROM DATABASES
                        WHERE owner=DMUSER")
```

El agente que lo reciba debe coincidir con el primer agente en la interpretación de los términos `dbid`, `name`, `DATABASES` y `owner`. Como el otro agente reconozca como único término posible para indicar el propietario de una base de datos `db--owner` en lugar de `owner`, entonces este mensaje no puede ser interpretado de la misma manera. Esta situación se da aunque ambos agentes usen KQML como lenguaje de comunicación y acepte sentencias SQL como contenido de dichos mensajes.

② El otro problema de este tipo de ontologías se centra en la escalabilidad de los sistemas desarrollados en base a las mismas. Puesto que existe una relación directa entre el diseño de las tareas del agente y los términos de la ontología que reconoce, la ampliación de un agente de este tipo para intentar aplicarlo a otro dominio ligeramente más amplia implica rediseñar por completo el agente para acomodarlo a un nuevo conjunto de términos.

3.3.4.2 Ontologías Estándares

Para superar las limitaciones de las soluciones anteriores, y dentro del proyecto global KSE de ARPA, se han establecido una serie de líneas de investigación que permitan intercambiar ontologías por medio de un formato de representación común. Estos esfuerzos en el desarrollo se encuentran recogidos dentro del proyecto de compartición de ontologías (*ontology sharing project*). Para el programa KSE de ARPA, la existencia de una ontología común es indispensable para que una serie de agentes heterogéneos sean capaces de comunicarse. Una ontología común permite compartir para los mismos términos las mismas restricciones y sobre todo la misma semántica.

Los dos principales resultados de esta línea de investigación han sido los proyectos de ARPA: KIF y Ontolingua.

❒ **KIF** (*Knowledge Interchange Format*) [Gin91, GF92, ANS98]: El proyecto KIF proporciona una lengua común entre agentes para expresar el conocimiento que disponen y para comunicar

mensajes a otro agente. KIF es un formato de representación similar en su sintaxis a Lisp, mediante el cual es posible expresar consultas, peticiones etc. Es posible hacer uso de sentencias KIF como contenido de mensajes y solicitudes KQML, como por ejemplo:

EJEMPLO KQML–3:

```
(ask-all
         :sender       Ag1
         :receiver     Ag2
         :in-reply-to  id0
         :reply-with   id1
         :language     KIF
         :ontology     DataMining
         :content      "(DATABASES
                        (= dbid ?id)
                        (= name ?n)
                        (= owner 'DMUSER))"
```

La consulta *KQML–3* es similar a la expresada en el ejemplo *KQML–1* pero expresada por medio de KIF. En ambos casos se asume que la ontología *DataMinig* define los términos de la consulta. Dichas definiciones pueden encontrarse expresadas también en KIF.

❒ **Ontolingua**: [Gru93, FFPR95] Este es un segundo proyecto financiado por ARPA para la reutilización de ontologías. *Ontolingua* proporciona una herramienta independiente de dominio para la traducción de ontologías a un formato común. Esta herramienta es completamente independiente del proceso de especificación y definición de una ontología. Al igual que KIF no se trata de una implementación sino una ayuda para conseguir "*una especificación explícita de una conceptualización*" ([Gru93] pp: 199), definición dada por Gruber de una Ontología. Por medio de este lenguaje es posible representar ontologías propias de aplicaciones concretas a diferentes lenguajes de representación de conocimiento (tales como *Loom* [Mac91]).

Las ontologías especificadas por medio de Ontolingua contienen elementos tales como objetos, eventos, recursos, restricciones, etc.; y su definición se encuentra asistida por una herramienta de edición de ontologías (*Ontology Editor* [FFR97]) existiendo además un servidor que contiene diferentes ontologías de distintos dominios (*Ontolingua Server*[1] [FFR96]).

Las sentencias y axiomas de Ontolingua se encuentran escritos en una extensión de la notación de KIF, así como por medio de lenguaje natural. Existen disponibles varias herramientas

[1] http://WWW-KSL-SVC.stanford.edu:5915/

además del editor antes citado, tales como herramientas de validación de la sintaxis y traductores a otros lenguajes.

3.3.4.3 Ontologías Globales

Al margen de las iniciativas indicadas anteriormente, existen proyectos para definir y generar ontologías más globales, en principio orientadas no a dominios concretos (como las anteriores), sino a su utilización en cualquier ámbito. Las dos propuestas más relevantes dentro de esta línea son

- *WorldNet* [Mil95], una base de datos con más de 150.000 entradas que recoge términos relacionados con determinados contextos, funciones y relaciones con otro términos (sinónimos, antónimos, ...). En realidad *WorldNet* no está dirigida directamente para el desarrollo de agentes, pero su aplicación como complemento a determinadas ontologías es muy interesante.

- *Cyc* [GL94, Len95] es un proyecto planteado a principio de 1.984 con la ambición de capturar y representar el conocimiento descrito como "*sentido común*". La orientación original del proyecto *Cyc* era dotar a los sistemas expertos desarrollados hasta el momento de un conocimiento amplio (no necesariamente profundo) fuera del ámbito estricto de aplicación del mismo. Es decir que un sistema experto en la diagnosis de enfermedades cardiacas fuese capaz de responder cuestiones fuera de sus restringido dominio de aplicación.

  Cyc actualmente contiene más de 100.000 términos, así como más de un millón de axiomas introducidos en su desarrollo, además de varios millones inferidos y almacenados por *Cyc*. Si finalmente el ambicioso proyecto *Cyc* alcanza las metas fijadas, supondrá una verdadera revolución dentro del campo, no sólo de agentes software, sino de los sistemas de información en general (incluido WWW). Actualmente y a corto plazo los sistemas multi-agente a desarrollar se encuentran restringidos a campos muy concretos y el uso de ontologías globales como la proporcionada por *Cyc* es tan costoso que en un futuro inmediato no se considera factible su aplicación directa.

3.4 Agentes y *Data Mining*

Como se ha visto, el empuje de la tecnología de agentes presenta un nuevo panorama en el cual el paradigma de programación orientada a agentes permite abordar nuevos proyectos o mejorar el diseño y las prestaciones de aplicaciones actuales. Nwana [NN98] augura como uno de los campos en los cuales la teoría de agentes puede cuajar con mayor impacto es el campo de *Data Mining*. Esto se debe a que tres de las materias concretas en las cuales los agentes tienen una aplicación

concreta y validada con éxito, se corresponden con funciones o tareas presentes en los sistemas de *Data Mining*:

- ❒ Asistentes inteligentes de usuario.
- ❒ Recopiladores de Información.
- ❒ Elementos de Control Automático.

3.4.1 Asistentes Inteligentes de Usuario

El proceso de consulta, o más concretamente de construcción de una consulta, de *Data Mining* es muy complejo. Esta complejidad se debe a que, finalmente, se está solicitando la ejecución de uno o varios algoritmos de inteligencia artificial, *Machine Learning* o estadística que se encuentra ajustado por multitud de diferentes parámetros. Además, por lo general, el proceso de resolución de una consulta de *Data Mining* implica la aplicación de varios de estos algoritmos.

Esto se traduce en que para el usuario el proceso de consulta se convierte en un trabajo de selección de algoritmos para los que se tiene que fijar los parámetros. Estos parámetros representan umbrales de selección, límites de iteraciones, factores de corrección, etc. Por lo general, sólo los usuarios con un conocimiento muy profundo de los algoritmos de análisis de datos son capaces de manejar todos esos factores. Estos usuarios distan mucho de los usuarios con menor conocimiento de los fundamentos matemáticos de las operaciones asociadas a la consulta, pero con una gran experiencia en el dominio al cual pertenecen los datos.

En resumen se puede afirmar que el rango de conocimientos, experiencia y habilidad de los usuarios de sistemas de *Data Mining* es muy amplio, e idealmente un sistema de este tipo debe de proporcionar a los usuarios la capacidad de ajustar todos los factores de sus consulta. La única forma de hacer que usuarios con diferentes perfiles accedan al sistema y aprovechen sus capacidades es por medio de asistentes personales que colaboren con los usuarios a definir la consulta. Estos agentes permitirán que un usuario inexperto pueda realizar una consulta únicamente indicando que desea modelos que definan una determina circunstancia; por otra parte también permitirán que un analista experto de datos seleccione las fuentes de datos más adecuadas a un determinado dominio, aun sin tener gran conocimiento del mismos.

3.4.1.1 Recopiladores de Información

Una de los requisitos del proceso de extracción de conocimiento a partir de datos es que dichos datos originales debe contener suficiente información como para satisfacer la consulta solicitada. Actualmente, los datos objeto de análisis se almacenan en un *Data Warehouse* diseñado como resultado del proceso de integración de múltiples fuentes de datos. Este proceso de integración es muy costoso y tanto más cuando el número y variabilidad de las fuentes de datos es alto.

Dao y Perry [DP96] representan un claro ejemplo de la aplicación de la tecnología de agentes a la construcción de redes de información. Esta idea permite acceder a la información contenida en sistemas de información dispersos, en concreto los autores la aplican a los escenarios de bases de datos federadas y a la red *World Wide Web*.

Dentro de este enfoque, Papyrus [GKM+98], es un sistema de *Data Mining* distribuido que hace uso de la tecnología de agentes para compartir e interpretar modelos representados en MIF (*Model Interchange Format*) que han sido generados por diferentes motores de *Data Mining* que trabajan sobre conjuntos de datos diferentes para posteriormente construir un modelo global. Esta aproximación puede verse también como un uso de agentes como intérpretes o asistentes en el proceso de recopilación de información, aunque en este caso la información recopilada se encuentra ya procesada y se corresponde con conocimiento (modelos y patrones) extraídos de los datos.

3.4.1.2 Elementos de Control Automático

La tercera línea en la cual se puede aplicar la tecnología de agentes al proceso de *Data Mining* es como elementos de control del mismo. Los elementos de control de un sistema complejo son los encargados de gestionar todos los parámetros internos de un sistema necesarios para que dicho sistema sea operativo.

La lógica de control para un sistema cualquiera crece exponencialmente según se añaden nuevas funcionalidades y componentes al sistema, debido principalmente a la aparición de múltiples interdependencias entre dichos componentes. Un segundo factor que añade mayor complejidad al proceso de control de un sistema es la concurrencia. En el momento en el que el sistema tiene más de un flujo de ejecución (sistemas distribuidos o servicios multiusuario) la complejidad se multiplica.

Hasta el momento, la gran mayoría de grandes sistemas simplificaban el problema fijando en cada uno de los componentes del mismo el comportamiento y los parámetros con los que debían operar dichos componentes. Es decir las funcionalidades del elemento y cómo controlarlas esta-

ban mezcladas. Esta solución plantea un claro problema de falta de flexibilidad y nula reacción de cara a los cambios en el entorno en el que opera el sistema o a nuevas funcionalidades del sistema.

Una solución para controlar esta complejidad y proporcionar un elevado grado de flexibilidad y adaptabilidad es delegar el control. El control puede ser delegado localmente al componente en agentes suficientemente cercanos al componente como para poder sondear su estado y tomar acciones correctivas. Estos agentes pueden cooperar con otros agentes para resolver problemas que afectan a varios componentes o incluso al sistema completo.

Esta aplicación no es un caso exclusivo de los sistemas de *Data Mining*, en general, pues se esta imponiendo como modelo de control en grandes redes. Ejemplos en esta línea se han utilizado en el campo de las telecomunicaciones, [WV95, WV98], HYBRID [Som96] para gestión de redes ATM o MAITE [GH92] implantado en España para gestión de fallos sobre IBERCOM. El uso de agentes con esta intención puede ser adecuado para cualquier sistema complejo, que esté sujeto a una serie de características, pero nunca han sido aplicados al área de *Data Mining* con este fin.

Parte II

ESTUDIO Y RESOLUCIÓN DEL PROBLEMA

Capítulo 4

MOTIVACIÓN Y ESTUDIO DEL PROBLEMA

Índice General

4.1 Motivación y Objetivos

En los capítulos anteriores se han estudiado las características que los sistemas de *Data Mining* distribuido más representativos soportan y las tecnologías en las que se basan para proporcionarlas. Todo nuevo sistema desarrollado debe tomar las funcionalidades ya alcanzadas como base e incorporar nuevas características. El objetivo de este capítulo es definir el problema a resolver en este trabajo de Tesis. El fin del trabajo es la especificación de la arquitectura de control para un sistema de *Data Mining* el cual ha de responder a las siguientes características:

- Integrado,

- Distribuido,
- Extensible y
- Flexible.

Parte de estas características, en concreto la **integración**, la **distribución** y la **extensibilidad** se encuentran ya soportadas por algunos sistemas tanto comerciales como experimentales, en la gran mayoría de casos por separado. Sin embargo, la **flexibilidad**, en el sentido que se explicará a lo largo de este capítulo no ha sido aplicada aun al problema de *Data Mining*. El objetivo de este trabajo de investigación consiste en la definición de un sistema de *Data Mining* capaz de proporcionar una flexibilidad en la configuración y funcionamiento del sistema, manteniendo el resto de características, identificadas como esenciales por los diferentes sistemas que las soportan. La flexibilidad de control es el punto central de este trabajo de investigación y la consecución de un sistema que manteniendo las otras características sea posible de reconfigurarlo dinámicamente es el objetivo de esta aportación.

No obstante, dotar a un sistema de *Data Mining* de las características mencionadas tiene una serie de problemas que se tendrán que resolver. La identificación de los problemas que se interponen entre la situación actual de los sistemas de *Data Mining* y el objetivo final de este trabajo de investigación se exponen a continuación.

4.2 Integración con los SGBD

En los sistemas actuales de tratamiento de información el patrón más habitual es el diseño de una nueva aplicación encargada del proceso de análisis, recuperación o sumarización de la información que trabaja sobre un sistema gestor de bases de datos. Adicionalmente, a este esquema se le pueden añadir nuevos niveles asociados con funciones relativas a la presentación de resultados al usuario, definiéndose arquitecturas estratificadas del tipo mostrado en la Figura 4.1.

Esta organización proporciona una serie de notable de ventajas, entre las cuales las más importantes son: (i) que la sustitución de una de las aplicaciones por otra de iguales o mayores prestaciones representa una modificación local muy modularizada que no repercute de forma traumática al resto del sistema, (ii) el uso como sistema de acceso a los datos de una base de datos hace que dicha información pueda ser compartida por varias aplicaciones siendo responsable de su gestión el SGBD únicamente, evitando de esta forma inconsistencias y (iii) los SGBD actuales son aplicaciones suficientemente probadas y de muy altas prestaciones en la manipulación de datos y la alternativa de rediseñarlo presenta un elevado riesgo de errores en la programación de dichas

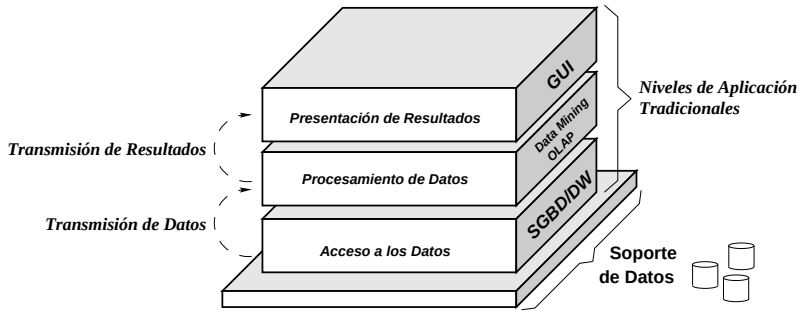


Figura 4.1: Sistemas de tratamiento de datos actuales

funcionalidades. Estos puntos representan una clara ventaja de este tipo de diseño para la gran mayoría de aplicaciones que han de tratar datos, pero no para todas.

Dentro del grupo de aplicaciones para los cuales esta organización no es adecuada se encuentran los sistemas que han de tratar grandes cantidades de datos, bien para extraer patrones (como en el caso de *Data Mining*) o para elaborar informes (como OLAP o herramientas de análisis multidimensional). Para estos sistemas que han de proporcionar un tiempo de respuesta adecuado a la vez que procesan tablas con cientos o miles de millones de registros, una arquitectura tan desacoplada entre el SGBD y la aplicación que procesa los datos es un severo *handicap*. Para cada uno de los registros (tuplas) de la base de datos que ha de ser procesado es necesario realizar el siguiente proceso:

① Cargar los datos en el espacio de memoria del SGBD, recuperándolos del almacenamiento físico en disco duro.

② Copiar los datos al espacio de memoria del módulo de proceso.

③ Sincronizar el proceso de proporcionar los datos a procesar y la ejecución de dicho proceso de los datos.

Cuando esta serie de operaciones se realizan sobre un volumen de información de varios Megabytes o Gigabytes de datos, entonces cobran una mayor importancia problemas tales como:

❒ la duplicación de recursos (memoria) usados por el SGBD y por las herramientas de niveles superiores y

❒ la utilización de mecanismos de sincronización que no aprovechan la ejecución en paralelo de la recuperación de datos y el procesamiento de la información, dándose constantemente situaciones de bloqueo entre ambas tareas.

Esta situación en general produce una degradación en el rendimiento que puede ser cuantificada en minutos u horas en el caso de procesar grandes bases de datos. Este mismo problema, evidentemente se da en todas las aplicaciones que obtienen los datos de un SGBD, aunque en menor escala. La diferencia radica en que en el momento en que se incrementan los órdenes de magnitud de la información a tratar este retardo aumenta de forma considerable y su importancia desplaza a las ventajas antes citadas.

4.2.1 Análisis del Problema

Una forma más sencilla de comprender el problema es comparar el procesamiento de la información (por parte de una herramienta de *Data Mining*) con la realización de una operación simple sobre una tabla de datos. Si suponemos que el primer proceso requiere una única pasada sobre toda la tabla, a priori nos encontramos con dos tareas de la misma complejidad ($O(n)$) sobre el tamaño de la tabla de información. Y si bien, en ambos casos se ha de recorrer la tabla de datos una única vez, ¿por qué la operación realizada por el SGBD es mucho más rápida?. La respuesta nada tiene que ver con la naturaleza del proceso aplicado sobre los datos puesto que si el proceso de selección realizado de forma interna por el SGBD se realizase de forma externa, entonces se observaría el mismo tipo de retardo en la ejecución. Esto se puede constatar en un entorno como el ilustrado en la Figura 4.2. Este experimento evalúa la realización de una operación simple del álgebra relacional dentro y fuera del gestor. La operación de cuenta y selección se ejecuta fuera del SGBD y dentro de él con y sin índices sobre los atributos de selección.

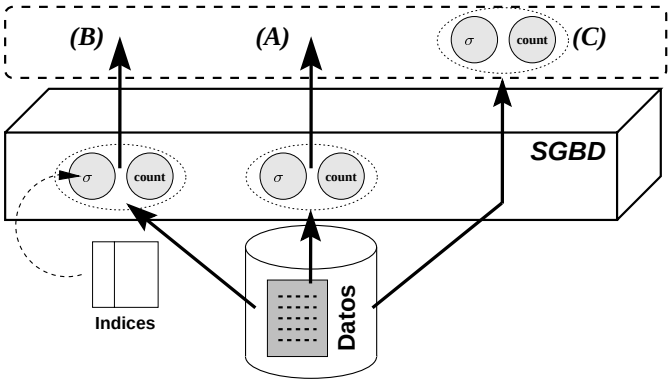


Figura 4.2: Esquema del mecanismo de experimentación

La diferencia en los resultados obtenidos en cada uno de los casos se debe, como ya se indicó anteriormente a las operaciones de cambio de contexto y duplicación de datos en memoria. Adicionalmente, en otros casos, factores aun más determinantes son la utilización de las estructuras internas de organización del gestor, como son los índices. La utilización de atributos indexados,

para cualquier operación implica un incremento en la eficiencia considerable, el problema es que esas estructuras son internas del gestor, por lo tanto su uso se restringe a las operaciones realizadas de forma interna y nunca su beneficio puede ser accesible para procesos realizados fuera del SGBD.

Si se generaliza el análisis, se puede argumentar que la realización de operaciones de tratamiento de datos dentro y fuera del gestor difieren notablemente en eficiencia. Esta diferencia es apreciable cuando el volumen de información es mayor y el número de veces que los datos se recorren aumenta (como es el caso de muchos de los algoritmos de *Data Mining* que requieren varias pasadas sobre los datos para obtener resultados finales). Este problema se encuentra amplificado de forma notable si el interfaz de acceso usado para acceder a los SGBD supone capas de tratamiento muy *pesadas* computacionalmente hablando. Este es el caso de las APIs de acceso de tipo ODBC o JDBC, cuya eficiencia es considerablemente baja en comparación a mecanismos de acceso basados en protocolos específicos de los gestores.

La solución al problema pasa por poder ejecutar estas operaciones de procesamiento masivo de datos de forma similar a la que se realizan el resto de operaciones internas del SGBD (selecciones, proyecciones, ordenaciones, *joins*, ...). Esto implicaría poder acceder a las estructuras internas de indexación (tablas *hash*, *B-trees*, ...), así como a otros recursos internos, además de compartir el mismo espacio de memoria usado por los SGBD para realizar sus operaciones, evitando así la duplicación de los datos a la hora de realizar una consulta. La forma de alcanzar esta solución es por medio de la **integración** entre los SGBD y las herramientas de procesamiento de datos:

> **Definición** INTEGRACIÓN: Definición y aplicación de las operaciones tradicionales de un SGBD y de las nuevas operaciones de procesamiento de datos (*Data Mining*, OLAP, ...) de una forma homogénea, optimizando el uso de los recursos (memoria, ...) y herramientas (índices, ...) internas del gestor.

4.2.1.1 Problemas Derivados

La consecución final del proceso de integración entre SBGD y herramientas de *Data Mining*. plantea el problema de rediseñar los gestores de bases de datos actuales, de forma que las nuevas operaciones puedan acceder a las estructuras internas de los mismos. Esto, evidentemente, es muy complejo y costoso, y en la actualidad sólo se está realizando de forma experimental por ciertos fabricantes de SGBD para un conjunto reducido de operaciones adicionales, tal es el caso de la versión fuertemente acoplada del algoritmo *Apriori* de Agrawal [AS96b]. Dentro de la construcción de sistemas de *Data Mining*, la integración con el SGBD es una de las características de *Papyrus* [GBST99] el cual usa *Osiris* un SGBD con funcionalidades restringidas y diseñado para aplicaciones

de *Data Mining*.

4.3 Distribución en los Sistemas de *Data Mining*

La utilización de entornos distribuidos permite utilizar la potencia y los recursos de una serie de nodos de computación que se encuentren interconectados por medio de una tecnología de comunicación. Una de las mayores ventajas del desarrollo de sistemas distribuidos es la utilización de toda la potencia de los nodos que lo componen para resolver problemas de cálculo masivo o que requieren una gran capacidad de computación. Esta faceta es de especial interés para los sistemas de tratamiento de datos (*Data Mining* entre ellos) que requieren potencia de cálculo suficiente para analizar millones de registros en el menor tiempo posible así como de procesar datos de naturaleza distribuidos (almacenados en diferentes localizaciones). Adicionalmente, aunque en menor medida, otras aportaciones de los sistemas distribuidos como es la duplicación de funcionalidades con vista a proporcionar tolerancia a fallos o la utilización de plataformas heterogéneas pueden resultar interesantes para este entorno.

La distribución actual en los sistemas de tratamiento de información se basan en arquitecturas cliente/servidor de dos o tres niveles en las cuales cada nivel asume una serie de funcionalidades. Un ejemplo de este tipo de arquitecturas es la del sistema comercial de *Data Mining* desarrollado por IBM, Intelligent Miner, [Tka98]. El esquema general de este sistema (tal y como se observa en la figura 4.3) está compuesto por una interfaz de usuario desarrollada en Java que actúa como aplicación de acceso al sistema. Por debajo de esta aplicación se encuentra el núcleo del sistema encargado de la ejecución de los algoritmos. Finalmente, todo el sistema descansa sobre un SGBD, por lo general DB2.

Intelligent Miner representa el típico grado de distribución que alcanzan la gran mayoría de sistemas de *Data Mining*. Este modelo de arquitecturas presentan un esquema de distribución muy simple cuyo grano de distribución es muy grande. Este sistema está conformado por tres grandes elementos monolíticos: herramientas de interfaz cliente, el conjunto completo del motor y el gestor de base de datos. En el mejor de los casos, cada uno de estos elementos puede estar ubicado en diferentes nodos. Esta configuración implica que para poder prestar un servicio de calidad en una situación de carga masiva del sistema, se requiere disponer de dos computadores de gran potencia, una de ellas para ejecutar el motor del sistema y otra para albergar al SGBD. Esta alternativa implica una inversión en costes elevada que podría ser reducida, equilibrando la carga de las dos tareas más costosas (ejecución de algoritmos y gestión de datos) en varios nodos de computación.

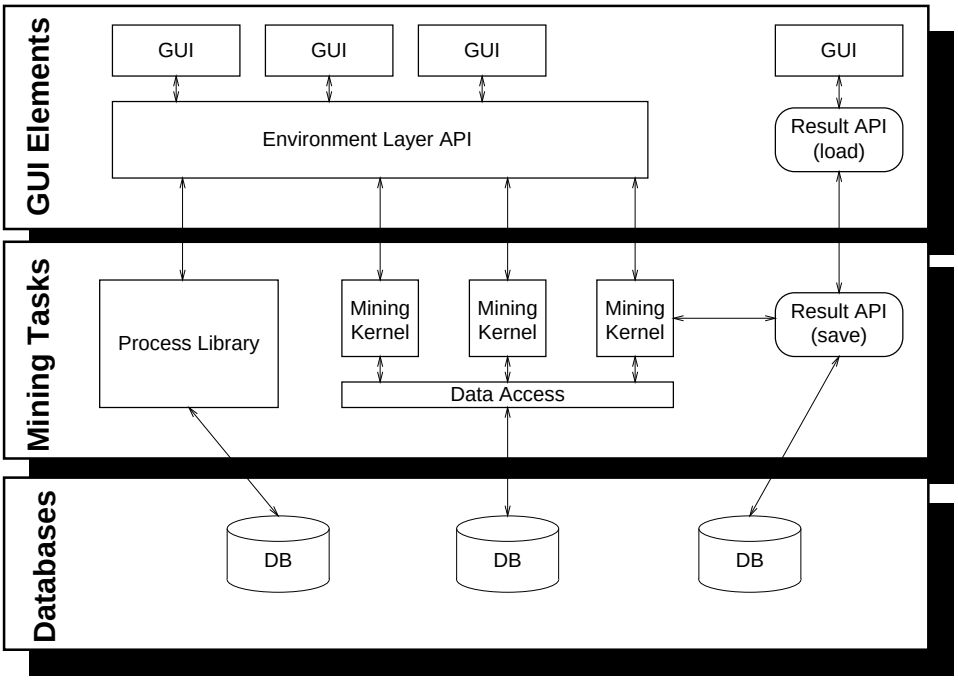


Figura 4.3: Niveles de la arquitectura de IMiner

4.3.1 Análisis del Problema

A la hora de representar gráficamente un esquema de distribución se plantea el análisis del problema sobre dos ejes. Por un lado se encuentran emplazadas las diferentes localizaciones o nodos sobre los cuales se pretende desplegar el sistema. Por otro lado, un segundo eje recoge las distintas funcionalidades o tareas en las que se descompone funcionalmente el sistema. En casos como el antes citado, la división en funcionalidades implica automáticamente su ubicación en un único nodo, proporcionando una representación lineal sobre el plano de estos dos ejes (ver Figura 4.4). Las arquitecturas cliente/servidor representan un primer grado de independencia entre localizaciones y tareas. Permitiendo que grupos de funciones sean proporcionados por distintos nodos del sistema.

El grado máximo de distribución se alcanza cuando no existe ningún anclaje entre las funcionalidades y el nodo del sistema que la ejecuta, permitiendo la redundancia de funcionalidades entre varios nodos, la migración de una funcionalidad de un nodo a otro, etc. Las ventajas que esto proporciona, son: (i) un mejor aprovechamiento de redes de nodos de computación interconectados, debido a que la distribución óptima de las funcionalidades debe equilibrar la carga de los nodos del sistema; (ii) beneficios tales como la tolerancia a fallos (ante caída de nodos) y la escalabilidad de las instalaciones de este tipo. Todo ello representa un punto a favor de los sistemas distribuidos,

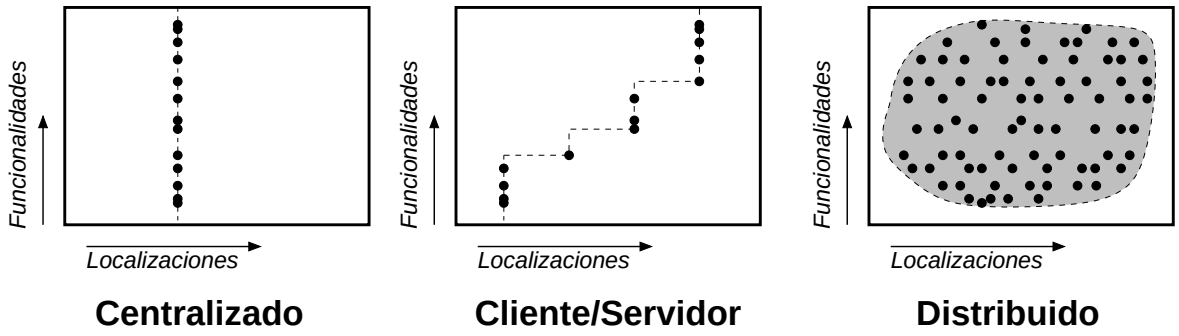


Figura 4.4: Tipos de sistemas según su distribución

especialmente para ciertos tipos de aplicaciones o problemas.

En resumen, dentro del entorno de sistemas de tratamiento de datos, la característica de **distribución** es muy interesante. Una definición más concreta del significado de distribución dentro de este campo sería:

> **Definición** DISTRIBUCIÓN: División de las funcionalidades proporcionadas por un sistema en componentes autónomos que puedan ser ejecutados en nodos de computación diferentes (o en un mismo nodo) y que intercambien mensajes entre ellos, de forma que cooperando en conjunto resuelvan las funcionalidades generales del sistema global.

4.3.1.1 Problemas Derivados

El principal problema que ha impedido hasta el momento el despegue de los sistemas distribuidos ha sido la complejidad en el diseño de los mismos, este problema se agudiza debido a una serie de factores, tales como: (i) trabajar con entornos heterogéneos (diferentes arquitecturas o sistemas operativos), (ii) acceder a recursos/funciones compartidas de forma concurrente, (iii) dimensionar no sólo recursos locales (cpu o memoria) sino recursos globales (ancho de banda o conexiones), entre otros problemas. Numerosas líneas de investigación están actualmente trabajando a diferentes niveles y desde diferentes perspectivas en la definición de herramientas (formales o de desarrollo) y tecnologías para poder resolver estos problemas. Este trabajo obvia muchos de dichos problemas, asumiendo que una tecnología de *middleware* apropiada es capaz de proporcionar interoparbilidad, centrándose de esta forma en aspectos organizativos o arquitectónicos del problema.

Por otro lado, a menudo se asocia la característica de integración, antes propuesta con centralización, resultando difícil su conjunción con un entorno distribuido. Esto, sin embargo, es total-

mente erróneo, el concepto de integración denota la definición de todas las funcionalidades de un sistema de forma homogénea, mientras que el concepto de distribución representa la localización de dichas funcionalidades en diferentes nodos interconectados. Como se puede ver, ambos términos son complementarios y en ningún caso incompatibles.

4.4 Sistemas Extensibles de *Data Mining*

La gran mayoría de sistemas complejos llevan asociado un gran número de módulos en los cuáles se implementan diversas funcionalidades. El diseño de estos módulos y sobre todo de las funcionalidades se realiza en fase de definición del sistema. Existen, sin embargo, situaciones a lo largo del tiempo de vida de un sistema en las cuáles es imprescindible replantear el diseño de dichos módulos o funcionalidades, debido a la exigencia de nuevas funciones o la implementación más eficiente de las ya definidas. Todo esto repercute en el mantenimiento y actualización de estos sistemas. Sin llegar al extremo del rediseño y definición de los componentes, es muy habitual la necesidad de reconfigurar los sistemas complejos durante su ejecución, añadiendo o retirando componentes *dinámicamente* mientras el sistema está en ejecución.

Un problema patente en el diseño actual de los sistemas de *Data Mining* es la evolución sufrida por el conjunto de capacidades y funcionalidades que proporcionan dichos sistemas. Estas funcionalidades determinan cómo procesan los datos estos sistemas y, por tanto, cuáles son los tipos de resultados obtenidos. La experiencia ha demostrado que a lo largo de los últimos años, en áreas del tratamiento de datos como las técnicas de *Data Mining*, se ha producido un proceso de investigación muy efectivo. Fruto de este trabajo se han obtenido multitud de algoritmos, técnicas y mecanismos capaces de obtener resultados interesantes cuando se aplican sobre diferentes entornos. Esto lleva a pensar que dicho campo se encuentra en constante evolución, y por lo tanto, el trabajo de expertos en áreas de *Machine Learning*, Estadística, Inteligencia Artificial, etc. producirá nuevos algoritmos o mejoras sobre los ya existentes que harán que los mecanismos actuales queden superados por estas nuevas aportaciones.

4.4.1 Análisis del Problema

Los primeros sistemas de *Data Mining* consistían en aplicaciones que ejecutaban *ad hoc* uno o como mucho un reducido grupo de algoritmos sobre ficheros plano de texto. Estos sistemas han seguido evolucionando hacia entornos más complejos en los cuales se integran múltiples técnicas de análisis y distintas funcionalidades de preproceso y tratamiento de datos, así como de herramientas de visualización de resultados. Esta secuencia de pasos evolutivos han apuntado al di-

seño de sistemas comerciales que proporcionan interfaces de programación mediante los cuales es posible diseñar nuevos componentes capaces de ser integrados en el sistema. En el campo de *Data Mining* sistemas tales como Clementine [ISL95] o IMiner [IBM99] permiten definir nuevas funcionalidades por medio de APIs de desarrollo de nuevos algoritmos. En este sentido se define extensibilidad como:

> **Definición** EXTENSIBILIDAD: Capacidad para ampliar, modificar o eliminar, y en general actualizar, el conjunto de funcionalidades o la forma en la que estas funcionalidades son proporcionadas de una forma que implique la menor repercusión a nivel de integridad, funcionamiento y servicio del sistema.

4.4.1.1 Problemas Derivados

El proceso de extensión de las funcionalidades de un sistema presenta tres posibles cuestiones de relevancia. En primer lugar, la capacidad de ser extensible de un sistema implica de forma inherente una mayor complejidad a nivel de diseño. Si un sistema se plantea como un entorno cerrado, su diseño y posterior implementación es más simple que si se pretende proporcionar la posibilidad de añadir nuevos elementos al mismo. Esta cuestión tiene una mayor repercusión en el caso de tratarse de sistemas de por si complejos y a la larga representa el límite de la capacidad de extensión de un sistema, en la práctica indica el punto a partir del cual la posibilidad de modificar o añadir nuevas funcionalidades representa una grado de complejidad excesivo, requiriendo que ciertas funciones elementales del sistema no sean extensibles sin un rediseño de una parte considerable del mismo.

En segundo lugar, a la hora de diseñar un sistema extensible un elemento clave es la definición de las interfaces entre los componentes del mismo. La utilización de una interfaz estándar permite que la integración de nuevos componentes al sistema requiera únicamente dar soporte a dicha interfaz. Estas interfaces han de ser definiciones abiertas suficientemente generales como para evitar:

- por un lado, que el diseño de nuevos elementos choque con una interfaz que de alguna forma restrinja la forma en la cual este componente interactúa con el sistema,
- por otro lado, una interfaz de componente demasiado genérica complica en exceso la interacción de componentes. Este problema se da en ciertos protocolos o lenguajes de comunicación que con la intención de ser genéricos plantean una interfaz abierta que a la hora de aplicar de forma práctica resulta imprescindible restringirla.

La elección de una buena interfaz de conexión entre componentes representa una tarea de diseño

crucial que debe encontrar un punto de equilibrio entre los dos extremos citados.

Por último, es argumentable que para ciertas aplicaciones el esfuerzo por dotarlas de extensibilidad sacrifique un porcentaje del rendimiento de dicho sistema. Esto se puede deber a la necesidad de una organización determinada de los componentes de dichos sistemas y una encapsulación adecuada de sus funcionalidades. Esto restringe el acceso directo de ciertas partes de los sistemas a datos o información de otro subsistema de forma directa. Este mecanismo representa una ganancia a nivel de eficiencia a costa de saltar la estructuración en el diseño de un sistema. El objetivo de una solución adecuada es mantener un equilibrio entre extensibilidad y eficiencia razonable.

4.5 Sistemas de *Data Mining* con Configuración Flexible

A medida que los sistemas complejos s diseñan en base a componentes intercambiables y ampliables, formando entornos abiertos, es importante no perder de vista otra de las restricciones fundamentales de este tipo de aplicaciones, la eficiencia. Para un sistema de *Data Mining* que requiere analizar por medio de diferentes mecanismos las peticiones de varios usuarios sobre múltiples tablas de datos de miles o millones de registros este factor es fundamental. Un sistema con estas prestaciones requiere unos recursos (grandes requerimientos de proceso, grandes cantidades de memoria y grandes cantidades de espacio en disco) que deben ser administrado de forma adecuada.

La problemática de la eficiencia en cualquier sistema requiere la definición previa del criterio usado para evaluar dicha característica. De esta forma, para ciertos sistemas la medida de la eficiencia se puede representar en base al número de operaciones simultáneas que el sistema puede servir concurrentemente con un retardo de respuesta aceptable. En otro tipo de entornos la eficiencia puede medirse por medio del tiempo requerido para la resolución de una operación compleja ejecutada por el sistema dedicado. Estos criterios y otros similares del funcionamiento del sistema difieren en diferentes decisiones (que se han denominado **decisiones de control**) que el sistema aplica durante la ejecución de sus tareas. Independientemente de cuales sean las decisiones de control, al evaluar la eficiencia en sistemas complejos se tienen dos facetas:

① La eficiencia individual de los componentes, relacionada con la realización de las funciones atómicas que proporcionan. Es decir, cómo se ejecutan las tareas realizadas completamente por dicho componentes (sin la intervención, soporte o interferencia con otros). Esta eficiencia puede ser alcanzada por medio de nuevas versiones de los componentes, diseñado e implementado de forma más eficiente.

② El otro factor de la eficiencia aparece en el momento en que dos o más componentes del sistema presentan alguna dependencia entre sí. Estas dependencias pueden ser relaciones de cooperación, cuando una función realizada por un componente requiere la colaboración de otro, o pueden ser relaciones de interferencia presentes cuando dos o más componentes compiten por algún tipo de recurso.

El concepto de extensibilidad expuesto en la sección anterior permite la evolución de un sistema que optimice la eficiencia tal y como se presenta en ①. Permitiendo incorporar implementaciones más eficientes de los componentes críticos del sistema. Pero la faceta ② se resiente, sin embargo, cuando los componentes del sistema no son fijos y no es posible prefijar en fase de diseño un plan de operación para componentes cuyas ejecuciones son interdependientes. Tal es el caso de que un nuevo componente puede hacer un uso abusivo de algún recurso cuyo reparto equitativo no se planteo en fase de diseño de componentes o del sistema propiamente dicho.

4.5.1 Análisis del Problema

El control de los componentes que operan con alguna interdependencia entre sí se presenta como la pieza clave para conseguir mejoras significativas en lo referente a la eficiencia de estos sistemas. Este concepto de control representa la posibilidad de regular la interacción entre varios componentes y la administración equitativa de los recursos del sistema. El concepto de control puede extenderse también a ciertas decisiones internas del componente y no sólo a su interacción con otros componentes. De esta forma se habla de **control** como "*del conjunto de decisiones y mecanismos que regulan el comportamiento interno de los componentes así como sus interacciones con otros componentes*". Basándose en esta definición cada componente debe de dividir las tareas que realiza en funciones meramente operacionales y por otro lado los criterios que indican cuándo y cómo se realizan estas funciones. Dichos criterios de control pueden afectar al resultado final obtenido mediante la función operacional o simplemente afectar a nivel de cómo es realizada internamente dicha operación.

Durante el análisis realizado del problema de la distribución de un sistema se ha presentado una representación en base a dos ejes perpendiculares donde se representan por un lado las funcionalidades del sistema y por otro las localizaciones donde un sistema puede ejecutarse. Las consideraciones sobre control hacen posible dibujar un tercer eje (perpendicular a ambos) que separe las tareas propias de la ejecución de las operaciones del sistema de las de control y gestión de las mismas. Todo esto define esta nueva dimensión de **operacional/control** de la forma presentada en la Figura 4.5.

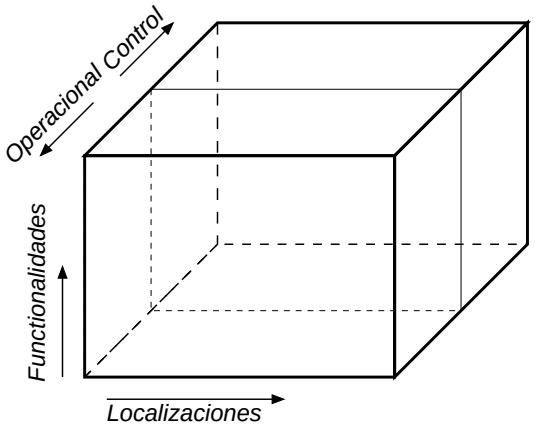


Figura 4.5: Despliegue de dimensiones de la arquitectura

Esta separación entre las tareas operacionales y las decisiones de control, al igual que la división de funcionalidades y localizaciones, permite que un sistema pueda alterar elementos ubicados en unas coordenadas determinadas de estos ejes, repercutiendo mínimamente en el resto de elementos vecinos. Por ejemplo, de la misma manera que una funcionalidad situada en un nodo podía ser rediseñada y desplazada a otro nodo, un componente podrá mantener sus funciones operacionales (es decir realizar las mismas operaciones) pero será posible modificar los criterios de control de forma independiente. A esta posibilidad de cambiar los criterios de control se la denomina flexibilidad de control:

Definición FLEXIBILIDAD DE CONTROL: Posibilidad de alterar (incluso de forma dinámica) los criterios de control de un conjunto de componentes de un sistema o incluso de todo el sistema sin alterar las funciones operacionales del sistema. Por medio de esta característica es posible optimizar el comportamiento del sistema a diferentes restricciones o medidas de eficiencia, en resumen a diferentes restricciones de funcionamiento.

En la actualidad los sistemas de *Data Mining* están diseñados de forma que las funciones de trabajo (proceso o manejo de datos, por ejemplo) se encuentran entremezcladas con las funciones de control (procesos de optimización, equilibrado de carga, *tunning*, gestión de *buffers*). La vía para alcanzar una flexibilidad de control que permita configurar el sistema fácilmente con diferentes criterios de control consiste en desacoplar estas dos capas de funciones (considerando las relaciones que hay entre ellas) desarrollan por separado las funciones de trabajo y las de control. Esta faceta no se encuentra soportada por ningún sistema de *Data Mining* hasta la fecha y es el objetivo central de este trabajo.

4.6 Conclusiones y Objetivos

Las necesidades de integración, distribución, extensibilidad y flexibilidad tal y como han sido definidas y analizadas en las secciones anteriores determinan el objetivo de este trabajo.

El planteamiento de soluciones a las diferentes necesidades descritas se ha dividido en dos líneas de solución a dos diferentes niveles:

① Por un lado definir arquitectura general de componentes distribuidos, descrita formalmente para que cumpla una serie de requisitos. Esta arquitectura tendrá que estar orientada hacia la problemática de control definida en el último apartado.

② En segundo lugar, realizar el diseño abstracto de un nuevo sistema de *Data Mining* que cumpla las restricciones de extensibilidad e integración comentadas anteriormente.

Estas dos líneas de acción convergen en el desarrollo del sistema diseñado en abstracto sobre la arquitectura planteada. Esta intersección de las dos vías planteadas tendrá que proporcionar:

(i) integración: el diseño abstracto contemplará tanto funcionalidades de *Data Mining* como tareas de un gestor de BD de forma homogénea,

(ii) distribución: hará uso de una arquitectura distribuida como soporte para el desarrollo del sistema (la arquitectura proporcionará mecanismos de comunicación, ejecución y similares sobre un entorno distribuido). El diseño deberá tener en consideración que el sistema a desarrollar es distribuido como una restricción o característica del mismo,

(iii) extensibilidad: el diseño del sistema deberá dejar abiertas las posibilidades para añadir nuevos algoritmos, operaciones, etc. Por otro lado la arquitectura deberá permitir la incorporación de nuevos componentes al entorno distribuido y

(iv) flexibilidad de control: los componentes de la arquitectura deberán diferenciar entre funcionalidades operacionales y de control y el diseño deberá definir para cada componente cuales son las responsabilidades de cada una de las partes en cada caso.

	Arquitectura	Diseño
Integración		✔
Distribución	✔	
Extensibilidad	✔	✔
Flexibilidad	✔	✔

Tabla 4.1: Líneas de solución y necesidades

La tabla 4.1 muestra para cada una de las necesidades mencionadas, si ésta es proporcionada por la arquitectura de componentes distribuidos o por el diseño del sistema.

Capítulo 5

ARQUITECTURA DISTRIBUIDA MOIRAE

Índice General

5.1 Introducción

El primer elemento de la solución propuesta es una arquitectura general para sistemas distribuidos. Esta arquitectura está compuesta por una serie de modelos, así como por una herramienta formal que la describe. La definición completa de esta nueva arquitectura de control, denominada **MOIRAE** (*Management of Operational Interconnected Robots using Agent Engines*), se obtiene por la combinación de los siguientes elementos:

- **Modelos de la Arquitectura**:
  - **Modelo de Componente**: Define las piezas elementales de la arquitectura, los componentes. Asimismo, se definen las partes en las que éstos se dividen, determinando las funciones determinadas o generales para cada uno de los elementos de un componente.
  - **Modelo de Relación**: Describe las relaciones entre componentes dentro de la arquitectura. Este modelo indica cómo intercambian solicitudes los componentes, creando relaciones entre sí, definiendo también como se gestionan dichas relaciones.
  - **Modelo de Arquitectura**: Basándose en los modelos de componente y relación se expresa la manera en la que la arquitectura opera y resuelve peticiones de forma cooperativa.
  - **Modelo de Control**: Define cómo se realizan las acciones de control, en concreto mediante la definición de operaciones aplicables a los elementos básicos de control (las políticas).
- Expresión formal de un sistema multicomponente y de los términos: estado, operación, política, etc. Esta expresión es una herramienta formal que puede permitir validar ciertos aspectos del diseño de un sistema.

Como se verá, esta arquitectura no se restringe a sistemas de *Data Mining* distribuidos únicamente, sino que puede ser aplicable a otros entornos. No obstante, la aplicación a otros problemas, así como las repercusiones que pudiera tener sobre los modelos o la arquitectura en sí, no se encuentran tratados en esta aportación.

5.1.1 Conceptos de la Arquitectura

Antes de entrar en detalle con la descripción de los modelos de la arquitectura MOIRAE, se definen los elementos básicos de forma intuitiva:

5.1.1.1 Concepto de Componente

El término central de la arquitectura se ha denominado **componente**. Un componente representa cada uno de los elementos del entorno distribuido que forman parte del sistema. Dependiendo de la tecnología sobre la que se implante esta arquitectura estos componentes pueden estar desarrollados como objetos (si el entorno es capaz de trabajar con ellos) o procesos/tareas (desde el punto de vista más tradicional).

> **Definición** COMPONENTE: Elemento básico de la arquitectura MOIRAE. Este elemento proporciona una serie de funcionalidades al resto de componentes con los que interactúa, ocultado la complejidad de asociada a la realización de dichas funciones. La forma en la que proporciona estas funcionalidades y la interacción con el resto de componentes está sujeta a una reglas y a estructuras determinadas.

5.1.1.2 Concepto de Código de Componente

Los componentes, como se ha indicado son o bien objetos o procesos dentro de un sistema determinado. El código ejecutable de dichos objetos o procesos se denomina **código del componente**.

> **Definición** CÓDIGO DE COMPONENTE: Código ejecutable asociado a cada componente del sistema. Un mismo código puede ejecutarse múltiples veces, arrancando varios componentes del sistema.

5.1.1.3 Concepto de Clase de Componente

Las definición de las funcionalidades que proporciona un determinado componente al resto de elementos del entorno se encuentra descrita por medio de una **clase de componente**.

> **Definición** CLASE DE COMPONENTE: Descripción de las funcionalidades proporcionadas por cada uno de los tipos de componentes del entorno. Esta descripción describe las características (operaciones e información) que el resto de los elementos de la arquitectura conocen y a las que pueden acceder. La clase de componente no describe cómo ha de encontrarse implementada o diseñada la resolución de las operaciones proporcionadas, sólo las describe.

Los tres términos clase de componente, código de componente y componente se encuentran relacionados. El primero describe para un grupo de componentes cuáles son las funcionalidades que proporcionan al resto de elementos de la arquitectura. El segundo término representa una de las múltiples implementaciones de las operaciones descritas por una clase de componente, siendo posible que existan varias implementaciones de una misma clase de componente. Es decir, el

término componente se asocia a instancias concretas de las definiciones (clase de componente), asociadas a una implementación concreta (código de componente) y que se encuentran en ejecución como un elemento de un sistema.

5.1.1.4 Concepto de Operación

Desde el punto de vista externo, una clase de componente está definida por las funcionalidades que proporciona al resto de elementos del entorno, es decir, qué tareas le pueden ser encomendadas. Estas tareas que un componente puede realizar se han denominado **operaciones**.

> **Definición** OPERACIÓN: Interfaz de métodos proporcionados por un componente que realizan parcial o totalmente tareas del sistema. Las operaciones son invocadas por otros componentes del sistema (como parte del proceso de realización de sus propias operaciones) o por elementos externos al sistema (usuarios, otros sistemas, . . .).

Las operaciones se encuentran definidas por la clase de componente e implementadas en un código de componente determinado y se solicitan a componentes concretos del sistema que se encuentren en ejecución.

5.1.1.5 Concepto de Estado

La definición de una clase de componente, además de incluir las **operaciones**, describe las variables de **estado** del componente. Esta información juega un papel importante en la realización de las operaciones.

> **Definición** ESTADO: Información interna del componente que describe la situación del mismo. Esta información representa cómo se encuentran en un momento determinado los elementos internos del componente y afecta a la ejecución de las operaciones del mismo. La información de estado se encuentra descrita por el valor de las variables de estado definidas en la clase del componente.

5.1.1.6 Concepto de Acciones Control

Debido a que la información de estado afecta a cómo se realizan las operaciones o incluso a si dichas operaciones son realizables o no, es necesario definir una serie de mecanismos que aseguren que la realización de las operaciones concluye de forma exitosa, independientemente del estado de los componentes. Estos mecanismos se denominan **acciones de control** y son las encargadas de corregir el estado del sistema para que éste funcione de forma correcta.

> **Definición** ACCIONES DE CONTROL: Funciones internas del sistema que no están asociadas a ninguna operación del exterior, pero que al ser ejecutadas por los componentes, su estado es modificado, de tal forma que las operaciones solicitadas a dicho componente sean ejecutadas satisfactoriamente. Estas funciones del sistema pueden afectar a un único componente o varios a la vez.

5.1.1.7 Políticas de Control

La forma en la cual las acciones de control se invocan para alterar el estado de los componentes del sistema se describe por medio de una serie de criterios denominados **políticas de control**.

> **Definición** POLÍTICAS DE CONTROL: Información usada por el sistema para definir cómo son activadas las diferentes acciones de control del sistema. Estas políticas determinan los criterios de funcionamiento del sistema, desde decisiones de alto nivel hasta criterios puntuales de funcionamiento de una determinada operación de un componente.

Las políticas de control actúan como elementos de configuración dinámica del sistema en relación a la optimización del sistema para un determinado rendimiento o tipo de funcionamiento.

5.1.2 Características de la Definición de Componentes

La definición de componentes por medio de la descripción dada en la clase de componente asociada debe cumplir las siguientes características:

- Atomicidad: Intentando encapsular cada una de las operaciones elementales del sistema en componentes diferentes que puedan realizarlas de forma independiente al resto de componentes. Estos componentes son capaces de realizar las operaciones elementales del sistema de forma interna sin interactuar con otros componentes. Estas operaciones se denominan operaciones atómicas del sistema.

- Cooperación: Asignando las operaciones más complejas del sistema a componentes encargados de dividirlas en otras operaciones más simples que pueden ser solicitadas a otros componentes. Un caso particular de este tipo de relación son los componentes que se asisten mutuamente.

- Completitud: Todas las funcionalidades son realizadas por al menos un componente y todos los componentes realizan al menos una función útil para el sistema.

- Divisibilidad: Todos los escenarios de aplicación del sistema (secuencia de acciones solicitadas desde el exterior) pueden ser descompuestos en operaciones, las cuales a su vez pueden

ser subdivididas hasta alcanzar el nivel de operaciones elementales (asociadas a un sólo componente y que no requieren división en nuevas operaciones).

- ❒ Abstracción Semántica: La relación entre componentes y operaciones permite la agrupación a diferentes niveles de abstracción de componentes en base a la semántica de las operaciones que proporcionan. Por ejemplo, las operaciones de lectura, escritura y actualización de datos, se pueden agrupar en operaciones de acceso a datos.

Adicionalmente, la arquitectura MOIRAE soporta componentes con las siguientes funcionalidades:

- ❒ Movilidad: Componentes que puedan migrar entre diferentes localizaciones dentro del entorno.
- ❒ Actualización Dinámica: Se pueden añadir o eliminar nuevos componentes del sistema, incluso durante la ejecución del mismo, siempre y cuando se cumplan en todo momento las consideraciones antes citadas.
- ❒ Independencia de la arquitectura: Permite que la implementación de sus componentes se realice sobre diferentes arquitecturas y en diferentes lenguajes de programación. Esta funcionalidad ha de encontrar un punto de equilibrio entre el uso de diferentes plataformas y las restricciones de movilidad.

5.2 Modelo de Componente

De los cuatro modelos que conforman la arquitectura MOIRAE, el primero de ellos es el **Modelo de Componente**. Este modelo describe cuales son las características de un componente tanto desde un punto de vista externo como interno. Este modelo también define la descomposición del componente en diferentes módulos los cuales realizan una serie de acciones determinadas. La estructura descrita en el modelo representa el esqueleto de cada una de las piezas que forman parte de un sistema tal y como se requiere por la arquitectura. Interiormente se encuentran desarrolladas las funcionalidades propias de cada componente.

En la figura 5.1 se muestran los elementos que conforman cada componente de la arquitectura MOIRAE. En primer lugar existe una división dentro de cada componente en dos grupos de elementos:

① Por un lado se encuentran los elementos encargados de proporcionar las funcionalidades necesarias para que el componente sea capaz de realizar las tareas que se requiere que desempeñe. De esta forma, por ejemplo, si se considera un componente encargado de cargar

tablas de la base de datos en ciertas áreas de memoria este componente, dentro de este primer conjunto de elementos tendrían los módulos que proporcionan funcionalidades como, localización de las bases de datos que contienen la información, funciones de acceso a tablas de dichas bases de datos, reserva de regiones de memoria compartidas entre componentes, etc. A todo este conjunto de funcionalidades que, para un componente concreto, consiguen que sea capaz de proporcionar el servicio que de él se espera se le denomina **plano operacional** del componente.

El plano operacional debe de ser capaz de satisfacer todas las operaciones ofertadas por el componente al resto de entidades del sistema así como a los entidades externas del sistema que lo utilizan.

② Por otro lado se encuentran los elementos del componente encargados de las tareas de control y gestión del mismo. Las funciones de estos elementos dirigen las tareas realizadas por los elementos del plano operacional. Este conjunto de funcionalidades conforman el **plano de control** del componente. Retomando el ejemplo del componente encargado de la carga de datos en regiones de memoria compartida, el plano de control dispondrá de las reglas que rigen, por ejemplo, el tamaño máximo de bloque de memoria reservado, el número máximo de bloques reservables, los criterios de descarte y renovación de bloques, por nombrar algunas.

El plano de control es el encargado de gestionar, aplicar y evaluar las políticas que definen secuencias de acciones de control sobre los propios componentes.

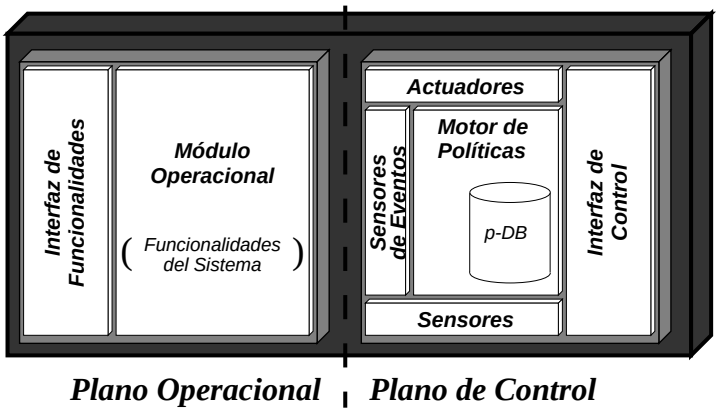


Figura 5.1: Esqueleto de los componentes de la arquitectura MOIRAE

5.2.1 Plano Operacional

Esta parte del componente encierra los elementos necesarios para que el componente sea capaz de desarrollar su trabajo. Su elementos principales son:

5.2.1.1 Interfaz de Funcionalidades

Este elemento es el que presenta los servicios que este componente proporciona al resto de componentes del sistema. En este interfaz se recogen las definiciones de las operaciones proporcionadas por el componente al exterior. Estas operaciones externas se denominan *servicios*. Los servicios se encuentran definidos en la **clase de componente**. Para definir los servicios se ha definido la siguiente notación[1]:

COMMAND SERVICES:

```
services
{
      domain   service_name([type domain]...)
               [throw event_name(domain,...)
               ...];
      ...
}
type: in|out|inout
```

Un ejemplo de la definición de estos servicios sería:

```
// DOMAINS
domaindef Message as string;
// SERVICES
services
{
  void store_message(in Message)
       throw NoMoreSpace();
  void recover_message(out Message)
       throw NoMoreMessages(),
             CorruptedMessage(Message);
  boolean has_more_messages();
}
```

5.2.1.2 Módulo Operacional

En este módulo es donde se integran todos los elementos que realizan las funciones proporcionadas por el interfaz anterior. El diseño de este módulo, evidentemente, depende las funcionalidades proporcionadas por el componente. Este módulo corresponde no a la clase de componente

[1] La sintaxis de toda la notación que se ha usado en la definición de los componentes se encuentra recopilada en el Anexo B.

como el anterior sino al código de componente, pues contiene la implementación de las operaciones definidas anteriormente.

Los elementos de este módulo pueden requerir los servicios y funcionalidades de otros componentes, por eso puede hacer uso del interfaz de funcionalidades de dichos componentes por medio del mecanismo de comunicación adecuado.

5.2.2 Plano de Control

A este nivel, como ya se ha dicho, se encuentran los elementos encargados de seleccionar y aplicar las decisiones de control del componente. En su conjunto, este plano se corresponde con un agente, que controla el funcionamiento del plano operacional. Este agente de control es capaz de monitorizar el estado del componente, alterarlo e interactuar con otros agentes de control para cooperar en la resolución de problemas. Este plano está dividido en:

5.2.2.1 Sensores

Este módulo está compuesto por un conjunto de **primitivas**, definidas en la clase de componente e implementadas en el código de componente. Estas primitivas permiten examinar el estado de los elementos del plano operacional. Por medio de los sensores debe ser posible obtener la información de todos los parámetros actuales del sistema que se consideran relevantes, de forma que se puede definir **estado de un componente** a los valores de dichas primitivas en un instante determinado.

Las primitivas representan la descripción de los posibles predicados de estado de un componente. Ejemplos de posibles primitivas son: `ocupado(`n`)`, `bloqueado()`, `contenido(`pos`,`$valor$`)` o `usuario(`$nombre$`,`$permisos$`,`$prioridad$`)`.

La sintaxis adoptada para la definición de las primitivas es:

SENSOR PRIMITIVES:

```
primitives
{
      primitive_name([domain]...);
      ...
}
```

Un ejemplo de una definición de primitivas sería:

```
// DOMAINS
domaindef Status       as {BUSY,RELEASED};
```

```
domaindef EntryNumber  as [1..10];
domaindef EntryNumbers as set<EntryNumber>;
// PRIMITIVES
primitives
{
  status(Status);
  candidate_entries(Status,EntryNumbers);
}
```

Como resulta evidente, el diseño de el componente (y de sus primitivas) está supeditado a las características y módulos del plano operacional.

Como un tipo especial de primitivas de sensores se encuentran los sensores volátiles. Estos sensores se diferencian de los anteriores en que su valor puede cambiar incluso cuando no se realizan operaciones sobre el componente. El resto de sensores sólo alteran sus valores en base a operaciones internas del componente o a servicios solicitados de forma externa al mismo. Este tipo especial de sensores se expresan utilizando la palabra `volatil`. Por ejemplo:

```
primitives
{
  volatil cpu_load(Pctg);
}
```

Los sensores volátiles, por lo general, están asociados a factores externos al componente, pero que pueden ser detectados por el mismo, aunque en muy raras ocasiones pueden ser alterados voluntariamente por el componente. La carga de la CPU vista en el ejemplo anterior o la accesibilidad a otro nodo o componente son los ejemplos más claros.

5.2.2.2 Actuadores

Dentro de este módulo se engloban un segundo grupo de operaciones del componente denominadas **comandos**. Estos comandos permiten alterar ciertos valores que rigen el comportamiento del sistema, así como intervenir directamente en ciertos recursos del mismo. Una serie típica de comandos podrían ser los encargados de gestionar los *buffers* internos de un componente determinado, existiendo comandos para añadir, borrar, vaciar, cargar o salvar dichos *buffers*. En resumen, los comandos de los actuadores, permiten que el plano de control intervenga en el funcionamiento del plano operacional.

Los comandos representan un segundo tipo de operaciones del componente, que a diferencia de las definidas en el interfaz de funcionalidades del plano operacional, sólo suelen ser invocadas como resultado de la aplicación de políticas, formando parte de las acciones de control.

El formato de un comando es el siguiente:

ACTOR COMMANDS:

```
commands
{
       domain   command_name([type domain]...)
                  [throw event_name(domain,...)
                                          ...];
       ...
}
type: in|out|inout
```

y un ejemplo de la definición de unos comandos sería:

```
// DOMAINS
domaindef string as BufferDataRef;
// COMMANDS
commands
{
  EntryNumber add_buffer()
       throw NoBufferSpace();
  void        delete_buffer(in EntryNumber)
       throw WrongBufferEntry(EntryNmber),
       throw BufferEntryInUse(EntryNmber);
  void       flush_buffer(in EntryNumber)
       throw WrongBufferEntry(EntryNmber),
       throw BufferEntryInUse(EntryNmber);
  void       load_buffer(in EntryNumber, out BufferDataRef)
       throw WrongBufferEntry(EntryNmber),
       throw BufferEntryInUse(EntryNmber);
  void       save_buffer(in EntryNumber, in BufferDataRef)
       throw WrongBufferEntry(EntryNmber),
       throw BufferEntryInUse(EntryNmber);
}
```

5.2.2.3 Sensores de Eventos

Este elemento es el encargado de recoger las situaciones relevantes producidas en el plano operacional que requieren la intervención del plano de control para resolverse. De esta forma, por ejemplo si llega una petición al componente encargado de cargar las tablas en memoria con unos determinados parámetros que no es posible ejecutar de forma directa, automáticamente se produce una notificación que recoge este módulo. Esta notificación se denomina **evento**. Al capturarse cada evento, se programa en el plano de control la secuencia de **comandos** de los actuadores que hay que realizar, dependiendo del estado del componente (que se analiza por medio de las **primitivas** de los sensores).

Los eventos se disparan por la ejecución tanto de servicios como de comandos del componente. En la definición de la clase de un componente se especifica qué operaciones son las que disparan

cada uno de los eventos declarados.

Para dotar de una mayor flexibilidad a los eventos existen una serie de argumentos de entrada para cada uno de los mismos, que representan los parámetros bajo los cuales se ha producido dicho evento. La definición de los eventos se realiza mediante la notación:

EVENTS:

```
events
{
       event_name([domain]...);
  ...
}
```

y un ejemplo de eventos definidos de esta forma sería:

```
// DOMAINS
domaindef set<Component> as Components;
// EVENTS
events
{
  BufferAccessCollision(EntryNumber,Components);
  BufferRequest(BufferDataRef);
  ComponentShutDown();
  ComponentStart();
  ComponentReset();
}
```

Ciertos eventos pueden surgir de acciones propias del componente, tales como el arranque o la parada del mismo o de acciones realizadas de forma periódica (con vistas a tareas de auditoría o *tuning* del sistema).

5.2.2.4 Motor de Políticas

Este elemento constituye el centro del plano de control del componente. Aunque se encuentra detallado más adelante (Modelo de Control, Sección 5.5) es importante destacar que este módulo es el encargado de programar las acciones necesarias para resolver conflictos o situaciones presentes en el plano operacional. La activación del Motor de Políticas se realiza, por ejemplo cuando el sensor de eventos captura una situación notificada por algún elemento del plano operacional. Este evento (junto con los argumentos asociados al mismo) se transmite al Motor, el cual aplica las reglas que tiene almacenadas en la base de datos de políticas (**p–DB**) obteniendo los comandos a ejecutar que en conjunto representan la respuesta del motor al evento producido. Al igual que los eventos, existen otros mecanismos mediante los cuales se requiere la intervención del Motor.

En ciertas circunstancias, el Motor de Políticas no es capaz de proporcionar una secuencia de comandos que resuelva el problema que se le plantea, en estos casos, se produce lo que se denomina **propagación del control**, es decir que el Motor de Políticas local transmite a otro Motor de Políticas el estado en el que se ha quedado en la resolución de una petición, siendo ahora ese otro Motor el responsable de solucionar el problema. Este proceso de comunicación se realiza por medio del interfaz de control.

El formato en el cual se expresan las reglas de las políticas, así como el proceso de evaluación de las reglas y construcción de la secuencia de operaciones se encuentran definidas en el Modelo de Control (Sección 5.3).

5.2.2.5 Interfaz de Control

Este último elemento de la parte de control es el encargado de proporcionar los mecanismos de comunicación entre diferentes planos de control de distintos componentes. Estos planos pueden requerir comunicarse entre si para realizar:

① **Gestión distribuida de políticas**: Mecanismo usado por los distintos componentes del sistema para manipular el conjunto de políticas y solicitar su aplicación de forma cooperativa. Utilizado, por ejemplo, cuando un Motor de Políticas no es capaz de resolver un conflicto, cuando un Motor de Políticas (por lo general de ámbito mayor), transmite a otro un bloque de políticas para que éste último las aplique en la resolución local de conflictos, etc.

② **Acceso remoto a los sensores**: Usado cuando un Motor de Políticas desea consultar los sensores de otro Motor.

③ **Acceso remoto a los actuadores**: Análogo al caso anterior para el uso de los comandos del módulo de actuadores.

Los mecanismos según los cuales se aplican las políticas y como se interconectan por medio de este módulo los distintos planos de control se encuentran tratados con más profundidad en el Modelo de Control (Sección 5.5).

5.3 Modelo de Relación

Los componentes del entorno definidos en el modelo anterior no están pensados para funcionar de forma aislada. Estos componentes al formar parte de un sistema pueden estar conectados entre sí por medio de relaciones. Estas relaciones se definen como:

Definición RELACIONES ENTRE COMPONENTES: Es un tipo de dependencia entre dos componentes del entorno mediante la cual uno de ellos o incluso los dos son capaces de enviar solicitudes al otro. Por medio de estas solicitudes se requiere la aplicación de servicios del segundo a petición del primero.

Independientemente del mecanismo de comunicación utilizado para permitir las solicitudes remotas de un componente a otro, se asume que el proceso de comunicación entre componentes requiere tres factores:

- Mecanismo de comunicación sobre el que se define la arquitectura,
- Información que permita localizar al componente dentro del entorno e
- Información relativa a los servicios proporcionados por el componente.

De esta forma, una relación no es más que la representación de dicha información (de localización y servicios disponibles) de forma directa o por medio de un paso de resolución intermedio (de forma indirecta). El formato de la información de localización depende del mecanismo de comunicación y la información referente a los servicios del componente se define por medio de la clase del componente asociada. En conjunto, ambas informaciones representan la referencia remota de un componente. Una vez localizado un componente por medio de esta información, la tecnología de comunicación debe posibilitar posteriormente la solicitud de servicios. Esta faceta de comunicación entre los componentes no es pues una responsabilidad de la arquitectura sino de la tecnología sobre la que se asiente.

Otra característica importante de las relaciones entre componentes es la navegabilidad, que representa, para cada par de componentes relacionados, cual de ellos puede solicitar servicios al otro. Una relación *unidireccional* implica que sólo uno de los componentes tiene información sobre el otro componente y por lo tanto es el único que puede solicitar servicios. En las relaciones *bidireccionales* ambos componentes tienen información para localizar al otro componente. Una relación bidireccional representa dos relaciones unidireccionales una en cada sentido.

Este Modelo clasifica las relaciones entre dos componentes del sistema en base a tres criterios diferentes. Por un lado, se puede establecer una distinción basada en el ámbito en el que es visible la relación entre dos componentes (relaciones *públicas* y *privadas*). Por otro lado, se puede definir una segunda división de las relaciones según el conocimiento que un componente tiene de su contraparte al otro extremo de la relación, definiendo de esta forma relaciones *identificadas* y relaciones *anónimas*. Finalmente, las relaciones pueden clasificarse según su cardinalidad, desde relaciones *simples* con un sólo elemento en cada extremo de la relación, hasta relaciones *múltiples*

con varios elementos en alguno de los extremos de la misma. Todas estas tipificaciones de las relaciones se encuentran expresadas a continuación.

5.3.1 Relaciones según su Visibilidad

Uno de los criterios de clasificación de las relaciones es su accesibilidad, tanto para consultar como para alterar propiedades de la misma. En base a este criterio se definen relaciones *privadas* y relaciones *públicas*.

5.3.1.1 Relaciones Privadas

Las relaciones privadas (o basadas en referencias internas) se establecen cuando uno de los componentes mantiene una referencia a otro componente remoto como una información privada a la parte operacional del mismo. El formato o tipo de esta referencia dependerá del soporte de comunicación que exista entre los dos componentes. La información asociada a este tipo de relaciones se caracteriza por ser accesible únicamente desde el plano operacional del componente que posee la referencia y sólo internamente dentro de esta parte operacional. Dicha información sólo puede ser consultada y alterada por medio de servicios que el propio componente disponga y ni siquiera el plano de control del propio componente es capaz de consultar ni modificar dicha relación.

La información asociada a las referencias privadas, únicamente tiene que indicar la navegabilidad de la relación, es decir qué componente puede solicitar los servicios de otro componente del cual mantiene la información para referenciarlo. Estas relaciones pueden ser tanto navegables en una única dirección, como bidireccionales (si ambos componentes mantienen una referencia al otro). Al tratarse de una información interna del plano operacional del componente es responsabilidad del propio componente la gestión de dicha referencia. Esto hace que el resto de características de la relación sean gestionadas y aplicadas por el propio componente y por tanto no han de ser indicadas expresamente, pues su desarrollo corresponde a la fase de diseño e implementación del sistema.

La única posibilidad que existe para que este valor sea alterado en base a una decisión externa es por medio de una o varias funciones proporcionadas por el componente que permitan actualizar su estado. En esos casos, lo más adecuado es optar por otro tipo de relación. Originalmente estas relaciones son apropiadas para interacciones entre componentes que sean fijas a lo largo de la vida (periodo de ejecución) del componente.

5.3.1.2 Relaciones Públicas

Las relaciones públicas (o basadas en referencias públicas) se establecen entre dos componentes del entorno por medio de unos mecanismos que permitan conocer, de forma externa, los componentes que están participando en ella. Esta clase de relación permite que un componente externo acceda a la relación pudiendo realizar tareas tales como la navegación por las relaciones establecidas para alcanzar componentes interrelacionados entre sí. Otra ventaja de estas relaciones es que pueden ser modificadas por un componente externo (un componente gestor, por ejemplo) que modifique uno de los participantes en la misma sin que ninguno de los dos lo noten.

Las referencias públicas, al igual que las anteriores definen una navegabilidad entre los componentes relacionados, pero en este caso es necesario proporcionar un identificador de la relación. Este identificador debe ser conocido por los terceros elementos que desean interactuar con dicha relación y desde el punto de vista de aplicación debe permitir localizar dicha relación dentro del entorno distribuido. Puede proporcionarse, de forma adicional, otra información relativa a la relación, para ser consultada y alterada tanto por los elementos externos que gestionan la relación, como por los propios participantes en la misma.

Un ejemplo de este tipo de relación puede ser la interacción entre un componente de un sistema que interactúa con el usuario y un componente interno que procesa dichas peticiones, en este caso existe una relación definida entre dos roles (interfaz y proceso, por ejemplo). Dichos roles pueden encarnarse en los dos componentes antes citados, pero también pueden existir componentes de otro tipo que pueden asumir alguno de los roles y por lo tanto puedan formar parte de dicha relación. Activándose un segundo componente de proceso y restableciendo la relación entre el componente de interfaz y este nuevo componente se puede redistribuir la carga de un sistema complejo sin que los componentes tengan que realizar dicha calibración.

5.3.2 Relaciones según Identificación

Un segundo criterio de clasificación de las relaciones es el establecido en base al conocimiento que cada elemento de la relación tiene de la identidad de los elementos con los que está relacionado. Por medio de este criterio se definen relaciones *identificadas* y relaciones *anónimas*.

5.3.2.1 Relaciones Identificadas

Las relaciones identificadas también se denominan relaciones basadas en solicitudes directas. En este tipo de relaciones cada uno de los elementos puede solicitar directamente al otro elemento de la relación una petición de forma directa. Para ello cada uno de los componentes conoce cuál

es el otro componente de la relación. Igual que en el caso de las relaciones privadas, la única información necesaria es la navegabilidad del enlace entre cada pareja de componentes.

5.3.2.2 Relaciones Anónimas

Mediante este tipo de relaciones (denominadas también relaciones basadas en notificaciones) dos componentes pueden interactuar por medio de la notificación, de forma asíncrona, de sucesos o peticiones que afecten a ambos. De esta forma, un elemento de la arquitectura puede solicitar a otro componente que se procese una petición. Este mecanismo se diferencia del anterior en que proporciona: (a) un esquema de atención de peticiones basado en colas de espera[2] y (b) una completa independencia entre el solicitante de la petición y el componente que ha de servirla, de forma que el conocimiento del número de ellos o la identificación de cada uno no es un requisito necesario para el funcionamiento de este esquema.

Las relaciones anónimas son unidireccionales únicamente. Se basan en la definición de un canal de eventos o notificaciones identificado por medio de un nombre. A este canal se suscriben los distintos componentes como emisores o receptores de notificaciones. El proceso de creación de nuevos canales y la asignación a los mismos de emisores y receptores puede realizarse desde componentes externos o desde los propios elementos que intervienen en la relación. La definición de relaciones bidireccionales de estas características de realiza por medio de la combinación de dos relaciones de esta clase unidireccionales. El nombre del canal de eventos, en estas relaciones, es el dato fundamental pues actúa como referencia del elemento intermedio necesario para interconectar emisor y receptor de mensajes.

Todo esto en conjunto, permite que un componente que pueda requerir una serie de servicios de otros componentes se subscriba a los canales de notificaciones que son escuchados por dichos objetos de servicio. Asimismo, si se desea activar un componente que proporcione un determinado servicio entonces dicho componente se registrará como receptor del canal de eventos antes indicado. En resumen, este mecanismo permite que el conjunto de componentes que solicitan o proporcionan un servicio puede alterarse a lo largo de la ejecución del sistema.

Un caso de aplicación de este tipo de relación el tratamiento de mensaje internos de un sistema distribuido. El sistema puede definir un canal de eventos para estos mensajes, de forma que todos los elementos del sistema se registran como emisores (todos puede generar un mensaje) y un conjunto pequeño de elementos puede definirse como receptores, siendo de diferentes tipos, desde el

[2] aunque en las alternativas anteriores el *middleware* de comunicaciones hacía uso de colas de peticiones para gestionar las solicitudes entre componentes, en este caso es el propio componente el que es capaz de definir el número y prioridad (o peso) de las colas de solicitudes

componente que registra todos los mensajes en un fichero, hasta el que analizando los mensajes de mayor prioridad manda un *mail* a una determinada dirección.

5.3.3 Relaciones según su Cardinalidad

El último factor para la clasificación de las relaciones se corresponde con el número de componentes que aparecen en cada uno de los extremos de una relación.

5.3.3.1 Relaciones Simples

La cardinalidad de dicha relación es 1 a 1, es decir que en cada extremo de la relación sólo existe un componente.

5.3.3.2 Relaciones Múltiples

El otro tipo de relaciones son las que tienen más de un componente en alguno de los extremos de la relación (1 a N), o incluso a ambos lados (N a M).

Este tipo de relaciones llevan asociadas una serie de consideraciones adicionales:

- La solicitud de un servicio a un conjunto de componentes que se sitúan al otro extremo de una relación, puede realizarse de forma que sea invocada para todos ellos. Esto implica que el solicitante del servicio recibe no una contestación, sino tantas como componentes realicen el servicio. Cada una de estas respuestas está compuesta por un vector de todos los argumentos de salida del servicio, así como de los valores de retorno de los servicios.
- Una segunda modalidad de solicitud de servicio a varios componentes, es la selección de únicamente uno de ellos como ejecutor del servicio. La forma en la cual se determina dicho componente de entre el conjunto de componentes posibles no se encuentra restringido por el modelo, permitiendo que dicha selección se realice por parte del solicitante, de forma aleatoria o dinámicamente configurable por medio de decisiones de control (el que menos carga tenga, el más próximo, ...).

5.3.4 Recapitulación sobre Relaciones

Estos seis tipos de relaciones pueden combinarse en base a los criterios que los definen. De esta forma, por ejemplo, pueden existir relaciones públicas anónimas simples o privadas identificadas múltiples. La Tabla 5.1 presenta las cuatro combinaciones posibles, descartando las combinaciones basadas en la cardinalidad (simples y múltiples). En esta tabla también se muestra la información asociada a cada tipo de relación.

	Pública	Privada
Identificada	Navegabilidad, identificación de la relación y otra información relativa a la relación.	Navegabilidad.
Anónima	Navegabilidad, identificación del canal de eventos, identificación de la relación y otra información relativa a la relación o al canal de eventos.	Navegabilidad, identificación del canal de eventos.

Tabla 5.1: Tipos de relaciones entre componentes.

En el caso de las relaciones públicas anónimas, y dependiendo de la implementación final, el identificador de canal puede coincidir con el identificador de la relación. De todas formas, a efectos formales, se considera que ambos identificadores corresponden a dos dominios diferentes y por lo tanto son distintos.

A la hora de especificar las relaciones de un componente en su definición, se utiliza la expresión:

COMPONENT RELATIONSHIP:

```
relations
{
      vis component_type id: {ident} {card}
      ...
}
 vis:    public|private
 ident:  identified|anonymous
 card:   simple|multiple
```

Un ejemplo de definición de varias relaciones sería:

```
// RELATIONS
relations
{
  private BufferCmpnt buffer:     identified simple;
  private BufferCmpnt aux_buffer: identified simple;
  public  ClientCmpnt clients:               multiple;
}
```

5.4 Modelo de Arquitectura

El tercero de los modelos que componen MOIRAE es el Modelo de Arquitectura que se basa en los modelos anteriores para indicar cómo colaboran los componentes en la realización de las tareas asignadas al sistema completo. Este modelo indica cómo se conforma un sistema distribuido que use los componentes descritos en el Modelo de Componente y que entre ellos interactúen por medio de las relaciones indicadas por el Modelo de Relación. Para ello, se define una arquitectura de referencia genérica sobre la que se matizan una serie de características:

- Se ha intentado que esta arquitectura sea lo menos restrictiva posible desde el punto de vista de su realización. Para ello su diseño y posterior implementación puede ser abordado por medio de distintas tecnologías y mecanismos.

- Un segundo objetivo es hacer que la aplicación de esta arquitectura a diferentes problemas sea posible sin alterar los conceptos propuestos en este Modelo, evitando de esta forma restringir la aplicabilidad para la construcción de sistemas distribuidos en otros dominios.

Estas dos facetas de la independencia garantizan la generalidad de la solución y la ausencia de ligaduras que hagan que los conceptos generales de la misma dependan, por ejemplo de la continuidad de tecnología de *middleware* concreta.

La arquitectura propuesta se puede presentar como un entorno distribuido de componentes (descritos como se ha visto anteriormente) que interactúan entre sí. Esta interacción se basa en la utilización de las funcionalidades proporcionadas por cada uno de los componentes y que son usados por el resto de elementos del sistema, resultando en conjunto de elementos capaz de responder a una serie de servicios generales proporcionados por el sistema en conjunto. Sin embargo, la interacción, debido a la propia estructura de los componentes, se puede realizar de dos formas diferentes:

1. **Interacción operacional**: Cuando dos componentes del sistema se encuentran interconectados por medio de su plano operacional. La interacción a este nivel proporciona la posibilidad de que uno de los componentes solicite al otro alguno de los servicios proporcionados por su plano operacional. Por ejemplo, un elemento conectado con un componente que actúe como compilador de una determinada gramática puede solicitar a éste la compilación de una cadena de caracteres concreta o el procesamiento de las sentencias almacenadas en un *buffer*; si existe un componente que actúe como una caché de cierto tipo de información, las operaciones que se le pueden solicitar es que almacene un nuevo bloque de información o que dé de baja uno ya existente.

② **Interacción de control**: Esta relación se establece cuando una serie de componentes trabajan a nivel de su plano operacional en un conjunto de tareas que presentar una interdependencia entre ellas, de forma que cualquier situación anómala o simplemente relevante que se dé en uno de ellos puede llegar a repercutir en los demás. Por ejemplo, a este nivel deberían encontrarse interconectados de alguna manera los elementos que conforman un subsistema de acceso a datos, considerando que dichos componentes pueden ser: espacios de almacenamiento intermedio, adaptadores de acceso a ciertos soportes de almacenamiento, elementos de acceso y preparación de consultas, *buffers* de *redo* o *rollback* (para control de transacciones y consistencia). En este ejemplo, una saturación del componente que gestiona el *buffer* de *rollback* repercute en todo el subsistema.

5.4.1 Grafos de Interacción

De la misma manera que a nivel de la descomposición de un componente atómico del sistema se definen dos planos de funciones (operacional y de control), a nivel de arquitectura deben existir relaciones entre componentes de esas dos mismas características. Esto hace que existan dos grafos de interacción dentro de cada sistema definido acorde a esta arquitectura. Cada uno de estos grafos plasma los componentes que se relacionan entre sí a cada uno de los niveles. La existencia de una relación entre dos componentes en un determinado plano no implica la necesidad de que exista ningún tipo de conexión entre ambos elementos en el otro plano. Esto hace que los dos planos se mantengan independientes entre sí, siguiendo un diseño diferente, en tanto en cuanto, las necesidades y la forma de resolverlas pueden ser diferentes.

Como formalismo para denotar estos grafos de relación se puede usar tanto notación UML [SMHP+97a, SMHP+97b] como cualquier otro tipo de representación. La representación elegida deber permitir indicar para cada una de las relaciones que se establezcan entre dos componentes la información asociada a la misma. Esta información dependerá de qué se trate de un grafo del plano operacional o del nivel de control, así como del tipo de relación.

5.4.1.1 Grafos Operacionales

Los grafos operacionales representan las relaciones existentes entre los componentes que se encuentran establecidas en base a su plano operacional. La arquitectura de referencia propuesta no define ninguna restricción o consideración a nivel de las relaciones de este plano. Cualquiera de los tipos de relaciones definidos anteriormente puede interconectar los componentes por medio de su plano operacional siguiendo cualquier topología o distribución.

Al no definir ningún conjunto de dependencias prestablecidas, la arquitectura genérica es adaptable a varios entornos de aplicación. En resumen, la generalidad de la arquitectura está garantizada al no indicar que la resolución de las operaciones (definida en el plano operacional) se tenga que ajustar a un patrón determinado. De esta forma, se puede diseñar un sistema distribuido siguiendo la arquitectura de referencia (que está dirigida principalmente hacia el plano de control) en base a la interconexión de los componentes según convenga en la fase de diseño.

5.4.1.2 Grafos de Control

A diferencia del plano operacional, en el plano de control se han definido unas pautas de organización de los componentes determinadas. Estas consideraciones de interconexión de componentes actúan como base para garantizar los objetivos de la arquitectura de control.

De igual manera que los planos operacionales de los componentes se encuentran interconectados para resolver las funciones de trabajo del sistema, los planos de control de varios componentes también se encuentran comunicados, con el objeto de, en conjunto, resolver los conflictos de control del sistema. Teniendo esto en cuenta, la organización de los componentes en base al plano de control se va a realizar conformando una estructura jerárquica. Esta organización permitirá que las decisiones de control se tomen a diferentes niveles, dependiendo de la transcendencia y ámbito de las mismas.

Evidentemente, las tareas de control no siempre son posibles de determinar usando la información local de un componente. Estos casos se dan cuando los criterios de decisión están parametrizados por características externas al componente (por lo general definidas en otros componentes). Por ejemplo, la saturación de una serie de componentes del sistema puede ser debida al exceso de componentes de otro tipo que solicitan ciertos servicios a los primeros y la solución puede pasar por que un tercer tipo de componentes regulen el flujo de peticiones.

El modelo de organización planteado para este problema es jerárquico. Esto permite establecer diferentes ámbitos de control donde una serie de componentes tienen responsabilidades de control sobre los componentes situados por debajo en la jerarquía. En esta estructura los elementos situados en la base son, en la gran mayoría de los casos, componentes con un plano operacional complejo y generalmente con un importante consumo de recursos (memoria, proceso, ...). A estos componentes se los denomina **componentes orientados a operaciones** y sus funciones de control están por lo general enfocadas a la gestión de recursos local. Por otro lado, existen una serie de componentes que se han denominado **componentes orientados a control**. Cada uno de estos otros elementos se encuentran por encima de varios de los anteriores y actúan como coordinado-

res de las tareas de control que afectan a más de uno de ellos, por lo tanto sus funcionalidades del plano de control son superiores a las del plano operacional.

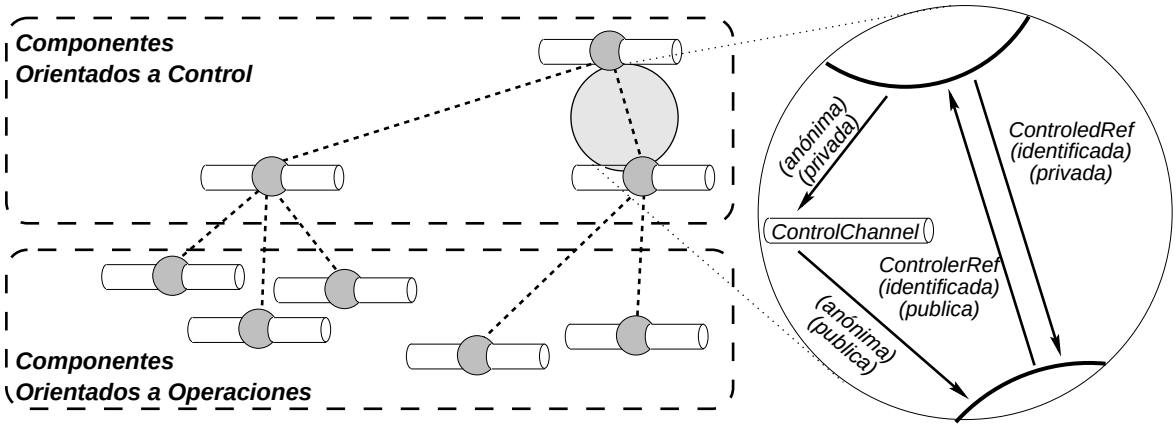


Figura 5.2: Relaciones entre los elementos de control

La comunicación que se ha de establecer entre los distintos niveles de la jerarquía determina, hasta cierto punto, el tipo de relación que se puede definir entre ellos. En el diseño de la arquitectura se distinguen dos grupos de funcionalidades diferentes, las de los elementos controladores, situados jerárquicamente por encima y los elementos controlados, por debajo de éstos. Como se puede observar un mismo componente a nivel de plano de control puede asumir ambos papeles, uno en relación de los elementos que se encuentren por debajo en la jerarquía y otro con los que están por encima de este. En la Figura 5.2 se observa, por cada pareja de elementos controlado/controlador se definen los siguientes criterios de relación:

- Una relación identificada-pública unidireccional, mediante la cual el elemento controlado realiza solicitudes al controlador. Al tratarse de una relación pública, es posible asociar el elemento controlado a otro controlador sin que este primero lo perciba.

- Una relación identificada-privada, también unidireccional, entre ambos elementos que permite que el controlador envíe comandos a los componentes que controla. En este caso la relación no es pública para evitar inconsistencias debidas a que un tercer elemento altere los elementos que son controlados por un determinado componente, puesto que existen ciertas operaciones en las cuales el componente de control almacena información de estado en relación a los componentes que controla.

- Finalmente, una tercera relación entre elementos controlados y controladores. Esta relación está definida sobre un canal de eventos (denominado *Canal de Control*). Por medio de este canal el controlador transmite órdenes a todos los componentes que controla. Existe un

canal de este tipo por cada componente controlador y a este canal se asocian los elementos controlados.

5.4.1.3 Componente *Seed* o Semilla

Un componente especial de la arquitectura MOIRAE es el componente *Seed*. Dicho componente se encuentra situado en el punto más alto de la jerarquía de componentes según el grafo de control. Las responsabilidades de dicho componente son las siguientes:

- ❒ Se trata de un componente con muy pocas funcionalidades operacionales y centrado básicamente en tareas de control.
- ❒ Su principal función es iniciar el resto de elementos del sistema. Esto lo puede realizar directamente o puede únicamente encargarse de arrancar un componente de control por cada subsistema, siendo cada uno de estos los responsables de iniciar el resto de componentes.
- ❒ Almacena y gestiona la base de datos de políticas de todo el sistema. *Todas* las políticas de *todos* los componentes del sistema se encuentran almacenadas aquí. De esta forma por medio del mecanismo de delegación de control que se verá posteriormente se alimentan las bases de datos de políticas del resto de componentes del sistema.
- ❒ Representa el punto de acceso de nuevos componentes que desean unirse al sistema en tiempo de ejecución. Estos nuevos componentes solicitan al *Seed* su inclusión en el sistema y éste los dirige al nivel de la jerarquía apropiado.

5.4.2 Nodos de Despliegue

Otro concepto importante dentro de la arquitectura genérica propuesta, es el concepto de **Nodo de Despliegue**.

> **Definición** NODOS DE DESPLIEGUE: Computadoras conectadas entre sí, que tienen la capacidad de albergar a componentes de un sistema distribuido.

Cada uno de estos nodos es capaz de ejecutar un componente distribuido del sistema. No es necesario que todos los nodos sean capaces de ejecutar todos los componentes, y por lo tanto, que ciertos componentes sólo estén disponibles en unos determinados nodos.

Una faceta muy importante de estos nodos es que la información relativa a los componentes disponibles en cada nodo debe de encontrase accesible, bien de forma distribuida o centralizada en un único repositorio. La forma de acceder a dicha información no debe restringirse a un

identificador único por cada tipo de componente, sino que debe de ser posible realizar consultas en base a otros datos del componente, tales como los servicios o funcionalidades que proporciona. Esto puede realizarse por medio de los distintos mecanismos que proporcione el soporte de comunicaciones *middleware*.

5.4.2.1 Componente de Gestión de Nodo

Para poder hacer uso de un nodo de despliegue la arquitectura requiere que exista algún proceso en dicho nodo capaz de arrancar otros procesos o componentes a petición de los elementos del sistema. Esta tarea se realiza por medio de una clase de componentes denominados **componentes de gestión de nodo**. Estos componentes proporcionan información del nodo en relación a los recursos disponibles del mismo y permiten que otros componentes puedan solicitar la ejecución de un nuevo componente en dicho nodo.

Sin la existencia de estos componentes de gestión no es posible utilizar los nodos de despliegue y su activación en cada nodo debe realizarse por medio de mecanismos ajenos a la arquitectura (ejecución de algún proceso en dicho nodo de forma remota o algún tipo de servicio de red de dicho nodo).

5.4.2.2 Base de Datos de Nodos de Despliegue

La información relativa a los nodos de despliegue disponibles en el sistema se encuentra almacenada en un repositorio de información único. Dicho repositorio puede encontrarse centralizado o distribuido entre varias localizaciones, siempre y cuando mantenga la consistencia de la información que almacena.

Esta información se usa a la hora de iniciar la ejecución de un nuevo componente. La información está compuesta por datos relativos a:

- Los nodos de despliegue disponibles: Recursos, identificación y características de los mismos. Dicha información se transmitirá por los componentes de gestión del nodo en el momento de ser arrancado y si con cierta periodicidad no se recibe un refresco de dicho componente entonces se considera que el nodo se encuentra dado de baja.
- Información sobre las clases de componentes: Definición de los servicios, comandos, etc. tal y como se encuentran descritos en el Modelo de Componente (Sección 5.2). La gestión de esta información se puede basar en ciertos repositorios de la tecnología de comunicaciones sobre la que se implemente la arquitectura.
- Información sobre las implementaciones de componentes: Relación entre clases de compo-

nentes y nodos de despliegue. Para poder ejecutar un componente en un determinado nodo resultará necesario disponer del código ejecutable de dicho componente para la arquitectura/sistema operativo del nodo, instalando los ficheros que contengan dichas implementaciones en el nodo.

Es importante recalcar que esta base de datos no almacena información en relación a dónde se encuentran ejecutando los componentes en un instante determinado. La información de esta base de datos se dirige hacia el proceso de arranque de nuevos componentes del sistema, proporcionando información útil para dicho proceso, pero no registrando qué componentes se están ejecutando y con quién están relacionados.

5.5 Modelo de Control

Uno de los factores fundamentales en el proceso de control del sistema, es la evaluación de las políticas. Esta tarea es necesaria desde el momento en el cual se da una circunstancia especial en el plano operacional y se requiere la realización de una serie de acciones para resolver dicha situación. La forma de alcanzar dicha solución está sujeta a una serie de características:

- La resolución de una de estas circunstancias implica la realización de 0 a n acciones.
- El conjunto de alternativas que solucionan cada situación es variable.
- La elección de la mejor alternativa (o una suficientemente buena) depende de multitud de factores, tanto locales como globales y siempre responde a un criterio de optimización determinado.
- Por causa de condicionamientos externos (optimización del sistema con diferentes criterios) es posible que se modifiquen las condiciones mediante las cuales se selecciona la mejor secuencia de acciones a una determinada situación.

El análisis realizado sobre el tratamiento de las decisiones de control se ha dividido en un estudio de las diferentes relaciones de control entre componentes y por otro lado en la gestión y organización de la información de control, denominada **políticas**.

5.5.1 Relación entre los Componentes de Control

Las estructura jerárquica de control definida anteriormente permite que la mayoría de las decisiones de control se resuelvan en los elementos más bajos de la jerarquía, es decir los componentes orientados a operaciones. Pero puede ser que se presenten problemas en los cuales un componente no tiene información suficiente para establecer la secuencia de acciones a realizar para resolver el problema. En estas situaciones se realiza lo que se denomina una **propagación del control**.

Definición PROPAGACIÓN DE CONTROL: Mecanismo mediante el cual un componente que no es capaz de encontrar una política de control para una situación, transmite a un elemento, jerárquicamente superior en la estructura de control, la situación que no ha podido resolver.

Por medio de este mecanismo los conflictos que se presentan en cualquier componente del sistema son tratados en dicho componente, si éste no puede solucionarlo, entonces se lo propaga al inmediatamente superior. Si este segundo componente tampoco fuese capaz de resolver el problema, entonces se propaga de nuevo. Al hacer uso de la propagación de control, se define el ámbito de control de cada componente, es decir las tareas en las cuales un componente tiene responsabilidades de control.

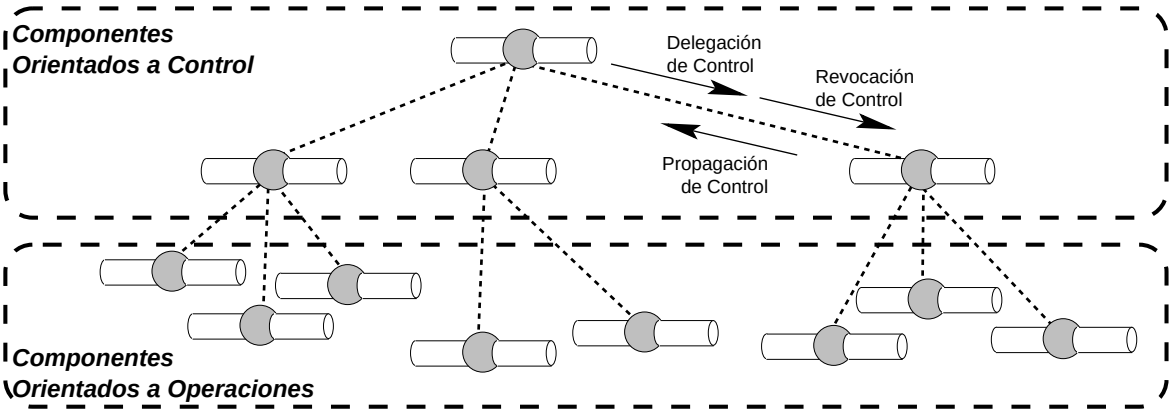


Figura 5.3: Jerarquía de elementos de control

Los beneficios alcanzados por medio de la propagación de control tienen, sin embargo una contrapartida importante a nivel de eficiencia. Esta desventaja viene apuntada por problemas de saturación de los elementos más altos de la jerarquía de control. Estos componentes pueden estar conectados con muchos elementos por debajo y en ciertos momentos (puntos de trabajo intenso del sistema) pueden tener que resolver un gran número de conflictos propagados de los elementos inferiores. Para resolver este problema existe un mecanismo análogo al de propagación de control denominado **delegación de control**.

Definición DELEGACIÓN DE CONTROL: Mecanismo por medio del cual un componente recibe de un elemento jerárquicamente superior una serie de políticas para que actualice el conjunto de las mismas que posee de forma local.

Haciendo uso de esta técnica un componente de control puede evitar su saturación delegando sus responsabilidades sobre los elementos inferiores. Los criterios mediante los cuales se decide

las situaciones en las cuales se realiza la delegación están sujetos a varios factores (tamaño de la base de datos de políticas, veces que se ha utilizado dicha política, . . .) así como de las características de la propia política a delegar.

El uso de la técnica de delegación de control permite subordinar tareas de control a otros componentes, pero debido a que el motivo por el cual se ha realizado está determinado por una serie de circunstancias puntuales, debe ser posible anular dicha acción en el momento que las situación general sea diferente. La operación contraria a la delegación es la **revocación de control**.

> **Definición** REVOCACIÓN DE CONTROL: Mecanismo usado para anular la delegación de una tarea de control otorgada a un componente. Por medio de esta operación un componente superior puede dar de baja una o varias políticas locales de otro componente.

5.5.2 Inicialización y Actualización de las Políticas

Una consecuencia derivada del proceso de distribución de los mecanismos de control en varios componentes es el aumento de la complejidad del proceso de gestión de las propias políticas. Dentro de este problema se plantean situaciones clave como es el caso de inicialización de las bases de datos de políticas de todos los componentes del sistema. Una solución a la inicialización es hacer que la propia implementación de estos componentes rellene estas bases de datos con una serie de políticas iniciales. Este conjunto de políticas ha de ser, sin embargo, muy reducido y completamente genérico, pues sino la flexibilidad proporcionada por la modificación dinámica de las políticas no sería posible.

Un segundo problema aparece a la hora de tratar la actualización de las políticas del sistema, por ejemplo, cuando se determina que la gestión de un determinado recurso (definida por medio de una serie de políticas de varios componentes) ha de ser cambiada de acuerdo a unos nuevos criterios. Este problema tiene mayor complejidad cuanto mayor es el conjunto de componentes que disponen de políticas relacionadas con los criterios que se desean cambiar.

Ambos problemas se resuelven por medio de la utilización de las técnicas (propagación, delegación y revocación). De esta forma (ver Figura 5.4), cuando un componente del sistema se arranca, inicialmente su base de datos de políticas se encuentra vacía. Como consecuencia del arranque cada componente genera en su plano operacional un evento de arranque (`ComponentStart()`) que es analizado por su plano de control. Debido a que inicialmente la base de datos de políticas está vacía, este componente no será capaz de planificar una secuencia de acciones adecuada a dicho

evento, por lo tanto realizará una *propagación de control*. Esta propagación se retransmitirá hasta alcanzar el elemento más alto de la jerarquía de control. Este componente es el único que debe ser inicializado con todas las políticas de todo el sistema, de forma que será capaz de responder al evento `ComponentStart()` lanzado. Como respuesta al mismo el componente selecciona el conjunto de políticas iniciales del componente recién arrancado y las delegará a dicho componente. De esta forma, un componente al ser arrancado, tras la propagación del evento de arranque, recibe de un componente superior la *delegación de control* que define sus políticas iniciales.

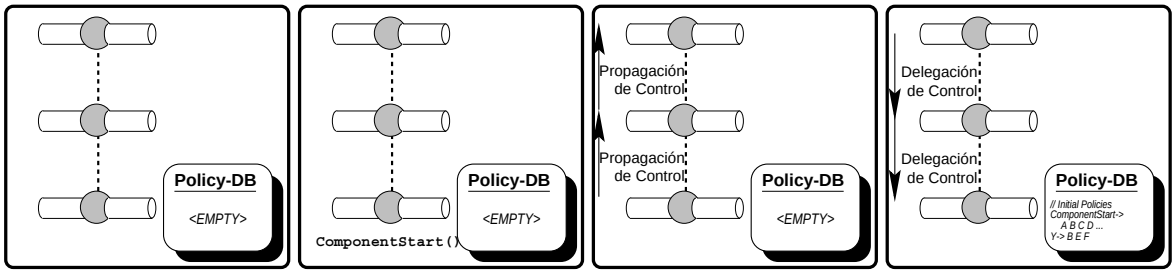
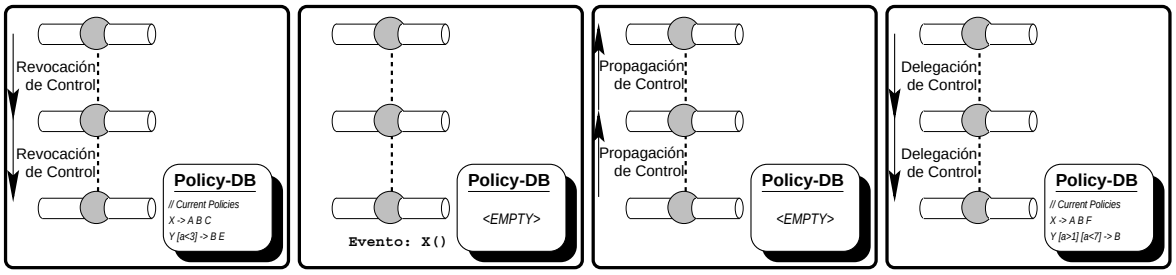


Figura 5.4: Inicialización de un componente

El caso del proceso de actualización es, hasta cierto punto, similar (ver Figura 5.5). El escenario comienza cuando el administrador del sistema desea modificar el conjunto de políticas (dando de baja una serie de ellas y definiendo unas nuevas). En este caso al elemento superior de la jerarquía de control se le comunica la lista de políticas a eliminar y las nuevas políticas. Para las políticas eliminadas se realiza una *revocación de control*. Por lo general, el proceso de modificación implica el reemplazamiento del conjunto de políticas asociadas a una serie de acciones similares, esto quiere decir que una serie de eventos generados por unos determinados componentes se tratarán de forma diferente. Una vez realizada la *revocación de control* dichos eventos no dispondrán en local de ninguna política que los trate por lo tanto se realizará una *propagación de control* que implicará la posterior *delegación de control* procedente de los elementos superiores de la jerarquía. De esta forma, se completa el proceso de actualización del conjunto de políticas. En ciertos casos, se puede realizar la delegación de control automáticamente tras la revocación sin necesidad de que se produzca la demanda por medio de una propagación.

5.5.3 Organización de las Políticas

Con el objetivo de facilitar la gestión las políticas, estas se encuentran agrupadas por dominios. El concepto de **dominio de políticas** se define como:



(a) Eliminación de política *(b) Deteción de evento* *(c) Propagación: Evento* *(d) Delegación: Evento*

Figura 5.5: Actualización de las políticas de un componente

> **Definición** DOMINIO DE POLÍTICAS: Conjuntos de políticas definidos con el objeto de clasificar dichas políticas en grupos homogéneos, caracterizados por tratar una serie de eventos o situaciones similares (relacionadas con el mismo tipo de componentes o encargadas del tratamiento de un tipo eventos). Cada dominio se expresa por medio de una serie de identificadores separados por puntos, de la forma:
> `dominio.dominio.dominio`

Una política determinada puede estar incluida en varios dominios, pero como mínimo debe pertenecer a uno. La definición de los dominios de políticas permite que para ciertas operaciones de gestión de las propias políticas, éstas estén clasificadas de una forma adecuada. Por ejemplo, los procesos de actualización de políticas suelen estar dirigidos a dominios de políticas (por ejemplo todas las políticas de acceso al sistema y autentificación o las políticas de integración del *Data Warehouse*).

5.5.3.1 Metapolíticas

Un dominio de políticas especial es el formado por lo que se denominan **metapolíticas**.

> **Definición** METAPOLÍTICAS: Conjunto de políticas encargadas de la gestión del resto de las políticas. Las funciones controladas por estas políticas son los criterios de delegación, propagación y revocación de políticas, así como las configuración y modificación de los árboles de jerarquías de control.

Estas políticas se encuentran agrupadas en un dominio denominado `Metapolicies`, que a su vez se encuentra dividido en otros dominios según las diferentes tareas, por ejemplo:

- `Metapolicies.Delegation`
- `Metapolicies.Propagation`
- `Metapolicies.Revoke`

❒ `Metapolicies.ControlHierarchy`

Como resulta evidente, no debe existir ningún conjunto de políticas que controle a las *metapolíticas*. La gestión de éstas se encuentra definida de forma interna en cada componente. De esta forma, las metapolíticas responden a los siguientes criterios:

❒ El proceso de inicialización delega automáticamente todas las metapolíticas. Esto ocurre antes de que active el evento `ComponentStart()`.

❒ El proceso de revocación sobre las metapolíticas afecta a todos los componentes del sistema y va inmediatamente seguido de una delegación de las nuevas metapolíticas[3]

❒ Un componente no puede nunca tener un conjunto de metapolíticas vacío. Si esto ocurriese, el componente no sería capaz de resolver problemas que no estuviesen tratados por sus políticas locales al no disponer de mecanismos para realizar la propagación de control.

5.5.4 Casos de Aplicación de Políticas

Para completar la descripción del Modelo de Control, se exponen a continuación una serie de tareas comunes a múltiples sistemas distribuidos y su posible solución por medio del mecanismo de control definido por medio de las políticas.

5.5.4.1 Despliegue del Sistema

Se denomina despliegue del sistema al proceso de inicialización del sistema, mediante el cual una serie de componentes han de ser arrancados en diferentes localizaciones.

❶ **Problemática**:
La inicialización de una aplicación no distribuida se reduce a la ejecución de un único programa. En el caso de aplicaciones distribuidas, resulta necesario ejecutar varios componentes en diferentes nodos, relacionándolos y comunicándolos entre sí para que sean capaces de resolver en conjunto una serie de tareas. El problema de iniciar varios componentes incrementa con el número de ellos a arrancar y con la cantidad de dependencias que existan entre ellos.

❷ **Configuración**:
Se parte de la situación de varios nodos de despliegue disponibles para ejecutar el sistema. Cada uno de dichos nodos dispone de un componente de control ejecutándose en dicho nodo.

[3] Esta operación, sobre un sistema con muchos componentes puede colapsar el funcionamiento del mismo. Dicha operación es teóricamente posible, pero prácticamente desaconsejable.

❸ **Desarrollo**:

El usuario desea iniciar el sistema, para ello inicia el componente *Seed*. Dicho componente conoce: (i)la información de los nodos de despliegue disponibles (si no fuera así debería descubrirlos dinámicamente) y (ii)el conjunto de políticas que han de regir el sistema.

El componente *Seed* será el encargado de iniciar cada uno de los subsistemas, lanzando la ejecución de un componente para cada uno de los subsistema. Estos componentes de cada subsistema, repetirán el proceso para otras subdivisiones dentro de cada subsistema hasta que todos los componentes se encuentren en ejecución.

❹ **Decisiones de Control**:

La realización de las tareas del componente *Seed* y de cada uno de los componentes de cada subsistema requieren resolver las siguientes decisiones de control:

- ➢ Determinar cuáles son los componentes a lanzar: Tanto componentes de control de un subsistema como componentes operacionales.
- ➢ Determinar dónde inicializar dichos componentes: Qué nodo de despliegue ha de albergarlos.
- ➢ Construir la jerarquía de control del sistema.

❺ **Plano de Control**:

El despliegue del sistema se define mediante el tratamiento que cada componente haga del evento:

- ❒ `ComponentStart()`.

Al iniciarse todo componente, éste recibe el evento `ComponentStart()`. El tratamiento de este evento por medio de las políticas determina cuáles son las acciones a realizar al iniciar la ejecución del componente. Los componentes de control de cada subsistema y el *Seed* en el tratamiento de este evento lanzan la ejecución de los componentes que directamente controlan. Mientras se van arrancando los subsistemas y los componentes, se define el árbol de control. Pudiendo ser reequilibrado durante la ejecución del sistema por medio de otros mecanismos.

Este proceso de inicialización de componentes requiere que los componentes de gestión de nodos sean capaces de iniciar la ejecución de los componentes en cada uno de los nodos a petición de otro componente. Este proceso puede requerir la instalación del código de ciertas implementaciones de componente. Esta distribución de código estará soportada por la base de datos de nodos de despliegue que deberá ser accesible por medio del componente *Seed*.

La opción de usar políticas de control a la hora de definir el despliegue del sistema permite elegir entre diferentes configuraciones de una forma flexible, pues dicho despliegue no está definido

dentro del código de los componentes pudiendo ser alterable de una forma dinámica.

5.5.4.2 Migración y Movilidad de Componentes

Una característica muy deseable de los sistemas distribuidos es la posibilidad de mover o replicar componentes o funcionalidades de un nodo a otro.

❶ **Problemática**:

El proceso migración de un código de un ordenador a otro a menudo se restringe a ciertas implementaciones de dicho código en lenguajes interpretados (*scripts*) o quasi-interpretados (como Java). Estas restricciones implican sin embargo una pérdida de generalidad de la arquitectura a la hora de soportar la movilidad de componentes. En su lugar MOIRAE propone la movilidad de cualquier tipo de componente sin restricciones en la implementación del mismo.

❷ **Configuración**:

La posibilidad de migrar un componente requiere que de una misma clase de componente existan implementaciones de componente capaces de ser ejecutadas tanto en el nodo originario del componente como en el nodo destino. Dichas implementaciones pueden encontrase instaladas en los respectivos nodos o pueden ser servidas bajo demanda por los componentes de gestión del nodo desde la base de datos de nodos de despliegue.

❸ **Desarrollo**:

El componente que solicita la migración (pudiendo ser el mismo componente o cualquier otro) notificará a los componentes de gestión de ambos nodos dicha operación. El componente de gestión del nodo original se encargará de congelar y salvaguardar el estado del componente y desactivarlo. El componente de gestión del nodo destino activará una nueva implementación de la misma clase de componente y solicitará al primer gestor el estado. Una vez obtenido el estado lo inyectará al componente activado.

❹ **Decisiones de Control**:

El proceso de migración de componentes requiere la cooperación de los componentes de gestión del nodo y los propios componentes a migrar. Cada componente de gestión deberá tener las siguientes consideraciones:

- ➢ Determinar cuál es el proceso a seguir para la activación y desactivación de componentes.
- ➢ Regulación de las restricciones bajo las cuales es posible o no la migración: Exceso de carga, ancho de banda necesario para transmitir el estado.
- ➢ Seleccionar el mecanismos para la transmisión del estado entre nodos.

Por otro lado, el componente a migrar debe controlar:

- ➢ Cómo se realiza el proceso de congelación y salvaguarda del estado y su posterior recuperación.

❺ **Plano de Control**:
Este escenario requiere de los siguientes servicios de los componentes de gestión del nodo:

- ❒ `bool MoveComponentTo(in Component c, in Node n).`
- ❒ `bool MoveComponentFrom(in Component c,in Node n).`

Además, se dispararán los eventos del plano de control del componente de gestión de nodo:

- ❒ `MigrationSource(Component c, Node n).`
- ❒ `MigrationTarget(Component c, Node n).`

Del componente a migrar se necesitan los servicios:

- ❒ `State SaveState().`
- ❒ `void RestoreState(in State s).`

La migración de un componente se inicia mediante la llamada al servicio `bool MoveComponentTo(in Component c, in Node n)` en el componente de gestión del nodo origen, el cual negociará con el nodo destino por medio del servicio `bool MoveComponentFrom(in Component c,in Node n)`. Dichos servicios generarán sendos eventos `MigrationSource(Component c, Node n)` y `MigrationTarget(Component c, Node n)` que invocarán al plano de control para la evaluación de los criterios que rigen la movilidad de los componentes. Este proceso de negociación determinará si la migración puede realizarse, dicha decisión puede requerir consultar al propio componente sobre la acción a realizar. Una vez decidido, se solicitará al componente el servicio `State SaveState()` que congela y salva su estado. Dicha operación (y la análoga de recuperación de estado: `void RestoreState(in State s)`) pueden realizarse tanto con decisiones del plano de control como sin ellas. La primera opción permite una mayor flexibilidad para controlar el proceso de salvaguarda y recuperación del estado que la otra no tiene.

Las diferentes resoluciones en el plano de control de los eventos tanto de los componentes de gestión de nodo como del componente a migrar discriminan las distintas estrategias a seguir en la migración de componentes.

5.5.4.3 Optimización y Equilibrio de Carga

La configuración óptima de un sistema para que ejecute peticiones es, quizás, una de las tareas de control de más alto nivel y por lo tanto más compleja. Las operaciones de equilibrio de carga hacen uso de la movilidad de los componentes para repartir las necesidades de proceso de la forma más adecuada a la potencia de los nodos disponibles. En conjunto, el equilibrio de carga y otras técnicas son las herramientas usadas para mejorar el rendimiento de un sistema distribuido.

❶ **Problemática**:

El concepto de optimización de un sistema sugiere la organización de componentes y parametrización de los mismos de forma adecuada para mejorar el comportamiento del sistema en base a una serie de criterios.

El número de decisiones relativas a la optimización del sistema es muy elevado y muchas de ellas son muy dependientes de cada sistema que se diseñe. El ámbito que se analiza en este punto va a ser muy general y su aplicación a casos reales dependerá de las características del entorno de aplicación concreto.

❷ **Configuración**:

Generalizando las tareas de optimización, se han identificado cuatro facetas de control: (i) equilibrio de carga, (ii) gestión de recursos, (iii) control de tareas y (iv) topología y proximidad.

Equilibrio de carga: implica el reparto de la ejecución de las tareas de forma que la carga de trabajo sea proporcional a la capacidad de procesamiento del nodo.

Gestión de recursos: se refiere al arbitraje realizado para que diferentes componentes que hacen uso de algún tipo de recurso tengan acceso al mismo en base a una serie de criterios.

Control de tareas: referido a cómo son gestionadas las diferentes tareas concurrentes del sistema. Es necesario establecer prioridades entre las tareas, deteniendo ciertas ejecuciones y evitando que determinados componentes se colapsen.

Topología y proximidad: determinada por la localización de componentes en los nodos. La interacción entre dos componentes es más eficiente cuanto más rápida es la comunicación entre ambos. La ubicación de componentes con alto grado de interacción entre sí dentro del mismo nodo.

❸ **Desarrollo**:

La activación de las decisiones de control se realiza bajo dos circunstancias diferentes. Por un lado, al crearse un nuevo componente su localización debe estudiarse en base a los cuatro criterios antes indicados. Por otro lado, periódicamente, es necesario evaluar si el sistema en conjunto responde a las restricciones definidas sobre los cuatro criterios de optimización. En ambos casos, la evaluación debe realizarse en primer lugar de forma general, si el sistema no

está cargado y no se han detectado conflictos. Si hay tareas detenidas, congestión en algún nodo, recursos insuficientes, entonces el proceso de evaluación tiene que ser más detallado. En ambos procesos de evaluación es posible utilizar diferentes conjuntos de políticas. También es posible descomponer los problemas del sistema en más escenarios (por ejemplo, problemas de interconexión entre nodos, uso de memoria, etc.).

❹ **Decisiones de Control**:
Los criterios y factores en la evaluación de las estrategias de equilibrio de carga son:

- ➢ Una métrica de consumo de recursos para cada uno de los componentes del sistema. Es evidente que no consume el mismo tiempo de CPU un componente que traduce las sentencias de un lenguaje de consulta que un algoritmo que procesa una tabla de datos.
- ➢ Definición de las restricciones de uso de recursos (memoria, CPU, disco) en los diferentes nodos de despliegue del sistema, así como entre diferentes nodos (anchos de banda).
- ➢ Unos criterios de priorización de tareas dentro del sistema, basados, bien en atributos propios del usuario que solicita las tareas u otorgados dinámicamente por el sistema.
- ➢ Mecanismos de congelación y reactivación de tareas y componentes. Mediante estos mecanismos, en situaciones de carga del sistema se podrán detener ciertas consultas de baja prioridad para retomarlas más adelante.

❺ **Plano de Control**:
La optimización del sistema se base en el servicio de los componentes de gestión del nodo:

- ❒ `bool CreateComponent(in ComponentClass k, out Component c).`

Este servicio puede generar los eventos:

- ❒ `LocateComponent(ComponenteClass k).`
- ❒ `StatusDiagnosis().`

La operaciones de creación de componentes, se realizan por medio del componente de gestión de nodo. Este elemento, es el que debe evaluar las políticas de optimización en la creación de nuevos componentes. La llamada al servicio `bool CreateComponent(in ComponentClass k, out Component c)` se realiza a un determinado gestor nodo, el cual intentará crear dicho componente, negociando con el solicitante y otro gestores de nodo la posibilidad de realizarlo y la localización del nuevo componente. Todas estas decisiones se toman en el plano de control, activado por el evento `LocateComponent(ComponenteClass k)`.
El caso de la evaluación periódica de los criterios de optimización, ésta se realiza por la activación en base a un temporizador del evento `StatusDiagnosis()` las acciones de control

asociadas a dicho evento evaluaran el estado local del componente (los conflictos que ha detectado y el estado de su ejecución) así como el estado de los componentes a los que éste controla.

Las diferentes soluciones dadas por el plano de control de los eventos tanto de los componentes de gestión de nodo como del componente a migrar discriminan las distintas estrategias a seguir en la migración de componentes.

5.6 Formalización

Para concluir la definición de esta nueva arquitectura, a continuación se propone una herramienta formal que permita modelizar un sistema basado en componentes como los descritos por los modelos de la arquitectura que se acaba de presentar. Los componentes de la arquitectura y los mecanismos que los articulan son representados por medio de una notación matemática formal. En base a dicha notación es posible el análisis de ciertas propiedades deseables de un sistema basado en estos componentes. Una evaluación del cumplimiento de dichas propiedades representa una herramienta de gran utilidad para la validación de sistemas.

El formalismo usado para representar los componentes MOIRAE es una extensión del modelo de sistemas asíncronos propuesto por Tuttle [LT87a] y Lynch y Tuttle [LT87b]. El modelo original se basa en el concepto de **autómata de entrada/salida** (*i/o automaton* o autómata E/S). En esta sección se extiende el formalismo para incluir las decisiones de control dentro de cada componente, pues el autómata original de Tuttle carece de parte de control. Aunque el nuevo modelo propuesto es sutilmente diferente, se define la transformación de un autómata MOIRAE a un autómata E/S y viceversa, estando en este segundo caso, restringido a una serie de condiciones. La posibilidad de transformar el nuevo modelo de autómata a un autómata E/S permite utilizar las aportaciones realizadas en relación a la validación de ciertas propiedades por sistemas distribuidos que han sido formuladas por medio de la notación de autómatas E/S.

A continuación se presenta únicamente la parte del modelo que extiende el autómata E/S. Dicho modelo original se encuentra recogido en el Apéndice A y de una forma más extensa en [LT87a].

5.6.1 Definición de un Componente

Cada uno de los componentes de un sistema basado en la arquitectura MOIRAE se encuentra modelizado por medio de un *autómata MOIRAE*. La definición de dicho término y de otros conceptos necesarios se describe a continuación.

5.6.1.1 Firma

En primer lugar se define el concepto de **firma** (*signature*) como:

> **Definición** FIRMA: Descripción de entradas y salidas de un elemento del sistema. Una firma, referida como S, es una tupla compuesta por los siguientes elementos: *servicios*, $srv(S)$; *peticiones al exterior*, $req(S)$; *comandos internos*, $cmd(S)$ y *protocolo de control*, $ctrl(S)$.

La definición de firma propuesta difiere de la enunciada por Tuttle en primer lugar, en la nomenclatura usada (para evitar la confusión) y principalmente en la inclusión del término $ctrl(S)$. El protocolo de control ($ctrl(S)$) se corresponde con una serie de peticiones que intercambian los diferentes planos de control. Los elementos de $ctrl(S)$ pertenecen al conjunto definido por el producto cartesiano $ctrl\_types \times NAMES \times NAMES$.

- $ctrl\_types$ es el conjunto de los tipos de peticiones. Los elementos de este conjunto han de ser iguales para todos los componentes, es decir todos han de reconocer el mismo protocolo de control. Estos tipos de acciones se encuentran divididos en tres subconjuntos disjuntos:
  - *peticiones de control* $ctrl\_queries$,
  - *respuestas de control* $ctrl\_answers$ y
  - *peticiones múltiples de control* $ctrl\_broadcast$.
- $NAMES$ es el espacio de nombrado. Conjunto de elementos simbólicos, unívocamente asociados a los componentes del sistema. Define tanto origen como destino de los mensajes del protocolo de control.

El conjunto formado por todas las acciones de la firma se representa por $acts(S)$ y se define como:

$$acts(S) = srv(S) \cup req(S) \cup cmd(S) \cup ctrl(S) \tag{5.1}$$

y el conjunto de las acciones controladas localmente por el componente se representa como $local(S)$ y está compuesto por:

$$local(S) = req(S) \cup cmd(S) \tag{5.2}$$

5.6.1.2 Autómata MOIRAE

En base al concepto de *firma* es posible definir el término **autómata MOIRAE** (*MOIRAE automaton*) como:

Definición AUTÓMATA MOIRAE: Modelo matemático que describe el comportamiento de cada uno de los componentes de cualquier sistema. Un autómata A está compuesto por varios elementos: $sig(A)$, $names(A)$, $states(A)$, $start(S)$, $events(A)$, $trans(A)$, $throw(A)$, $pDB(A)$ y $tasks(A)$.

Definición ELEMENTOS DE UN AUTÓMATA MOIRAE: Un autómata MOIRAE A se compone de los siguientes elementos:

- ❒ Firma: $sig(A)$ descripción de las acciones del componente.
- ❒ Nombre: $names(A)$ un subconjunto del espacio de nombrado $NAMES$ que representa el conjunto de nombres asociado a cada componente ($names(A) \subseteq NAMES$).
- ❒ Conjunto de estados (no necesariamente finito): $states(A)$.
- ❒ Conjunto de estados inicial: $start(A) \subseteq states(A)$.
- ❒ Conjunto de eventos: $events(A)$.
- ❒ Transición entre estados: $trans(a)$ representable por medio de un subconjunto del producto cartesiano de estados y acciones ($trans(A) \subset states(A) \times acts(sig(A)) \times states(A)$).
- ❒ Condiciones de evento: $throw(A)$ complementaria a la anterior ($throw(A) \subset states(A) \times acts(sig(A)) \times events(A)$).
- ❒ Una base de datos de políticas de control: $pDB(A)$ usada por el *algoritmo de control de componente* que se vera más adelante (Sección 5.6.2).
- ❒ Una partición del espacio de tareas: $tasks(A)$, una relación de equivalencia sobre las acciones locales del autómata $local(sig(A))$.

A partir de este punto y en adelante su usará la notación $srv(A)$ en lugar de $srv(sig(A))$ y de forma análoga $cmd(A)$, $ctrl(A)$, ...

Este formalismo de descripción de un componente es una definición de la información interna del componente así como de la que se conoce desde el exterior. De la misma forma, esta representación describe tanto las operaciones que el componente procesa como de la situación interna del mismo. Si se analiza por separado esta información se tiene:

- ❒ **Acciones**: Las operaciones en las que un componente interviene se denominan acciones. Las acciones se encuentran descritas por la *firma* ($sig(A)$). Estas acciones pueden ser dividías en base a quién activa dicha acción y a quién es el destinatario.
  - ➢ **Acciones entrantes**: Son aquellas que son solicitadas al componente y que son activadas por elementos externos. El conjunto de dichas acciones está compuesto por los servicios

$srv(A)$ y por los elementos del protocolo de control que tengan como destino el nombre de este componente.

$$input(A) = srv(A) \cup \{(ct, n_s, n_d) \in ctrl(A) : n_d \in names(A)\} \tag{5.3}$$

➢ **Acciones salientes**: Son las que el componente solicita a otros elementos externos o incluso a él mismo. Dicho conjunto lo conforman las peticiones al exterior $req(A)$, los comandos internos $cmd(A)$ y las acciones de control de las que es origen.

$$output(A) = req(A) \cup cmd(A) \cup \{(ct, n_s, n_d) \in ctrl(A) : n_s \in names(A)\} \tag{5.4}$$

❒ **Transiciones**: Representan cual es proceso de modificación del estado del componente en base a la realización de acciones. Esta información se encuentra representada por el conjunto de transiciones y las condiciones de evento. Ambos conjuntos para cada autómata A, han de cumplir la siguiente propiedad:

Propiedad:

$$\forall (s_t, \pi_t, s'_t) \in trans(A) : \nexists (s_e, \pi_e, e) \in throw(A) \text{ tal que } s_e = s_t \wedge \pi_e = \pi_t \tag{5.5}$$

$$\forall (s_e, \pi_e, e) \in throw(A) : \nexists (s_t, \pi_t, s'_t) \in trans(A) \text{ tal que } s_e = s_t \wedge \pi_e = \pi_t \tag{5.6}$$

$$\forall s \in state(A) \wedge \forall \pi \in input(A) \text{ debe cumplirse que } \begin{cases} \exists (s, \pi, s') & \in trans(S) \\ & \text{ó} \\ \exists (s, \pi, e) & \in throw(S) \end{cases} \tag{5.7}$$

5.6.1.3 Ejecuciones y Trazas

El funcionamiento de un autómata A se encuentra definido por medio de lo que se denominan **fragmentos de ejecución**.

> **Definición** FRAGMENTO DE EJECUCIÓN: Es una secuencia finita del tipo $s_0, \pi_1, s_1, \pi_2, s_2, \ldots, \pi_r, s_r$ o infinita $s_0, \pi_1, s_1, \pi_2, s_2, \ldots, \pi_r, s_r, \ldots$ alternada de estados y acciones que debe cumplir que para todo $i \geq 0$, exista un elemento de $trans(A)$ del tipo $(s_i, \pi_{i+1}, s_{i+1})$.

Si un *fragmento de ejecución* comienza en un estado perteneciente a $start(A)$ entonces se denomina **ejecución**.

> **Definición** FRAGMENTO DE EJECUCIÓN CON CONFLICTO: Es un fragmento de ejecución finito del tipo $s_0, \pi_1, s_1, \pi_2, s_2, \ldots, \pi_r, e$, donde e es un elemento de $events(A)$. Dicha secuencia debe cumplir que (s_{r-1}, π_r, e) pertenezca a $throw(A)$.

Igual que en el caso anterior si el estado inicial s_0 pertenece a $start(A)$ entonces dicho fragmento se denomina **ejecución con conflicto**.

El funcionamiento del autómata puede corresponderse con cualquiera de los dos tipos de ejecuciones, en el caso de las ejecuciones con conflicto, se trata de una secuencia de acciones que han llevado a un estado en el cual el componente no es capaz de responder a alguna acción que debe realizar, por lo cual produce un evento. Dicho evento es considerado como una situación anómala que debe ser resuelta por medio de decisiones de control. La evaluación de las diferentes acciones de control se basa en el concepto de **ejecución equivalente**.

> **Definición** EJECUCIÓN EQUIVALENTE: Se denomina *ejecución equivalente* a una ejecución con conflicto $EXE = s_0, \pi_1, s_1, \pi_2, s_2, \ldots, \pi_r, e$ a la ejecución sin conflicto asociada $EXE' = s_0, \pi_1, s_1, \pi_2, s_2, \ldots, \pi'_r, s'_r, \ldots \pi'_{r+n}, s'_{r+n}$. Donde $\pi'_r, s'_r, \ldots \pi'_{r+n}, s'_{r+n}$ es producido por la función $control\_seq(s_{r-1}, e)$. Al resultado de dicha función se denomina *secuencia de acciones de control.*

La función $control\_seq(s_{r-1}, e)$ se calcula por medio del *algoritmo de control de componente* (Sección 5.6.2).

A menudo, no interesa cuales han sido los estados u operaciones internas del funcionamiento de un componente sino, únicamente la perspectiva externa del mismo. De esta forma se define **traza** de una ejecución como:

> **Definición** TRAZA: Dada una ejecución α de un autómata A, a la subsecuencia formada por elementos pertenecientes a $srv(A) \cup req(A)$ se la denomina *traza* de α y se denota $trace(\alpha)$.

De forma análoga, se denomina **traza de control** a:

> **Definición** TRAZA DE CONTROL: Es la subsecuencia de una ejecución α del autómata A compuesta únicamente por elementos de $ctrl(A)$. Se representa como $ctrl\_trace(A)$.

El conjunto de todas las ejecuciones sin conflictos de un autómata A se representa como $execs(A)$. De forma análoga al conjunto de todas las trazas derivadas de ejecuciones de A se las denota como $traces(A)$ y $ctrl\_traces(A)$ representa todas las trazas de control del autómata A.

5.6.1.4 Composición de Autómatas

El objetivo de los componentes de un sistema distribuido es la cooperación para la resolución de tareas en conjunto. Formalmente esta cooperación se define en base a la operación de **composición**. Dicha operación implica que cuando un componente ejecuta una acción π el resto de

componentes que tengan π en su firma deben realizarla igualmente.

A la hora de aplicar la composición a varios autómatas, descritos por medio de un conjunto finito de firmas $\{S_i\}_{i\in I}$, éstas deben cumplir una serie de propiedades, denominadas **condiciones de compatibilidad**:

① $cmd(S_i) \cap acts(S_j) = \emptyset \forall i, j \in I, i \neq j$

② $req(S_i) \cap req(S_j) = \emptyset \forall i, j \in I, i \neq j$

③ No existe ninguna acción π tal que $\pi \in acts(S_i) \forall i \in I$.

Un conjunto finito de autómatas $\{A_i\}_{i\in I}$ es **compatible** si el conjunto de firmas asociado lo es (cumple las condiciones de compatibilidad) y además cumple:

① $events(A_i) \cap events(A_j) = \emptyset \ \forall i, j \in I, i \neq j.$

② $names(A_i) \cap names(A_j) = \emptyset \ \forall i, j \in I, i \neq j.$

Para dicho conjunto de autómatas debe existir una función de **resolución de nombres** $resolve$, de la forma:

$$resolve : NAMES \longrightarrow \{A_i\}_{i\in I}$$

De forma que $resolve(N) = A_i$ si y sólo si $N \in names(A_i)$. Esta función es una función unívoca y onyectiva (o parcial, es decir que no todos los elementos del espacio de nombres $NAME$ tienen asociado un componente).

La composición de un conjunto compatible de autómatas es a su vez un autómata. La composición de las firmas $\{S_i\}_{i\in I}$ denotada por $\prod_{i\in I} S_i$ se define como:

$$\begin{array}{lcllcl} srv(\prod_{i\in I} S_i) & = & \bigcup_{i\in I} srv(S_i) & req(\prod_{i\in I} S_i) & = & \bigcup_{i\in I} req(S_i) \\ cmd(\prod_{i\in I} S_i) & = & \bigcup_{i\in I} cmd(S_i) & ctrl(\prod_{i\in I} S_i) & = & \bigcup_{i\in I} ctrl(S_i) \end{array} \tag{5.8}$$

La composición de un conjunto finito de autómatas compatibles $\{A_i\}_{i\in I}$ es un nuevo autómata representado por $\prod_{i\in I} A_i$ y definido como:

$$\begin{array}{lcllcl} sig(\prod_{i\in I} A_i) & = & \prod_{i\in I} sig(A_i) & names(\prod_{i\in I} A_i) & = & \bigcup_{i\in I} names(A_i) \\ states(\prod_{i\in I} A_i) & = & \prod_{i\in I} states(A_i)^{(1)} & start(\prod_{i\in I} A_i) & = & \prod_{i\in I} start(A_i)^{(1)} \\ events(\prod_{i\in I} A_i) & = & \bigcup_{i\in I} events(A_i) & trans(\prod_{i\in I} A_i) & = & \{(s,\pi,s')\}^{(2)} \\ throw(\prod_{i\in I} A_i) & = & \{(s,\pi,e)\}^{(3)} & pDB(\prod_{i\in I} A_i) & = & \bigcup_{i\in I} pDB(A_i) \\ tasks(\prod_{i\in I} A_i) & = & \bigcup_{i\in I} tasks(A_i) & & & \end{array} \tag{5.9}$$

(1) Denotando mediante $\prod$ el producto cartesiano de todos los conjuntos de estados y de estados iniciales respectivamente.

(2) El conjunto de tuplas (s, π, s') de forma que para cada $i \in I$, si $\pi \in acts(A_i)$, entonces $(s_i, \pi, s'_i) \in trans(A_i)$ (donde s_i representa el componente i-esimo del vector de estado), en otro caso $s_i = s'_i$.

(3) El conjunto de tuplas (s, π, e) si existe un $i \in I$, tal que $\pi \in acts(A_i)$ y $(s_i, \pi, e) \in events(A_i)$ (donde s_i representa el componente i-esimo del vector de estado).

5.6.2 Algoritmos de Control

Como ya se ha visto, el funcionamiento del autómata A se dirige por medio de dos conjuntos de transiciones $trans(A)$ y $throw(A)$. La primera representa como varía el estado del componente en base a acciones y la segunda identifica los estados en los cuales ciertas acciones no se pueden emprender. Dichos pares de estado y acción se denominan **conflictos**. El conjunto $throw(A)$ asocia a cada conflicto un elemento del conjunto de eventos $events(A)$. El elemento clave de los algoritmos de control es la base de datos de políticas. Este elemento representa un autómata de control definido como:

Definición BASE DE DATOS DE POLÍTICAS: Una base de datos de políticas $pDB(A)$ es una tupla compuesta por:

- ❒ *Estados de control*: $control\_states(pDB(A))$ conjunto de estados internos del proceso de control.
- ❒ *Estados iniciales*: $control\_start(pDB(A))$ elementos del conjunto $control\_states(pDB(A))$ que se corresponden con los estados de partida del autómata.
- ❒ *Estados finales*: $control\_final(pDB(A))$ subconjunto de $control\_states(pDB(A))$ asociado con los estado de finalización del autómata.
- ❒ *Entrada de control*: $control\_input(pDB(A))$ es una relación que asocia eventos y estados de A con estados iniciales de $pDB(A)$. $control\_input(pDB(A)) \subseteq events(A) \times states(A) \times control\_start(pDB(A))$.
- ❒ *Transiciones de control*: $control\_trans(pDB(A))$ transiciones de estado del autómata de control representadas por medio de tuplas del tipo: (cs, in, cs') donde cs y cs' son elementos de $control\_states(p)$, in es un elemento del conjunto $input(A)$ de acciones entrantes del autómata. $control\_trans(pDB(A)) \subseteq control\_states(pDB(A)) \times input(A) \times control\_states(pDB(A))$
- ❒ *Acciones de control*: $control\_actions(pDB(A))$ relación existente entre estados del autómata de control $control\_states(pDB(A))$ y secuencia de acciones salientes $output(A)$.

Sobre la base de datos de políticas $pDB(A)$ se definen una serie de términos:

- **Ejecución de Control**: Dada una base de datos de políticas $pDB(A)$, a la secuencia finita $\alpha = s_0, \rho_1, s_1, \ldots, \rho_n, s_n$; tal que $s_0 \in control\_start(pDB(A))$, $s_n \in control\_final(pDB(A))$, para todo $1 \geq i < n$ se cumple que $s_n \notin control\_start(pDB(A)) \cup control\_final(pDB(A))$ y para todo $j \geq 0$ se cumpla que $(s_j, \rho_{j+1}, s_{j+1}) \in control\_trans(pDB(A))$ entonces se la denomina *ejecución de control* de $pDB(A)$.

- **Accesibilidad**: Dada una base de datos de políticas $pDB(A)$ y un estado $s \in control\_states(pDB(A))$, se dice que s es accesible si existe alguna ejecución de control α de $pDB(A)$ que contenga a s.

Los algoritmos de control se utilizan para gestionar estos conflictos en base a decisiones de control. Los algoritmos de control son:

① **Algoritmo de General de Control**.

① **Operación de Propagación de Control**.

① **Operación de Acción Directa de Control**.

① **Operación de Delegación de Control**.

① **Operación de Revocación de Control**.

5.6.2.1 Algoritmo General de Control

Este algoritmo define la manera en la que se resuelve un conflicto (identificado por un evento) acontecido en un estado determinado por medio de la definición de una secuencia de acciones y es el que se muestra a continuación:

```
Algorithm: control_seq
Input: Evento e, Estado s
Output: Secuencia de Acciones {a}
Data: Base de Datos de Políticas pDB(A)

control_seq(Event e, Status s)
{
   cs=control_input(pDB(A)).find(e,s)
   /*Estado de control asociado. Tupla (e,s,cs)
     perteneciente al conjunto control_input(pDB(A)) */

   {a}=control_actions(pDB(A)).find(cs)
   /*La secuencia inicial de acciones se
     corresponde con las acciones de control
     del estado de control inicial. */

   exec({a})
   /*Se ejecuta la secuencia de acciones
     de control del estado inicial. */
```

```
   while(cs NOT IN control_final(pDB(A)))
   {
      in=read_input()
      /*Se espera la llegada de una acción
        exterior.*/

      new_cs=control_trans(pDB(A)).find(cs,in)
      /*Se busca el siguiente estado que
        se alcanza en base a dicha acción. */

      if(new_cs == NOT FOUND)
      {
         input_queue.append(in)
         /*Si la acción no implica la transición
           a otro estado, no es la acción
           esperada, y se realimenta a la cola
           de acciones entrantes. */
      }
      else
      {
         exec(in) /*Se ejecuta la acción entrante.*/

         cs=new_cs /*Se trata el siguiente estado.*/

         {a}=control_actions(pDB(A)).find(cs)
         /*Se obtiene la siguiente secuencia de
           acciones de control. */

         exec({a})
         /*Se ejecuta la secuencia de acciones.*/
      }
   }
}
```

La principal ventaja de este algoritmo de control es que es suficientemente general como para regir el comportamiento de todas las acciones de control del componente, incluidas las acciones de control que modifican las propias políticas de control del componente, como a se muestra en las secciones siguientes.

5.6.2.2 Operación de Propagación de Control

El algoritmo general de control se usa para la resolución de los conflictos detectados, usando la información almacenada en la base de datos de políticas de un componente. No siempre es posible resolver todos los conflictos por medio de dicha información. En el momento en el cual la llamada `cs=control_input(pDB(A)).find(e,s)` del algoritmo general no encuentra un estado por donde comenzar, es necesario realizar una *propagación de control*.

El objetivo del proceso de propagación de control es notificar externamente la imposibilidad de resolver un conflicto. El resultado de la operación puede ser una secuencia de acciones plani-

ficada externamente (por medio de una *acción directa de control*) o más información a gestionar por la base de datos de políticas que permita resolver el conflicto (por medio de una *delegación de control*).

La operación de propagación de control consta de los siguientes pasos:

① Transmisión al componente superior de la jerarquía de control del evento producido mediante la petición de control $control\_propagation(e, s)$. Esto implica que exista un elemento $(control\_propagation(e, s), n_1, n_2)$ en el conjunto $ctrl(A)$ como posible petición de control, donde n_1 es uno de los nombres del componente ($n_1 \in names(A)$) y n_2 es uno de los nombres del componente de control inmediatamente superior.

② Se espera la notificación de una secuencia de acciones a realizar, procedente del componente de control superior. Esta notificación llega por medio de una acción entrante denominada $actions\_requested(\beta_{(e,s)})$. El argumento de esta acción es la secuencia de acciones de control a realizar.

③ Por medio la acción interna $execute\_action\_sequence(\beta_{(e,s)})$ se ejecuta la secuencia de acciones transmitida $\beta_{(e,s)}$.

El mecanismo mediante el cual se realiza esta operación es representable por medio de una secuencia de acciones de control planificada por medio del algoritmo general de control. Para ello, dado el evento e en el estado s, se deben añadir a la base de datos de políticas los siguientes elementos:

❒ $cs_{(e,s)}, cs_{end} \in control\_state(pDB(A))$

❒ $cs_{(e,s)} \in control\_start(pDB(A))$

❒ $cs_{end} \in control\_final(pDB(A))$

❒ $(e, s, cs_{(e,s)}) \in control\_input(pDB(A))$

❒ $(cs_{(e,s)}, actions\_requested(\beta_{(e,s)}), cs_{end}) \in control\_trans(pDB(A))$

❒ $(cs_{(e,s)}, \alpha_{(e,s)}), (cs_{end}, \gamma(\beta_{(e,s)})) \in control\_actions(pDB(A))$

Donde $\alpha_{(e,s)}$ es la secuencia de acciones $\{control\_propagation(e, s)\}$, $\beta_{(e,s)}$ es la secuencia de acciones planificada por el controlador superior y $\gamma(\beta_{(e,s)})$ es la secuencia de acciones consistente en: $\{execute\_action\_sequence(\beta_{(e,s)})\}$, siendo $\beta_{(e,s)}$ la secuencia de acciones antes citada.

Representado de forma gráfica, estos estados se pueden observar en el subautómata de la Figura 5.6.

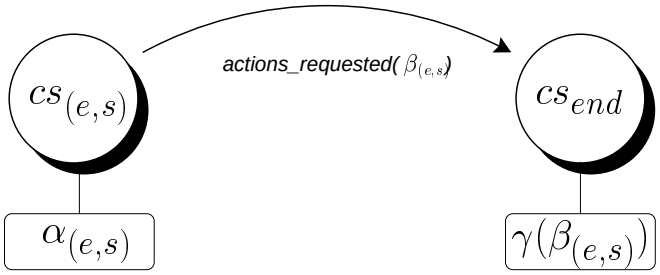


Figura 5.6: Estados asociados a la operación de propagación

5.6.2.3 Operación de Acción Directa de Control

Como resultado del proceso de *propagación de control* una de las respuestas es la intervención directa de otro componente de control que define la secuencia de acciones a realizar y se las transmite al componente que originó la propagación.

El objetivo del proceso de acción directa es usar la información de un componente de control de mayor ámbito para resolver conflictos presentes en un componente situado por debajo en la jerarquía de control.

La operación de acción directa de control consta de los siguientes pasos:

① Se comienza en un estado de control asociado a un evento y estado remoto $cs'_{(e,s)}$. Se llega a dicho estado por medio de la recepción de la acción entrante $control\_propagation(e,s)$ asociada con la vista en la operación anterior.

② Las transiciones de control y los estados son similares a la resolución de un conflicto local. La única diferencia es que mientras que en el caso de conflictos locales, las acciones emprendidas son siempre locales, para el caso de conflictos remotos, se definen tanto acciones locales, como se planifican acciones remotas a transmitir al componente.

③ En el estado final al que se transita desde el estado inicial se le debe añadir una última acción del tipo $actions\_requested(\beta_{(e,s)})$ mediante la cual se transmiten las acciones a realizar por el componente originario del conflicto.

La forma mediante la cual se realiza esta operación es exactamente igual a la resolución de un conflicto local por medio del algoritmo general de control. Es decir, que para que un conflicto retransmitido por otro componente se pueda resolver por un autómata dado, se debe disponer de los estados, transiciones y demás elementos asociados a dicho evento.

5.6.2.4 Operación de Delegación de Control

La otra alternativa a la resolución de conflicto remoto es la *delegación de control*. Cuando un componente solicita por medio de la *propagación de control* que otro componente resuelva un conflicto, este segundo componente puede transmitir información a añadir a la base de datos de políticas. Esta nueva información, representada por un nuevo sub-autómata de control contiene los estados y transiciones necesarios para que el componente que originó el conflicto pueda tratarlo.

El objeto del proceso de delegación de control es la actualización de las bases de datos de políticas con información para que tanto el conflicto en cuestión como futuras situaciones se puedan resolver localmente.

La operación de delegación de control afecta tanto al componente que genera el conflicto como al que realiza la propia delegación. El primero recibirá una información que el segundo debe definir. La información transmitida serán un conjunto de estados, transiciones y demás elementos que conforman un sub-autómata de control.

En el caso de los componentes que originan el conflicto, es necesario definir los siguientes elementos:

① $cs'_{(e,s)} \in control\_state(pDB(A))$

② $(cs_{(e,s)}, control\_delegation(CA_{(e,s)}), cs'_{(e,s)}) \in control\_trans(pDB(A))$

③ $(cs'_{(e,s)}, \delta(CA_{(e,s)})) \in control\_actions(pDB(A))$

Donde $CA_{(e,s)}$ es el sub-autómata de control transmitido que ha de ser combinado con el autómata actual. $\delta(CA_{(e,s)})$ es la secuencia de acciones compuesta por $combine\_automaton(CA_{(e,s)})$, la cual combina el autómata de control actual con el transmitido.

La Figura 5.7 representa el autómata de control asociado a los nuevos elementos añadidos a la base de datos de políticas. Debido a que el caso de la delegación de control es una alternativa a la respuesta data a la propagación de control, el estado $cs_{(e,s)}$ y su secuencia de acciones asociadas $\alpha_{(e,s)}$ son los mismos que en aquel caso.

En el caso del componente que define el sub-autómata de control, éste realiza un proceso análogo al descrito para la operación de acción directa de control, con la diferencia de que la acción realizada es $control\_delegation(CA_{(e,s)})$.

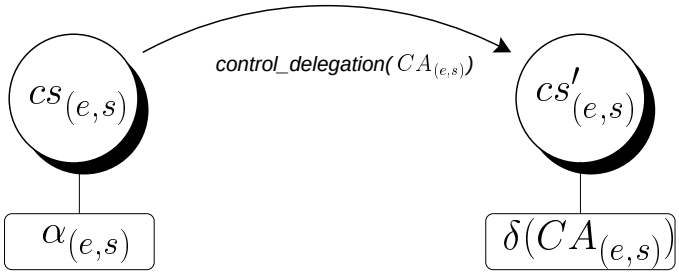


Figura 5.7: Estados asociados a la operación de delegación

Definición COMBINACIÓN AUTÓMATAS DE CONTROL: Dadas dos bases de datos de políticas $pDB(A)$ y $pDB(B)$ representadas por sendos autómatas de control, se denomina combinación de dichas bases de datos o de los autómatas de control asociados al nuevo autómata $pDB(A \triangleleft B)$ formado por:

- *Estados de control*:
  $control\_states(pDB(A \triangleleft B)) = control\_states(pDB(A)) \cup control\_states(pDB(B))$.
- *Estados iniciales*:
  $control\_start(pDB(A \triangleleft B)) = control\_start(pDB(A)) \cup control\_start(pDB(B))$.
- *Estados finales*:
  $control\_final(pDB(A \triangleleft B)) = control\_final(pDB(A)) \cup control\_final(pDB(B))$.
- *Entrada de control*:
  $control\_input(pDB(A \triangleleft B)) = control\_input(pDB(B)) \cup \{(e,s,cs) \in control\_input(pDB(A)) / \nexists (e,s,cs) \in control\_input(pDB(B))\}$.
- *Transiciones de control*:
  $control\_trans(pDB(A \triangleleft B)) = control\_trans(pDB(B)) \cup \{(cs,in,cs') \in control\_trans(pDB(A)) / \nexists (cs,in',cs'') \in control\_trans(pDB(B))\}$.
- *Acciones de control*:
  $control\_actions(pDB(A \triangleleft B)) = control\_actions(pDB(B)) \cup \{(cs,\alpha) \in control\_actions(pDB(A)) / \nexists (cs,\beta) \in control\_actions(pDB(B))\}$.

5.6.2.5 Operación de Revocación de Control

La operación contraria a la de *delegación de control* es la *revocación de control*. Por medio de dicha operación se elimina cierta información de la base de datos de políticas. El proceso de revocación de control se usa para eliminar la posibilidad de control local de una serie de conflictos debido a que el criterio actual del sistema requiere que dichos conflictos sean resueltos por com-

ponentes superiores en la jerarquía de control. La operación de revocación de control, a diferencia de la acción directa o la delegación no se realiza durante el proceso de resolución de un conflicto. Un componente de control superior, puede decidir que en un momento dado, se realice dicha operación sobre uno o varios componentes inmediatamente inferiores en la jerarquía. Para hacer que la aplicación de la revocación sea tratada por medio del algoritmo general de control, se asocia a la acción entrante $control\_revoke(CA)$ un evento determinado $e_{rev} \in events(A)$ en el autómata MOIRAE general A ($\forall s \in states(A)(s, control\_revoke(CA), e_{rev}) \in throw(A)$). Como resultado de dicho evento se inicia el autómata de control del componente que debe tener un estado de control nuevo cs_{rev}. Dicho autómata deberá disponer de los elementos:

① $cs_{rev} \in control\_state(pDB(A))$

② $cs_{rev} \in control\_start(pDB(A))$

③ $cs_{rev} \in control\_final(pDB(A))$

④ $\forall s \in states(A)\ (e_{rev}, s, cs_{rev} \in control\_input(pDB(A))$

⑤ $(cs_{rev}, \epsilon(CA)) \in control\_actions(pDB(A))$

Definición COCIENTE DE AUTÓMATAS DE CONTROL: Dadas dos bases de datos de políticas $pDB(A)$ y $pDB(B)$ representadas por sendos autómatas de control, se denomina cociente de dichas bases de datos o de los autómatas de control asociados al nuevo autómata $pDB(A \triangleright B)$ formado por:
Estados de control:
$control\_states(pDB(A \triangleright B)) = control\_states(pDB(A)) - control\_states(pDB(B))$.
Estados iniciales:
$control\_start(pDB(A \triangleright B)) = control\_start(pDB(A)) - control\_start(pDB(B))$.
Estados finales:
$control\_final(pDB(A \triangleright B)) = control\_final(pDB(A)) - control\_final(pDB(B))$.
Entrada de control:
$control\_input(pDB(A \triangleright B)) = control\_input(pDB(A)) - control\_input(pDB(B)) - \{(e, s, cs) \in control\_input(pDB(A))/ cs \in control\_states(pDB(B))\}$.
Transiciones de control:
$control\_trans(pDB(A \triangleright B)) = control\_trans(pDB(A)) - control\_trans(pDB(B)) - \{(cs, in, cs') \in control\_trans(pDB(A))/ cs, cs' \in control\_states(pDB(B))\}$.
Acciones de control:
$control\_actions(pDB(A \triangleright B)) = control\_actions(pDB(A)) - control\_actions(pDB(B)) - \{(cs, \alpha) \in control\_actions(pDB(A))/ cs \in control\_states(pDB(B))\}$.

Donde $\epsilon(CA)$ es la secuencia de acciones formada por un único elemento $delete\_automaton(CA)$ que realiza el cociente del autómata de control actual y del pasado como argumento.

5.6.2.6 Modificación de las Operaciones de Control

Como se ha visto, todas las operaciones de control una vez definido el algoritmo general de control pueden ser expresadas por medio de estados y transiciones asociados a ciertas acciones dentro de los autómatas de control. Esta característica permite que las propias operaciones de control de un componente sean redefinibles por medio de las mismas operaciones de control. Esto posibilita la modificación de estas operaciones para, por ejemplo plantear nuevas topologías de control distintas de las jerarquías descritas en los modelos de la arquitectura.

Los sub-autómatas usados en las operaciones de combinación y cociente pueden estar definidos en base a los dominios de políticas vistos en la sección 5.5.3. Esta organización facilita su clasificación y mantenimiento. Por ejemplo, asociado al problema del equilibrio de carga se pueden agrupar todas las políticas en un dominio que son transmitidas o eliminadas en bloque y que en conjunto definen los criterios para realizar dicha acción.

Hay que tener en cuenta que la operación cociente de un autómata no es la inversa de la combinación pues la fórmula $pDB((A \triangleleft B) \triangleright B) = pDB(A)$ no se cumple para todo A y B sub-autómatas. Contraejemplo válido es cualquier caso en el que algún estado o transición pertenezca tanto a $pDB(A)$ como a $pDB(B)$.

5.6.3 Transformación entre los Autómatas E/S y MOIRAE

Como se ha indicado anteriormente el modelo de autómata MOIRAE se encuentra basado en el modelo de autómata de E/S propuesto por Tuttle. El nuevo modelo incluye el concepto de evento como entrada a un segundo autómata, denominado de control, cuya información se encuentra definida en base a la base de datos de políticas $pDB(A)$.

El formalismo de Tuttle se usa en la actualidad para modelizar numerosos problemas de concurrencia o algoritmos sobre sistemas distribuidos. Asimismo, existen ciertas propiedades y teoremas válidos para autómatas E/S que resultan interesantes para cualquier proceso de validación y análisis de un conjunto de componentes. Con el objetivo de encontrar una equiparación entre los dos formalismos se definen las siguientes transformaciones:

Definición: Dado un autómata E/S A se define un autómata MOIRAE A' denominado equivalente,

tal que todo componente real modelizable por medio de A en el formalismo de automatas E/S, lo es también por medio de A' en automatas MOIRAE.

Desarrollo: Estando el autómata E/S A compuesto por: $sig(A)$, $states(A)$, $start(A)$, $trans(A)$ y $tasks(A)$. Se define un autómata MOIRAE A' compuesto por: $sig(A')$, $names(A')$, $states(A')$, $start(S)$, $events(A')$, $trans(A')$, $throw(A')$, $pDB(A')$ y $tasks(A')$, tal que:

- $sig(A')$ contiene los elementos:
  - $srv(sig(A')) = in(sig(A))$.
  - $req(sig(A')) = out(sig(A))$.
  - $cmd(sig(A')) = int(sig(A))$.
  - $ctrl(sig(A')) = \emptyset$.
- $names(A') = \{\Delta_A\}$, siendo Δ_A un identificador único para el componente A.
- $states(A') = states(A)$.
- $start(A') = start(A)$.
- $events(A') = \emptyset$.
- $trans(A') = trans(A)$.
- $throw(A') = \emptyset$.
- $pDB(A') = \emptyset$.
- $tasks(A') = tasks(A)$.

El autómata A' representará a los mismos componentes que A pues el conjunto de acciones que soportan es el mismo $acts(A) = acts(A')$ y el funcionamiento interno también coincide $trans(A') = trans(A)$ y $tasks(A') = tasks(A)$.

Definición: Dado un autómata MOIRAE A' existe un autómata E/S A denominado equivalente, tal que todo componente real modelizable por medio de A' en un formalismo, lo es también por medio de A en el otro.

Desarrollo: Sea el autómata MOIRAE A' compuesto por: $sig(A')$, $names(A')$, $states(A')$, $start(S)$, $events(A')$, $trans(A')$, $throw(A')$, $pDB(A')$ y $tasks(A')$. Se define un autómata E/S A compuesto por: $sig(A)$, $states(A)$, $start(A)$, $trans(A)$ y $tasks(A)$, tal que:

- $sig(A)$ contiene los elementos:
  - $in(sig(A)) = srv(sig(A')) \cup \{(ct, c_1, c_2) \in ctrl(sig(A'))$ tal que $c_1 \notin names(A') \wedge c_2 \in names(A')\}$.

- ❍ $out(sig(A)) = req(sig(A')) \cup \{(ct, c_1, c_2) \in ctrl(sig(A'))$ tal que $c_1 \in names(A') \wedge c_2 \notin names(A')\}$.
- ❍ $int(sig(A)) = cmd(sig(A')) \cup \{(ct, c_1, c_2) \in ctrl(sig(A'))$ tal que $c_1 \in names(A') \wedge c_2 \in names(A')\}$.

❒ $states(A) = states(A') \cup ST$.

❒ $trans(A) = trans(A') \cup TR$.

❒ $tasks(A) = tasks(A')$.

Donde los conjuntos ST y TR son, respectivamente, el conjunto de estados y el de transiciones asociados al autómata de control. Dichos conjunto se calculan de la siguiente manera:

① Para todo $cs \in control\_start(A')$
entonces $cs \in ST$

② Para todo $cs \in control\_start(A')$
$\forall (s, \pi, e) \in throw(A')$
$\forall (e, s', cs') \in control\_input(pDB(A'))$
tal que $s = s' \wedge cs = cs'$
entonces $(s\pi, cs) \in TR$

③ Para todo $cs \in control\_state(A')$
si existe $(cs, \alpha) \in control\_action(pDB(A'))$
donde $\alpha = \pi_1, \pi_2, \ldots \pi_n$
entonces $cs_1, cs_2, \ldots cs_{n+1} \in ST$
entonces $\forall 0 < i \leq n\ (cs_i, \pi_{n+1}, p_{n+1} \in TR$

④ Para todo $(cs, \pi, cs') \in control\_trans(pDB(A'))$
entonces $(cs, \pi, cs') \in TR$

Para regresar de los estados de control de ST a los estados generales es necesario restringir que:

⑤ Para todo $cs \in control\_final(pDB(A'))$
para toda ejecución de control α y
todo fragmento de ejecución del autómata general β
tales que las acciones de α y β coinciden
entonces el estado final de control debe coincidir con el último estado de β.

5.6.3.1 Restricciones de la Equivalencia entre Autómatas

El mecanismo de traducción de un autómata de E/S a un autómata MOIRAE es válido para todo autómata del primer tipo posible. Sin embargo, para el caso contrario, no todos los sistemas representables por medio de un autómata MOIRAE son ajustables al formalismo expresado por Tuttle. La restricción radica en aquellas interacciones entre componentes basadas en las operaciones de control antes citadas. Tanto la propagación como la delegación, etc. son operaciones que definen nuevas secuencias de acciones no definidas en el autómata que origina el conflicto resolviéndose por la modificación dinámica de dicho autómata o por la evaluación externa de los conflictos. Ambas soluciones son sólo representables por medio autómatas de E/S que engloben ambos componentes. Debido a que los procesos de delegación y propagación no se restringen a elementos contiguos en la jerarquía de control, se deduce que sólo es posible representar un sistema formalizado mediante autómatas MOIRAE por medio de un único autómata de E/S que define el sistema completo.

5.7 Conclusiones

La característica que impide la transformación de todo autómata MOIRAE a un autómata E/S remarca la principal ventaja de la arquitectura MOIRAE al permitir la definición de sistemas cuyo control sea dinámico. Dicha faceta no es posible de realizar en otros tipos de arquitecturas, salvo en el caso del rediseño del sistema completo para incluir diversas operaciones explícitas de comunicación entre los componentes. En tal caso, se estarían mezclando las funcionalidades operacionales y las de control al mismo nivel. Como ya se remarcó en la definición del problema tal dependencia en el diseño aumenta la rigidez de los sistemas.

La arquitectura propuesta en este trabajo permite definir sistemas distribuidos, cuyas decisiones de control puedan ser modificadas dinámicamente durante la ejecución del mismo. La arquitectura descompone cada componente del sistema en funcionalidades operacionales y decisiones de control. A la hora de aplicar esta arquitectura, corresponde al diseñador del sistema la responsabilidad de dividir para cada componente cuales son tareas operacionales y cuales son las decisiones de control asociadas, implementando las primeras como código del componente y las segundas como políticas de control.

Los modelos que componen la arquitectura son una división arbitraria de las facetas que componen la arquitectura y no constituyen una metodología de diseño para estos sistemas. La tarea de diseño es de naturaleza diferente, aunque ha de considerar las características y facilidades que esta arquitectura proporciona.

Capítulo 6

APLICACIÓN AL DESARROLLO DE SISTEMAS DE DATA MINING

Índice General

6.1 Introducción

Una vez presentada la arquitectura MOIRAE como modelo para el desarrollo de sistemas distribuidos, a lo largo de este capítulo se va a presentar la aplicación de dicha arquitectura al desarrollo de un sistema distribuido de *Data Mining*. Parte de los requisitos presentados a lo largo del estudio del problema, en el Capítulo 4, se han resuelto por medio de la arquitectura MOIRAE, en concreto,

la distribución y la flexibilidad de control. El resto de características como la integración con los SGBD y la extensibilidad se han de resolver por medio del diseño adecuado de los componentes del sistema.

El nuevo sistema a diseñar es una extensión del sistema *DAMISYS*[1]. Dicho sistema abordaba la problemática de la extensibilidad de los sistemas de *Data Mining*. La nueva versión *DI-DAMISYS* (*Distributed DAMISYS*) conserva la flexibilidad en la incorporación de algoritmos de su predecesor, planteando soluciones al resto de facetas.

6.2 Visión General de la Arquitectura

El nivel de descomposición de la nueva arquitectura, pretende conseguir que los componentes del sistema alcancen la granularidad mínima posible, siempre y cuando no se comprometa con ello la eficiencia global del sistema.

Para el estudio de las funciones de un sistema, por encima del nivel de detalle alcanzado a nivel de componente, se ha definido un nivel intermedio de análisis. El primer paso de diseño ha sido la definición de una serie de grupos de componentes que realizan tareas relacionadas, denominados **federaciones de componentes**. Este concepto, tomado prestado de la terminología de agentes queda definido como:

> **Definición** FEDERACIÓN DE COMPONENTES: El grupo de componentes entre los cuales se establece una relación de cooperación mediante la cual, y en conjunto, es posible resolver una serie de tareas de alto nivel de una forma distribuida y escalable.
> Dicho grupo puede crecer o reducirse dinámicamente, añadiendo nuevos componentes de igual o distinta naturaleza o dando de baja alguno de los existentes.

De esta forma, el esquema presentado en la Figura 6.1 representa las diferentes federaciones de componentes que constituyen el sistema DI-DAMISYS:

- **Interacción con el Usuario**: Los componentes de este grupo son los encargados de facilitar al usuario el acceso a los servicios del sistema. Dentro de este grupo no sólo se encuentra el interfaz de usuario, tal y como se encuentra desarrollado en las versiones anteriores del sistema. Desde el punto de vista general, dentro de estos componentes existirán:

[1] *DAta MIning SYStem*, desarrollado por el grupo de investigación de Data Mining del Departamento de Lenguajes y Sistemas Informáticos de la Facultad de Informática [FPM+99, FMP+00].

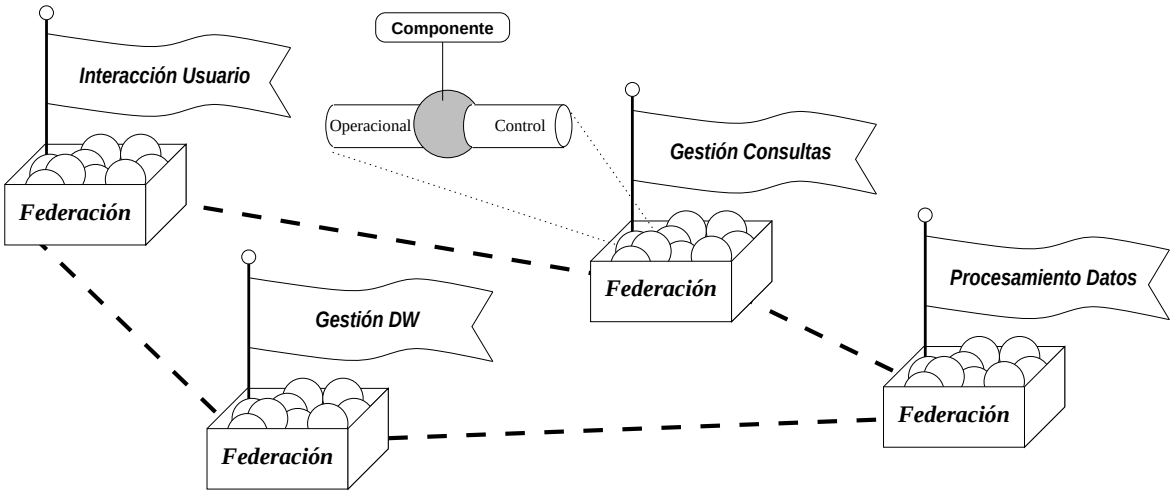


Figura 6.1: Diferentes federaciones de componentes

- ❍ Aplicaciones gráficas que van desde interfaces de usuario para acceso al sistema completos hasta componentes gráficos ligeros encargados de la visualización, consulta, monitorización de información concreta.
- ❍ Asistentes de acceso al sistema que sean capaces de tratar consultas de alto nivel expresadas de la forma más próxima al lenguaje del usuario, abstrayendo la dificultad de expresar la consulta en base a algoritmos y parámetros.

en resumen, se podría indicar que estos componentes se caracterizan por ser los que interactúan de una forma más directa con los usuarios del sistema (tanto usuarios humanos como otros sistemas).
Junto con los componentes asociados a los usuarios del sistema, dentro de esta federación existen algunos componentes internos del sistema asociados con la gestión de conexiones y programación de consultas.
Los componentes de esta federación asociados a interfaces de usuario no se encuentran desarrollados en el diseño presentado en este trabajo.

- ❒ **Gestión del *Data Warehouse*** : Como resulta habitual en los sistemas de análisis de datos (y *Data Mining*), los datos origen son recopilados de las bases de datos transaccionales de la organización. Una de las condiciones necesarias para que el proceso de análisis tenga éxito es que los datos utilizados se mantengan estables a lo largo del proceso de análisis. Esto implica, que en el caso de usar datos transaccionales (dinámicamente modificables) estos datos han de ser trasladados a una base de datos estable durante largos periodos de tiempo. Un *Data Warehouse* asegura estas condiciones, además de dar un soporte (en base a una serie de características de comportamiento) a los procesos de análisis. De esta forma,

los elementos de este grupo proporcionan los mecanismos de mantenimiento y tratamiento de la información del *Data Warehouse* usado por el sistema. Dentro de estas funciones se encuadran, entre otras:

- ❍ selección de los procesos de limpieza y preparación de datos
- ❍ detección de cambios en las fuentes de datos
- ❍ actualización de la información
- ❍ auditoría de las consultas
- ❍ ciclo de vida de los datos del sistema

Como se puede observar este grupo de componentes engloba a todas las tareas relativas a la gestión de los datos usados por el sistema. Estas tareas, como resulta evidente, implican como sujeto de las mismas los datos, pero es importante indicar que los datos, es decir los componentes que representan los datos, no se encuentran dentro de este grupo. Este conjunto de componentes se encargan únicamente de controlar y programar el proceso de actualización, renovación y observación de los datos, desde el punto de vista de su dependencia con los datos origen (situados fuera del sistema) y de su uso por parte de todo el sistema.

- ❒ **Gestión de Consultas**: Estos componentes son los encargados de planificar las acciones necesarias para tratar los datos disponibles por el sistema según las peticiones de información solicitadas por los usuarios. Estas acciones, en el caso más general son consultas de *Data Mining* y representan la ejecución de un proceso de *Data Mining* sobre un conjunto seleccionado de los datos del sistema. De todas formas, no sólo las consultas de *Data Mining* son planificadas por estos componentes, también las funciones derivadas de la preparación de datos asociados al proceso de *Data Mining* solicitado. Asimismo, las tareas de administración de las funciones de análisis son responsabilidad de estos componentes.
  Se podría concluir que, de forma similar a las funciones definidas para el grupo de componentes del punto anterior que enfocaban el dinamismo y control de los datos causado por los mismos datos (su modificación y uso), este conjunto de componentes se encargan del tratamiento de los datos inducido por las consultas solicitados por los usuarios.

- ❒ **Procesamiento de Datos**: Este grupo de componentes son los encargados del almacenamiento y tratamiento directo de los datos usados por el sistema. Las dos federaciones anteriores contienen los componentes responsables de la definición de las operaciones a realizar sobre los datos, pero es en esta última federación donde dichos planes de operaciones son ejecutados.
  Los componentes que pertenecen a esta federación son clasificables en tres categorías:

  - ❍ componentes que representan datos del sistema

- componentes que representan operaciones sobre los datos del sistema
- componentes de control, sincronización y gestión de las actividades de los dos tipos de componentes anteriores

La datos tratados por el sistema pueden encontrarse, o no, almacenados en SGBD externos. En el caso de que así fuese los componentes que representan dichos datos actúan como enlaces entre las llamadas del sistema y el SGBD. Una alternativa más integrada y por lo tanto, la recomendada en el diseño es la de dejar bajo la responsabilidad del sistema, la gestión de los datos que ha de tratar, integrando las funcionalidades como sistema gestor de bases de datos con las de *Data Mining*.
Los componentes que representan las operaciones sobre los datos se corresponderán con implementaciones de algoritmos o fases de algoritmos de *Data Mining* y demás tareas de preparación y tratamiento de datos y/o resultados, así como múltiples operaciones elementales de tratamiento de datos (operadores del álgebra relacional). Dichas operaciones serán planificadas por las federaciones de gestión y ejecutadas en esta federación de procesamiento de datos.

6.3 Federación de Interacción con el Usuario

Esta federación contiene los componentes activados por los distintos usuarios que acceden al sistema, además de ciertos componentes asociados con tareas de gestión de conexiones.

6.3.1 Componentes de la Federación

Los componentes de esta federación se encuentran representados en la figura 6.2 y se dividen en tres grupos:

- **Interfaces de usuario**.
- **Programación de consultas**.
- **Gestor de conexiones**.

6.3.1.1 Interfaces de Usuario

Se pretende no restringir el tipo de interfaces que el sistema puede disponer ni en el momento del diseño ni en etapas posteriores. En principio, se distinguen varios tipos de interfaces:

- **Interfaces de consulta de usuario**: con capacidades para asistir a usuarios no expertos o con mayor flexibilidad para permitir un mayor control de las consultas por parte de usuarios

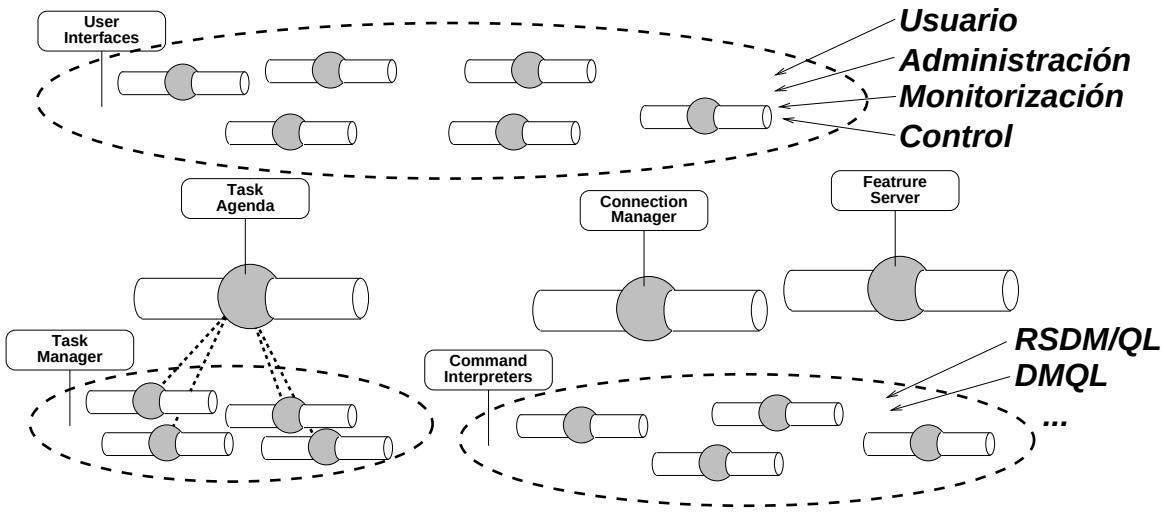


Figura 6.2: Componentes de la Federación de Interacción con el Usuario

avanzados.

- **Interfaces de administración**: orientados a la interacción del administrador con el sistema con el fin crear, modificar o eliminar usuario, datos, operaciones, etc.
- **Herramientas de monitorización**: aplicaciones usadas par representar de forma visual información relativa a las operaciones en ejecución del sistema, el uso de recursos, la topología del sistema e información similar.
- **Administrador de control**: herramienta de administración orientada a la gestión de los planos de control de los componentes y de la jerarquía de los mismos. Por medio de dicha herramienta debe ser posible actualizar el conjunto de políticas del sistema.

Una característica común a todos los componentes de este tipo es que su ejecución no es lanzada por el sistema al iniciarse, sino que es invocada por los usuarios al conectarse al mismo. La forma en la que se realiza este proceso se detalla a continuación (Sección 6.3.2.1).

La extensibilidad del sistema es uno de los principales objetivos del diseño, dicha característica debe de aplicarse también al diseño de los interfaces del mismo. De esta forma, la ampliación de las funcionalidades del sistema no debe implicar el rediseño de las herramientas de acceso al mismo.

6.3.1.2 Programación de Consultas

Una de las características de los procesos de *Data Mining* sobre datos reales es que a menudo el volumen de información a procesar es muy elevado y por lo tanto la ejecución de consultas

implica un periodo de duración considerable. Para este tipo de consultas, así como para consultas sistemáticas DI-DAMISYS proporcionará un servicio de programación de consultas encargado:

① de gestionar las consultas solicitadas *on-line* por un usuario cuando éste se desconecta del sistema sin que la consulta haya terminado de procesarse y

② de mantener una lista de consultas programadas para ejecución en determinados instantes definidos por el usuario

Para realizar estas tareas existe un componente denominado Agenda de Tareas (`Task Agenda`) encargado de registrar las peticiones pendientes, conociendo, el tipo de consulta u operación a realizar, el usuario que la solicitó, información de privilegios de la operación y el estado de la misma: en ejecución (*executing*), pendiente (*scheduled*) o ya procesada (*done*). Por cada consulta que este componente tenga en ejecución asociará un componente de tipo Gestor de Tareas (`Task Manager`) que hace las veces del usuario para este tipo de operaciones no interactivas. A dicho componente se le indica cuando ha finalizado la consulta de la que está encargado, si se ha producido algún tipo de error o incidencia y dónde se encuentran los datos resultantes, información que anotará en la Agenda de Tareas (`Task Agenda`) hasta que el usuario vuelva a conectarse o se cumplan una serie de condiciones de caducidad.

6.3.1.3 Gestor de Conexiones

La parte análoga a los componentes de interfaz de usuario a la hora de representar la conexión de un usuario al sistema se encuentra representada por los componentes de gestión de conexiones. El elemento central de este grupo es el componente Gestor de Conexiones (`Connection Manager`). Dicho componente proporciona un servicio de *login* al sistema mediante el cual cada nuevo usuario se identifica por medio de un nombre y una clave de acceso que es validada por dicho componente (consultado la información de los usuarios del sistema) generando un *stub* o resguardo para la sesión. Este *stub* contiene información relativa a:

- ❒ los privilegios de acceso del usuario,
- ❒ el nombre de usuario,
- ❒ la referencia al componente desde el cual se ha realizado la conexión,
- ❒ un identificador único de sesión y
- ❒ un código de redundancia cíclica de dicha información

El *stub* de sesión del usuario se usa en todas las operaciones como garantía de los privilegios del usuario y para evitar verificaciones redundantes de identidad para cada operación. El modelo de

autenticación delegada es muy simple, pero su actualización por un modelo más complejo puede resolverse a nivel de diferentes criterios de seguridad definibles mediante políticas de control.

Una vez identificado el usuario ante el sistema, el componente Gestor de Conexiones (`Connection Manager`) proporciona al interfaz la referencia del componente Servidor de Funcionalidades (`Feature Server`) que indica al interfaz las funcionalidades proporcionadas por el sistema, así como cuáles son los componentes que las proporcionan. Su funcionamiento se detalla a continuación.

6.3.2 Funcionamiento de la Federación

Las tareas de esta federación son tres:

① **Conexión de usuarios**.

② **Notificación de funcionalidades**.

③ **Programación de consultas**.

6.3.2.1 Conexión de Usuarios

Como ya se ha dicho los componentes de interfaz de usuario difieren del resto de componentes del sistema en que no son inicializados por el proceso de arranque del sistema, sino que a lo largo de la ejecución del mismo son creados y eliminados independientemente. Para que estos componentes sean capaces de interactuar con el sistema han de conseguir el antes mencionado *stub* de sesión, proporcionado por el componente Gestor de Conexiones (`Connection Manager`). Para localizar a este componente, los interfaces del sistema debe seguir los siguientes pasos:

① Localizar el componente central del sistema. Dicha localización es conocida inicialmente. Dicho componente denominado *Seed* es el componente superior de la jerarquía de control.

① Solicitar a dicho componente la referencia al componente Gestor de Conexiones (`Connection Manager`).

① Identificarse ante el componente Gestor de Conexiones (`Connection Manager`) por medio del nombre de usuario y clave de acceso.

① Recoger el *stub* de sesión y la referencia al componente Servidor de Funcionalidades (`Feature Server`) para la notificación de las funcionalidades del sistema.

6.3.2.2 Notificación de Funcionalidades

Con la intención de proporcionar un sistema flexible a la hora de añadir nuevas funcionalidades, es necesario definir algún mecanismo de identificación de las mismas entre los componentes

que las proporcionan y los que han de utilizar (los interfaces de usuario). Esta tarea se realiza por medio del Servidor de Funcionalidades (`Feature Server`).

La interacción con dicho componente por parte de los elementos que proporcionan cada una de las funcionalidades del sistema consiste en registrar dicha funcionalidad por medio de:

- ❒ un identificador a la que se asocia
- ❒ una descripción de dicha funcionalidad y
- ❒ una referencia al componente que la proporciona

Los interfaces de usuario al conectarse al sistema, solicitan del Servidor de Funcionalidades (`Feature Server`) la lista de identificadores de funcionalidades disponibles. Por cada funcionalidad soportada por el interfaz se reconocerá el identificador y se iniciará el diálogo entre el interfaz y el componente que la proporciona para definir los parámetros de la misma.

De esta forma, si por ejemplo se habilita una funcionalidad para realizar análisis estadístico de los datos, el componente que proporciona dicha funcionalidad se da de alta con un identificador determinado. Por otro lado, todo interfaz que soporte dicha funcionalidad reconocerá el identificador y permitirá al usuario solicitar dichas operaciones.

Este tipo de mecanismos de descubrimiento dinámico de servicios del sistema es muy adecuado para el desarrollo de interfaces configurables, en base a elementos del tipo *Java Beans* que construyan paletas de operaciones disponibles en tiempo de ejecución.

Los componentes encargados de registrar las funcionalidades, por lo general pertenecerán tanto a la Federación de Gestión del *Data Warehouse* como de Gestión de Consultas. Una excepción son los componentes asociados a la interpretación de comandos. Dichos componentes implementan analizadores para gramáticas de interrogación a sistemas de *Data Mining* o *Data Warehousing*, tales como RSDM/QL [FPM+99, Lóp99] o DMQL de Han[Mal97] . Estos componentes se encuentran situados en esta misma federación y se encargan de traducir consultas expresadas en dichos lenguajes de interrogación en peticiones a los componentes que realizan dichas operaciones.

6.3.2.3 Programación de Consultas

La tercera tarea proporcionada por esta federación es la ejecución de consultas fuera de línea (*off-line*). Mediante este servicio un usuario puede realizar una consulta sin estar conectado durante la ejecución de la misma, encargando al sistema la ejecución de la misma y del almacenamiento

de los resultados hasta que se conecte de nuevo al sistema.

Este servicio proporcionado por la Agenda de Tareas (`Task Agenda`) se registra en el Servidor de Funcionalidades (`Feature Server`) igual que todos los servicios ofertados por el sistema. Los interfaces de usuario que puedan interactuar con él podrán editar los registros de dicho usuario, declarando operaciones a ejecutar en instantes determinados o cada ciertos intervalos de tiempo.

Las acciones declaradas en este componente se activarán por medio de un reloj que arrancará la ejecución de un componente de tipo Gestor de Tareas (`Task Manager`) para cada entrada activada que se conectará con el componente asociado a la operación indicada, esperará la finalización de la misma y actualizará la entrada en la Agenda de Tareas (`Task Agenda`) con los resultados obtenidos.

Los criterios de gestión del servicio de programación de consultas, tales como:

- número de operaciones se pueden programar
- prioridades de las operaciones
- operaciones se pueden programar y cuáles no
- criterios de eliminación de los resultados o entradas (criterios de caducidad de resultados)

se definen por medio de políticas de control del componente y por lo tanto son fácilmente configurables sin necesidad de rediseñarlo.

6.4 Federación de Gestión del *Data Warehouse*

Esta federación incluye todos los componentes encargados de la planificación de las operaciones de gestión de los datos usados por el sistema. Los datos usados para un proceso de *Data Mining* maduran a lo largo de un proceso de ciclo de vida que es controlado por medio de los servicios proporcionados por estos componentes.

6.4.1 Componentes de la Federación

Los componentes de esta federación se muestran en las figuras 6.3 y 6.4, clasificándose en seis grupos:

- **Ciclo de vida de los datos**.
- **Limpieza de los datos**.

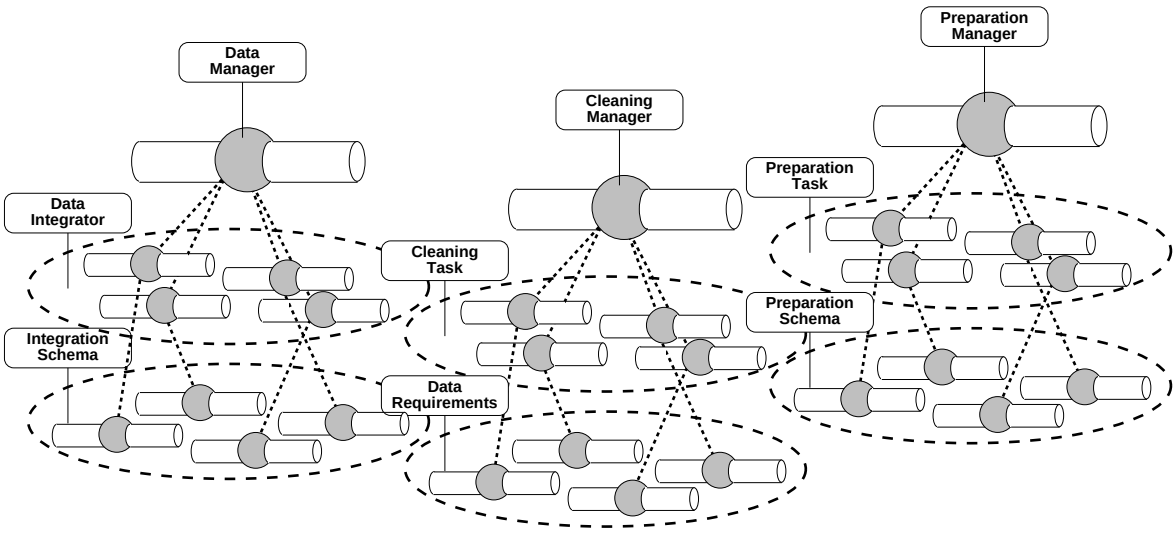


Figura 6.3: Componentes de la Federación de Gestión del *Data Warehouse* (Ciclo de Vida de los Datos)

- ❒ **Preparación de los datos**.
- ❒ **Sentencias de administración**.
- ❒ **Monitorización de las fuentes de datos**.
- ❒ **Auditoría del sistema**.

6.4.1.1 Ciclo de Vida de los Datos

Los datos que se incorporan al sistema han de cumplir una serie de etapas antes de que se puedan utilizar. Estas etapas se controlan por un componente denominado Gestor de Datos (`Data Manager`). Este componente proporciona las funcionalidades básicas del ciclo de vida de los datos que son su creación y su eliminación.

El proceso de creación de los datos se realiza por medio de la importación desde fuentes de datos externas. Por lo general, una tabla de datos vista desde la perspectiva de un sistema de *Data Mining* proviene de varias tablas de datos originales. El proceso de combinación de estas tablas de datos se denomina integración y por cada uno de estos procesos se activa un componente denominado Integrador de Datos (`Data Integrator`). Dicho componente controla el proceso de integración de una tabla de datos a partir de los datos externos del sistema.

La tarea de eliminación de datos también requiere una serie de pasos, dependientes de ciertos criterios de uso del sistema. Los datos retirados pueden ser o no almacenados como datos

históricos y la información asociada a los mismos (consultas realizadas, estadísticas de uso y tablas derivadas) puede ser o no eliminada.

Todas las operaciones del ciclo de vida de los datos (no sólo su creación y eliminación) se encuentran registradas por el componente Gestor de Datos (`Data Manager`). Esto incluye también las tareas proporcionadas por el resto de componentes de la federación.

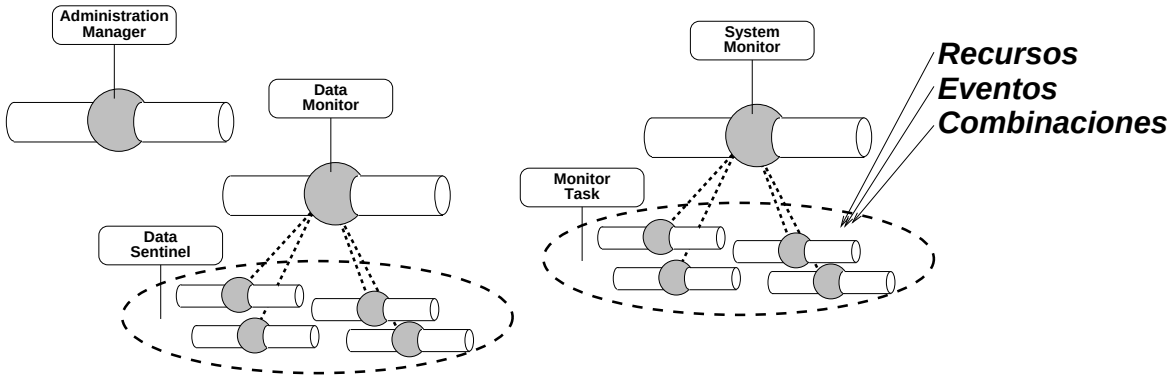


Figura 6.4: Componentes de la Federación de Gestión del *Data Warehouse* (Administración y Monitorización)

6.4.1.2 Limpieza de los Datos

Por lo general, los datos una vez integrados contienen múltiples problemas de consistencia derivados del proceso de combinación de las fuentes de datos. Este problema no se encuentra presente únicamente en los casos de procesos de integración muy simples o en conjuntos de datos previamente analizados para algún otro fin.

La tarea de limpieza de los datos cargados en el sistema se gestiona por medio del componente Gestor de Limpieaz (`Cleaning Manager`). Éste delega cada una de las tareas de limpieza de datos en un componente de la clase Tarea de Limpieza (`Cleaning Task`). Estos componentes son los encargados de gestionar un proceso de limpieza en particular sobre un conjunto de datos determinado. Las tareas que dicho proceso realiza pueden incluso requerir ciertas funcionalidades de la Federación de Gestión de Consultas. Las operaciones emprendidas por el proceso de limpieza se asocian con lo que se han denominado *data requirements* que son una serie de condiciones que los datos han de cumplir, una vez que han sido limpiados. El objetivo de cada uno de los procesos de limpieza es satisfacer estas restricciones de calidad de los datos.

6.4.1.3 Preparación de los Datos

Otra operación perteneciente al ciclo de vida de los datos en el sistema es la preparación de los mismos, entendiendo como tal la generación de información adicional sobre la tabla previamente limpiada orientada a su posterior uso en procesos de consulta.

El componente Gestor de Preparación (`Preparation Manager`) es el encargado de proporcionar las funcionalidades de preparación de los datos. Por cada proceso de preparación este componente ejecuta un componente denominado Tarea de Preparación (`Preparation Task`). Estos componentes son los encargados de interpretar una información denominada *preparation schema* que representa las operaciones asociadas al proceso de preparación de los datos solicitado. A una misma tabla de datos se le pueden aplicar diferentes operaciones de preparación (es decir diferentes *preparation schemas*).

6.4.1.4 Sentencias de Administración

Además de los datos, el sistema gestiona otra serie de entidades que necesita para su funcionamiento. La creación, modificación y eliminación de estas otras entidades es procesada por el componente Gestor de Administración (`Administration Manager`). Dicho componente se encarga de verificar las peticiones realizadas en relación a privilegios, dependencias y otras restricciones para su aplicación. Si la sentencia es ejecutable, entonces ésta se lleva a cabo dentro de la Federación de Procesamiento de Datos.

6.4.1.5 Monitorización de las Fuentes de Datos

Los datos una vez son cargados en el sistema desde las fuentes de datos originales no se modifican dentro del sistema, sin embargo, fuera de éste, los datos pueden seguir siendo actualizados por los procesos transaccionales que los usan. Aunque los datos han de ser estables durante los procesos de consulta, en ciertos entornos interesa actualizar los datos a analizar a medida que se modifican los datos originales.

Los criterios bajo los cuales se actualizan los datos del sistema pueden fijarse en base a condiciones temporales, definiendo la periodicidad de los procesos de actualización. Otros criterios para este proceso se definen en base a condiciones en los datos originales, por ejemplo un cambio de más de un 10% de los datos o cuando más de 5000 tuplas sean modificadas. Estas condiciones de actualización requieren que existan ciertos componentes denominados Centinelas de Datos (`Data Sentinels`) asociados a cada fuente de datos que monitorice cuando se disparan estas condiciones.

Todos los Centinelas de Datos (`Data Sentinels`) conocen las condiciones que han de vigilar y se comunican con un único componente denominado Monitor de Datos (`Data Monitor`) que es el encargado de destinarlos a las diferentes localizaciones y de asegurar que dichos procesos de monitorización se realizan con éxito, y en el caso de que alguna condición se active es el encargado de realizar las tareas adecuadas de actualización, usando para ello el resto de componentes de la federación.

6.4.1.6 Auditoría del Sistema

El último grupo de componentes de la federación no tiene un uso dirigido a los datos únicamente. Se encargan de evaluar el estado del sistema mediante una serie de parámetros y estadísticas de uso y demás muestras de utilización de los recursos y elementos internos del sistema. El componente Monitor de Sistema (`System Monitor`) es el encargado de registrar este servicio del sistema y en base a la configuración del mismo ejecutará una serie de componentes denominados Tareas Monitor (`Monitor Task`) encargados de la realización de cada una de las tareas de auditoría identificadas.

Cada uno de estos componentes de monitorización del sistema puede estar asociado a:

- ❒ recursos del sistema (nodos de ejecución, componentes),
- ❒ eventos dependientes del tiempo (*snapshots* del sistema),
- ❒ eventos dependientes de las operaciones (activados cuando un determinado componente realiza una determinada acción)
- ❒ sucesos dentro del plano de control del sistema o
- ❒ agregación, coordinación y gestión de otros Monitores de Sistema (`System Monitor`) (predicados asociados a varios eventos simultáneos).

Estos componentes gestionan, por ejemplo los *logs* generados por el sistema durante su ejecución, el tipo y periodicidad de ciertas operaciones y el uso computacional de un determinado nodo.

6.4.2 Funcionamiento de la Federación

Dentro de las tareas realizadas por esta federación, a continuación se detallan las siguientes:

① **Gestión del catálogo de datos**.

② **Integración de los datos**.

③ **Limpieza y preparación de los datos**.

6.4.2.1 Gestión del Catálogo de Datos

Toda la información en relación a los datos almacenados por el sistema, es decir aquellos datos usados en las consultas está gestionada en esta federación. Esta información en relación a los datos del sistema se denomina *catálogo de datos*. La representación física de esta información se almacena en una serie de tablas residentes dentro de los elementos de la Federación de Procesamiento de Datos, aunque su gestión se dirige dentro de la Federación del *Data Warehouse*.

El catálogo de datos mantiene información relativa al ciclo de vida de los datos en el sistema y demás información relevante de los mismos:

- ❒ información de los datos (número de atributos, dominios, número de tuplas, etc.),
- ❒ fuentes de datos de la integración,
- ❒ parámetros de integración,
- ❒ operaciones de limpieza y preparación realizadas,
- ❒ otras tablas de datos derivadas y
- ❒ estadísticas e información de uso de los datos

Cualquier operación de creación, modificación o eliminación de datos afecta a dicho catálogo y a este nivel se define cómo se modifican dichos datos. El uso de dicha información está disponible para el resto de componentes del sistema, los cuales solicitará a la Federación de Procesamiento de Datos la referencia a las tablas del catálogo de datos para consultar la información que necesite.

El catálogo de datos es parte de un conjunto de tablas más extenso denominada *catálogo del sistema* que además de información relativa a los datos mantiene un conjunto de tablas en relación al resto de entidades del sistema (usuarios, operaciones, componentes, ...). Estas otras tablas son las usadas en la ejecución de las sentencias de administración del sistema (por medio del Gestor de Administración (`Administration Manager`)).

6.4.2.2 Integración de los Datos

El proceso de integración de datos es entendido en este contexto como la combinación de varias fuentes externas de datos para crear una nueva tabla de datos dentro del sistema. Dicho proceso está compuesto de diferentes operaciones:

① Localización de cada una de las tablas externas y obtención del acceso para leerla.

② Definición de los datos relevantes de cada una de las tablas externas y su representación como atributos finales de la tabla creada.

③ Combinación de tablas externas en base a campos clave, requiriendo posibles adaptaciones y transformaciones para ser equiparables.

④ Operaciones de traducción y conversión de valores así como de agregación de registros en base a contadores, medias, máximos u otros valores estadísticos.

⑤ Almacenamiento de la tabla integrada dentro del sistema.

El proceso completo está dirigido por cada Integrador de Datos (`Data Integrator`). Las tareas ① y ⑤ se realizan por componentes dentro de la Federación de Procesamiento de Datos asociados a tablas de datos (externas y locales, respectivamente). La tarea ② se define en base a un componente que se denominan Esquemas de Integración (`Integration Schema`) que contiene la información del proceso de integración. Las tareas ③ y ④ son realizadas también dentro de la Federación de Procesamiento de Datos por medio de operadores relacionales y de integración dentro del conjunto de componentes de procesamiento de datos.

6.4.2.3 Limpieza y Preparación de los Datos

Las operaciones de limpieza y preparación de datos conservan una relativa similitud con la ya vista de integración. Estas operaciones son controladas por un componente dentro de esta federación, existiendo otro componente que representa los parámetros de la operación. Todas las tareas de manipulación y acceso de los datos se transmiten para que la Federación de Procesamiento de Datos las realice.

En el caso de la operación de limpieza:

❒ El componente de control de cada operación de limpieza se denomina Tarea de Limpieza (`Cleaning Task`).

❒ Los parámetros de la operación los contiene un componente de tipo Requisitos de Datos (`Data Requirements`). Estos componente representan restricciones que han de cumplirse por parte de los datos limpios, tales como eliminación de datos redundantes, tuplas con gran número de valores nulos, etc.

En el caso de las operaciones de preparación de los datos:

❒ Un componente de tipo Tarea de Preparación (`Preparation Task`) controla la ejecución de cada operación.

- Uno o varios Esquemas de Preparación (`Preparation Schema`) son proporcionados para modelizar el modelo de preparación de los datos. La información contenida en estos componentes puede ser desde operaciones asociadas a la creación de tablas derivadas (agregación y cálculo de parámetros estadísticos) hasta la definición de modelos de *Data Warehousing* tales como el modelo en estrella (*star model*) o en copo de nieve (*snowflake model*).

6.5 Federación de Gestión de Consultas

Esta federación realiza las tareas análogas a la de Gestión del *Data Warehouse*, pero para las peticiones relacionadas con la aplicación de consultas a los datos del sistema. Básicamente existen dos procesos de interrogación del sistema, por un lado se encuentran las consultas relacionales tradicionales, que han de estar soportadas por el sistema, pues actúa como un SGBD restringido. Por otro lado las consultas para el descubrimiento de información o consultas de *Data Mining* también se encuentran gestionadas por esta federación.

6.5.1 Componentes de la Federación

La figura 6.5 muestra los componentes de esta federación que se encuentran divididos en los siguientes grupos:

- **Gestión de consultas relacionales**.
- **Gestión de consultas de *Data Mining*** .
- **Componentes comunes**.

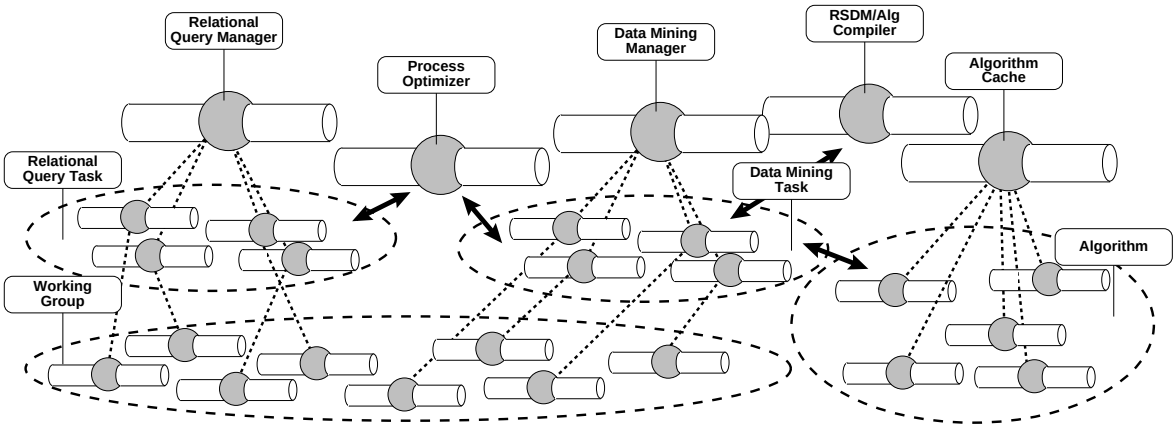


Figura 6.5: Componentes de la Federación de Gestión de Consultas

6.5.1.1 Gestión de Consultas Relacionales

Los servicios de consultas relacionales se encuentran proporcionados por el componente Gestor de Consultas Relacionales (`Relational Query Manager`) el cual registra en el componente Servidor de Funcionalidades (`Feature Server`) de la Federación de Interacción con el Usuario las consultas relacionales que soporta. Se definen cinco distintos niveles de consultas relacionales:

① Selecciones y proyecciones (cláusula `WHERE`).

② Cálculo de agregados y cláusula `HAVING`.

③ Ordenaciones (cláusula `ORDER BY`).

④ Uniones naturales (*joins*) y otras operaciones de combinación de relaciones (tales como `UNION`, `INTERSECT`, `...`).

⑤ Sub-*queries* y consultas anidadas (`EXISTS`, `NOT EXISTS`, `IN`, `...`).

El servidor proporcionará hasta un nivel de consultas determinado dependiendo de cuáles sean las funcionalidades implementadas.

Por cada consulta recogida por el Gestor de Consultas Relacionales (`Relational Query Manager`) se ejecuta un componente del tipo Tarea de Consulta Relacional (`Relational Query Task`) encargado de la coordinación de las tareas de cada una de las consultas. Estos componentes se dividen en diferentes subclases dependiendo del nivel de servicio proporcionado.

6.5.1.2 Gestión de Consultas de *Data Mining*

De forma análoga a las consultas relacionales, existe un componente denominado Gestor de *Data Mining* (`Data Mining Manager`) que recoge las peticiones solicitadas y arranca un componente Tarea de *Data Mining* (`Data Mining Task`) por cada consulta recibida. Las operaciones de *Data Mining* llevan también asociada una tipología, dividida en:

① Extracción de modelos.

② Derivación de datos (pre-procesamiento de datos).

③ Validación y aplicación de modelos.

④ Fusión y tratamiento de modelos.

A diferencia del caso de las consultas relacionales, esta tipificación no está asociada a una subclasificación de los componentes encargados de la realización de las operaciones, pues una consulta de *Data Mining* puede representar un proceso suficientemente complejo como para incluir

varias de estas operaciones.

Para describir los pasos de una consulta, el algoritmo que se desea aplicar sobre los datos se define por medio de un lenguaje denominado *RSDM/Alg* en notación procedimental. Este lenguaje de descripción de algoritmos es el ya utilizado por el sistema *DAMISYS* para describir dichos algoritmos.

6.5.1.3 Componentes Comunes

Una vez en ejecución las consultas, tanto relacionales como las de *Data Mining*, se descomponen en fragmentos en base a una serie de criterios de interdependencia y sincronización. Cada una de estas sub-operaciones es ejecutada por medio de un componente del tipo Grupo de Trabajo (`Working Group`). Este tipo de componentes es común tanto para un tipo de consultas como para el otro. Por cada consulta se pueden definir de uno a varios componentes de este tipo. Estos componentes son los encargados de interaccionar directamente con los componentes de la Federación de Procesamiento de Datos.

6.5.2 Funcionamiento de la Federación

En detalle se presentan las siguientes tareas relevantes del sistema:

① **Algoritmos en *RSDM/Alg***.

② **Ejecución de consultas**.

6.5.2.1 Algoritmos en *RSDM/Alg*

Las operaciones de consulta de *Data Mining* son muy abiertas y la flexibilidad de un sistema de *Data Mining* se mide en la posibilidad de definir de una forma sencilla nuevas operaciones de análisis de datos, por medio de las operaciones o algoritmos definidos hasta el momento o definiendo nuevas operaciones y algoritmos. Para ello en *DI-DAMISYS*, y ya incluso la versión anterior *DAMISYS*, se optó por dividir los algoritmos de *Data Mining* en operaciones elementales definidas como *operadores*. Cada uno de estos operadores no es más que una operación atómica que toma como entrada una o varias tablas relacionales y genera una o varias tablas del mismo tipo. La forma en la cual se enlazan estos operadores se encuentra definida por medio del lenguaje de descripción de algoritmos *RSDM/Alg*. La sintaxis y funcionales de este lenguaje no son objeto de estudio en este trabajo y se referencia para un estudio más detallado a [Mar99, FMP+00].

Los algoritmos en *RSDM/Alg* son ficheros de texto cuyo uso por el sistema requiere un proceso de interpretación. Dicho proceso se denomina compilación del algoritmo y se realiza por medio

del componente Compilador de RSDM/Alg (`RSDM/Alg Compiler`), el cual accede a las descripciones de algoritmos en este lenguaje y genera una representación interna en forma de un componente denominado Algoritmo (`Algorithm`). Este componente se encuentra asociado a cada uno de las Tareas de *Data Mining* (`Data Mining Task`) los cuales son los encargados de instancias las variables de entrada del algoritmo y aplicarlo a los datos solicitados.

En ciertas ocasiones, el proceso de compilación del algoritmo no es necesario, si ésta ya ha sido realizado recientemente alguna vez. En estos casos se dispone de una *cache* de algoritmos compilados encargada de servir los componentes Algoritmo (`Algorithm`) que se encuentren registrados. Esta *cache* de algoritmos se denomina `Algorithm Cache`.

6.5.2.2 Ejecución de Consultas

La ejecución de las consultas se resume en la aplicación de componentes de proceso sobre componentes de datos dentro de la Federación de Procesamiento de Datos. Sin embargo, la aplicación de unos componentes sobre los datos de otros debe encontrarse controlada. Dentro de la Federación de Procesamiento de Datos existen mecanismos de control de estas operaciones, pero su ámbito de control es local y carece de la perspectiva del proceso completo a realizar. Por ello, por encima de las funcionalidades de control de dicha federación, dentro de ésta se definen elementos de control para la ejecución de consultas. Estos elementos se encuentran representados por medio de los componentes Grupo de Trabajo (`Working Group`). Para una consulta compleja se aplican una serie de criterios para dividir la operación en una serie de tramos de ejecución que se asocian a diferentes componentes Grupo de Trabajo (`Working Group`). Estos criterios definen los puntos de división dentro del código de un algoritmo que separan cada uno de los diferentes tramos. Dichos puntos de ruptura se caracterizan por propiedades tales como:

- ❒ Tratarse de resultados intermedios reutilizables en otros procesos.
- ❒ Coincidir con dependencias entre datos, es decir tablas que un tramo de ejecución tiene que generar completamente antes de que el siguiente pueda comenzar a ejecutar.
- ❒ Consideraciones topológicas o de ubicación de datos u operadores.
- ❒ Criterios de equilibrio de carga o consumo de recursos.

Cualquiera de estos criterios o la combinación e inclusión de nuevos criterios pueden ser definidos por medio de las políticas de control de las Tareas de *Data Mining* (`Data Mining Task`) o Tareas de Consulta Relacional (`Relational Query Task`). De esta forma se pueden definir más o menos componentes Grupo de Trabajo (`Working Group`) con mayor o menor número de operaciones dependiendo de decisiones fácilmente configurables.

Uno de los posibles criterios de definición de Grupos de Trabajo (`Working Group`) es que la operación a realizar, por ejemplo haya sido realizada anteriormente y su resultado se encuentre almacenado. Esta información y cualquier otra relativa a decisiones de optimización global de todas las consultas en ejecución del sistema se encuentra centralizada en un componente denominado Optimizador de Proceso (`Process Optimizer`). Con este componente se comunican todos los elementos de control de consultas (`Data Mining Task` y `Relational Query Task`) para negociar y aplicar criterios de optimización global. Este mecanismo permite no repetir esfuerzos a la hora de ejecutar consultas que en todo o en parte coinciden.

6.6 Federación de Procesamiento de Datos

Como soporte a las acciones planificadas por los componentes de las federaciones anteriores, en esta federación se localizan los componentes capacitados para realizar dichas acciones. Estos componentes proporcionan en conjunto funciones de tratamiento directo de los datos accesibles por el sistema. Esto quiere decir que sólo estos componentes son capaces de localizar los datos, acceder a ellos, aplicar las tareas de transformación, análisis o exploración planificadas y de almacenar nuevos resultados si esto fuese conveniente. En consecuencia, estos componentes representan la mayor carga computacional y de recursos del sistema, pues si bien el número de funcionalidades generales es más bien ligeramente menor que el de cualquiera de los otras federaciones de componentes, el volumen de información a los que estas funcionalidades son aplicadas es muy importante (del orden de Mb a Gb de datos). Esta característica requiere que los métodos de control definidos por las políticas estén orientados al uso eficiente de los recursos de acuerdo con los criterios de uso generales del sistema (menor tiempo de respuesta, más consultas en paralelo).

6.6.1 Componentes de la Federación

La figura 6.6 representa los principales componentes de la federación y como se encuentran desplegados. Estos componentes se dividen en tres grupos:

- **Componentes del bus de procesos**.
- **Componentes del bus de datos**.
- **Control de operaciones**.

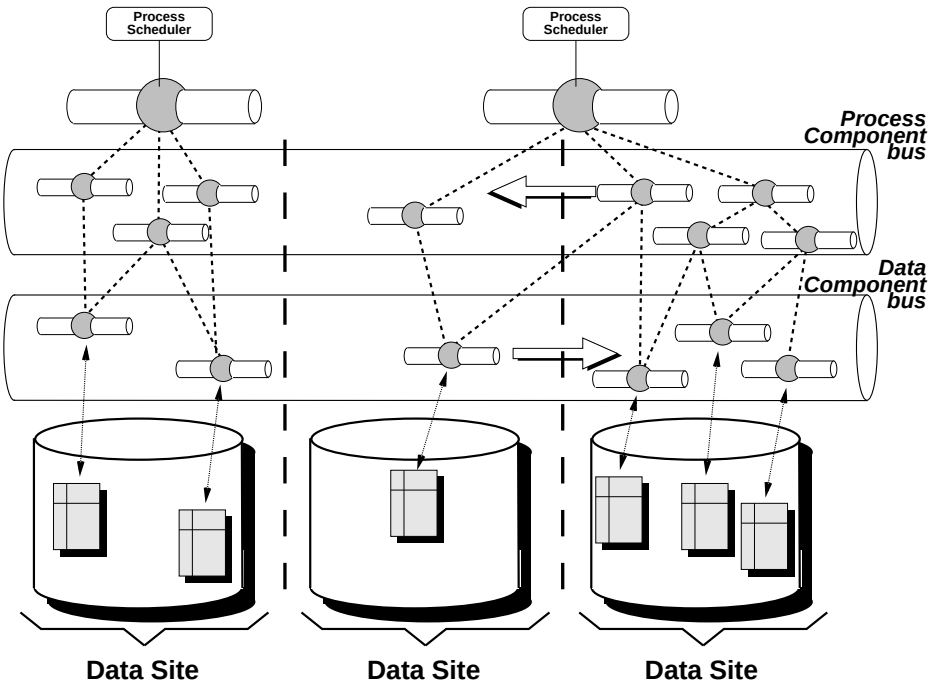


Figura 6.6: Componentes de la Federación de Procesamiento de Datos

6.6.1.1 Componente del Bus de Procesos

El conjunto de todas las operaciones tanto relacionales como partes de algoritmos de *Data Mining* se encuentran representados por medio de **operadores**. Estos elementos representan cada una de las operaciones elementales en las que se puede descomponer cualquier proceso de tratamiento de datos. Cada operador está asociado con una o más tablas de datos a la entrada y una o más tablas de datos a la salida.

Todos los operadores forman parte de un bus lógico definido entre los diferentes nodos del sistema. Este bus se denomina **bus de componentes de proceso** (*Process Componente Bus*) y define el conjunto de nodos o *Data Sites* donde se puede ejecutar un determinado operador. Los operadores son entidades que dentro de este bus pueden desplazarse de un nodo a otro dependiendo de una serie de criterios y restricciones.

6.6.1.2 Componente del Bus de Datos

Dentro la arquitectura MOIRAE no existe ningún elemento que represente datos pasivos (como pueden ser tuplas de una base de datos), ya que el elemento atómico que constituye la arquitectura es el componente. Esto hace que para poder acceder a los datos dentro de la arquitectura propuesta estos tengan que estar recubiertos de una capa que los adapte a la estructura de un

componente. Esta capa proporcionará funcionalidades tales como ordenación, selección o proyección entre otras. Esto hace que los datos se comporten como un componente más del sistema al cual otros componentes pueden solicitar servicios.

Evidentemente, si bien cada tabla de datos que se encuentra a disposición del sistema ha de encontrarse representada por un componente, sólo existe un reducido número de tipos de componentes. Estos tipos de componentes se encuentran clasificados por el soporte donde se encuentran almacenados los datos, es decir por el mecanismo de almacenamiento físico usado, tales como un gestor de base de datos (Oracle, Sybase, DB2, ...) o formato de almacenamiento en el caso de bloques directamente gestionados por el sistema. El uso de componentes de datos asociados a gestores relacionales no está pensado para el procesamiento de datos dentro del sistema sino para el proceso de importación y exportación de datos desde o hacia bases de datos externas. Como alternativa, se sugiere que los datos dentro del sistema se encuentren gestionados por éste. De esta forma, los datos están almacenados directamente en un soporte de almacenamiento con el formato que directamente le interese al sistema. Junto con los datos, se almacenarán las estructuras de ordenación y demás información de organización (índices, árboles de búsqueda, etc.) que interesen. Una tabla de datos asociada a determinados bloques de disco es representada por un componente cuando el sistema interaccione con ella.

No resulta necesario tener activados todos los componentes asociados a todas las tablas de datos posibles, esta activación puede ser realizada bajo demanda según sean requeridos para una consulta y posteriormente serán desactivados. Sólo existirá un componente por cada tabla de datos en uso, de forma que si dos procesos de consulta usan la misma tabla, ambos procesos interactuarán con el mismo componente, que será el encargado de mantener la integridad de dicha tabla de datos.

Al tratarse de componentes asociados a volúmenes de información grande, la posibilidad de cargar todos los datos de una tabla en espacio de memoria del componente es, en la mayoría de casos, impracticable. En su lugar es más adecuado optar por gestionar páginas de datos en memoria que se cargarán con las tuplas originales según dejen de ser accedidas. El control sobre el tamaño y número de dichas páginas se definirá en base a las políticas del plano de control, dependiendo de los recursos asignados a cada consulta.

Al igual que los componentes de proceso, los componentes de datos están desplegados sobre un bus lógico denominado **bus de componentes de datos** o *Data Component Bus*. Dicho bus de datos permite la migración de componentes de datos de un nodo a otro, de forma que si el con-

junto de datos (tabla) representado por dicho componente es accedido por un componente determinado, se pueda trasladar al mismo computador de forma que se minimice la comunicación vía red. Esta propiedad de movilidad a la hora de Componentes de Datos requiere que la tabla se encuentre también en el computador destino. Lo que implica que la copia de dicha tabla esté en dicho computador. En el caso de que una tabla este replicada múltiples veces en distintos nodos, es necesario definir políticas de coherencia entre las copias, en el caso de que dicha información sea modificada.

6.6.1.3 Control de Operaciones

Además de los componentes de proceso y datos, esta federación dispone de un tercer tipo de componentes, encargados de controlar las operaciones realizadas por los otros dos tipos. Los componentes de control definidos en esta federación son dos:

① **Planificador de Procesos**: Encargado de coordinar una secuencia de operaciones asociada o bien con un fragmento de una consulta o con una operación de administración. Este componente desconoce el objetivo del proceso que realiza y su tarea se centra en la sincronización de los diferentes componentes de proceso y de la negociación de la ubicación de dichos componentes. Este componente pertenece al tipo Planificador de Procesos (`Process Scheduler`).

② **Controlador del Buses**: Asociados a cada bus de la federación y por cada nodo o *Data Site* existe un componente de control para dicho bus. Estos componentes son los encargados de dirigir los procesos de ejecución de los componentes dentro de los buses, así como de los procesos de migración de un nodo a otro. La información usada por estos componentes en base a la cual negocian la ejecución de componentes es el uso de recursos asociado a cada nodo y la carga de proceso actual del nodo.

6.6.2 Funcionamiento de la Federación

Dentro de las funciones realizadas por esta federación se va a centrar la atención en dos de ellas:

① **Ejecución de operaciones**.

② **Equilibrio de carga**.

6.6.2.1 Ejecución de Operaciones

El proceso de ejecución dentro de esta operación se inicia cuando un componente de control en alguna de las otras federaciones desea ejecutar una serie de operaciones sobre unos datos. Para ello, este componente de control ejecuta uno o varios componentes del tipo Planificador de Pro-

cesos (`Process Scheduler`). La información suministrada a estos componentes para que ejecuten cada uno de los fragmentos de la operación es:

- ❒ Operadores a ejecutar (componentes del bus de proceso).
- ❒ Tablas de datos usadas como datos de entrada (componentes del bus de datos).
- ❒ Nuevas tablas a generar o tablas a modificar.
- ❒ Esquema de interacción entre operadores (cómo los operadores se interconectan entre si por medio de tablas de datos).

La descripción completa de la información gestionada por estos componentes se denomina *cadena de operadores*.

Una vez que el Planificador de Procesos (`Process Scheduler`) conoce la cadena a ejecutar, éste determina dónde se han de ejecutar los operadores, para ello negocia con los diferentes gestores del bus de proceso de los diferentes nodos en base a unas restricciones que el componente de control que lo creó puede haberle indicado. El grado de libertad que este componente puede tener en la planificación de la consulta vendrá determinado por la rigidez de las restricciones que se le hayan proporcionado, para ciertos casos puede ser aconsejable definir dónde se ejecutarán las operaciones desde la perspectiva del componente de control de la federación de gestión correspondiente.

Una vez que los componentes de proceso en el bus están en ejecución, el Planificador de Procesos (`Process Scheduler`) activa los componentes del bus de datos asociados a las tablas de datos indicadas. Esto incluye la creación de las tablas de salida adecuadas. Una vez definidos ambos tipos de componentes, estos se enlazan entre sí y se comienza la ejecución. Cuando ésta finaliza se notifica al componente de control superior y se eliminan los resultados intermedios o finales que no interese mantener.

6.6.2.2 Equilibrio de Carga

Durante el tiempo de funcionamiento del sistema, se pretende que éste sea capaz de ejecutar múltiples operaciones en paralelo. Para poder realizar tal objetivo de una forma eficiente es indispensable gestionar los recursos del sistema de forma adecuada. La Federación de Procesamiento de Datos es sin duda la más exigente a este respecto pues la aplicación de operaciones sobre grandes volúmenes de datos y el almacenamiento de los mismos en memoria es mucho más costoso que el resto de operaciones del sistema.

Una herramienta para gestionar eficientemente los recursos es el **equilibrado de carga**. Se denota como tal, a la estrategia para ubicar las tareas en diferentes nodos de forma que los recursos asignados a cada tarea en cada nodo sean proporcionales y equitativos. La proporcionalidad y la equitatividad son dos conceptos muy imprecisos y su aplicación puede depender de consideraciones generales del rendimiento del sistema, tales como fomentar la ejecución de un mayor número de consultas a la vez o reducir el tiempo de ejecución de las consultas más prioritarias. Con la intención de no restringir los criterios de explotación de los recursos del sistema, el sistema *DIDAMISYS* propone la definición de las estrategias de equilibrado de carga en base a las políticas de control.

Una vez definidos los criterios que rigen el equilibrio de carga, estas decisiones afectan a la ejecución de operaciones de tres formas:

- ❒ Restricciones de localización.
- ❒ Congelación de la ejecución.
- ❒ Migración de tareas.

Las **restricciones de localización** afectan a la hora de arrancar los componentes de proceso y los datos asociados. Cuando el Planificador de Procesos (`Process Scheduler`) negocia con cada uno de los controladores de los buses, el estado de carga y los recursos disponibles en cada nodo restringen la ejecución de nuevas tareas que entrarían en conflicto con las ya en ejecución en el nodo.

La **congelación de la ejecución** consiste en detener una o más tareas actualmente en ejecución, para reasignar sus recursos a otras tareas del nodo. Esta opción por lo general es la última alternativa usada pues implica bloquear la ejecución de ciertas consultas.

La opción más compleja es la **migración de tareas**. Esta operación consiste en detener la ejecución de una operación momentáneamente, almacenar su estado y activar los componentes que la realizaban en otra localización para que una vez restaurado el estado puedan continuar con la operación. La capacidad para salvaguardar y restaurar el estado de los componentes de proceso determina la posibilidad o no de realizar esta operación. Como alternativa existe la migración destructiva que consiste en comenzar de nuevo la ejecución de la operación en el nuevo destino de los operadores.

Una de las mayores restricciones a la hora de realizar el reparto de carga es la ubicación física de los datos usados en las operaciones. No se obtiene ningún beneficio al desplazar un proceso

costoso de cómputo a un nodo que se encuentre libre si dicho proceso hace uso de unos datos almacenados en otro nodo. La interacción remota con los componentes de datos es una de las operaciones menos eficientes. Los componentes de datos están diseñados para usar mecanismos de memoria compartida en la interacción con otros componentes dentro del mismo espacio de memoria (dentro del mismo nodo). La solicitud de información desde otro nodo implica un trasiego por la red de multitud de pequeños paquetes de datos para cada operación. Para conseguir paliar esta restricción un proceso de optimización del sistema consiste en la replicación de los datos más comúnmente utilizados sobre todo si no son modificados. Esto hace que la ejecución de operaciones sea casi siempre con datos locales.

Capítulo 7

IMPLANTACIÓN DE LA ARQUITECTURA Y DEL DISEÑO

Índice General

7.1 Introducción

Este último capítulo presenta una serie de directrices de implantación de la arquitectura genérica MOIRAE y del diseño de sistema de *Data Mining* propuesto. Estas directrices no son vinculantes y sólo pretenden indicar una de las múltiples formas de implementar las funcionalidades genéricas expuestas en los capítulos anteriores.

Para detallar más el nivel de especificación definido en el diseño, tanto de la arquitectura general como del sistema, es necesario seleccionar una serie de tecnologías en concreto. Estas tecnologías, lenguajes o herramientas específicas aproximan la definición genérica tanto del sistema como de la arquitectura a lo que será su implementación final. Dentro de estos criterios de implementación se destacan los siguientes elementos:

① ***Middleware* de comunicaciones CORBA**: La tecnología CORBA es mucho más abierta que soluciones dependientes del fabricante (DCOM) o del lenguaje de implementación (EJB/*Jini*). Otro factor determinante en la selección de esta tecnología es la disponibilidad de numerosos servicios que permite soportar ciertas funcionalidades de la arquitectura/sistema.

① **Protocolo de control basado en KQML**: KQML es un lenguaje de comunicación entre agentes suficientemente genérico y flexible para su adaptación a cualquier entorno. Adicionalmente, KQML se encuentra globalmente aceptado por la comunidad de investigadores del campo y se encuentra en fase de estandarización. Además, existen diversas implementaciones y utilidades para gestionar protocolos basados en KQML.

① **Prolog como lenguaje de definición de políticas**: Prolog tiene la ventaja de ser un lenguaje maduro, potente y flexible en el cual los programas son cortos y compactos, adecuados para su transmisión por la red. La sintaxis de Prolog está estandarizada por ISO y sus programas son ejecutables bajo diferentes plataformas. El algoritmo de unificación usado por Prolog para la resolución de consultas se adecua convenientemente al mecanismo de funcionamiento del motor de políticas.

El resto del capítulo presenta cada uno de los puntos, tanto de la arquitectura como del sistema, para los cuales se han adoptado recomendaciones de implementación mas relevantes, en relación a la tecnología usada. Estos puntos son:

❒ Funcionalidades y Diseño de un Componente (*toolkit* de MOIRAE).

❒ Motor de Políticas.

❒ Interacciones de Control.

- ❒ Relaciones Operacionales entre Componentes.
- ❒ Gestión de los Datos.
- ❒ Organización del Sistema (Federaciones).
- ❒ Tareas Concretas de Ciertos Componentes.

7.2 Funcionalidades y Diseño de Componentes

Como herramienta para el desarrollo de componentes, la arquitectura MOIRAE proporciona un lenguaje de descripción e implementación de componentes mediante el cual se definen cada uno de estos elementos de la misma. Dichos ficheros de descripción e implementación son procesados por medio de herramienta que generará un código en lenguaje de alto nivel (por ejemplo, C o C++) compilable para la generación de un ejecutable.

7.2.1 Ficheros MOIRAE

El diseño de un componente para MOIRAE implica la definición de al menos dos ficheros, uno de implementación (`.moi`) y uno o más de descripción (`.def`):

- ❒ **Ficheros de descripción**: Estos ficheros contienen la información que el resto de componentes o elementos externos deben conocer de un componente para poder solicitar servicios o hacer uso de las funcionales que proporciona dicho componente. Básicamente es una perspectiva externa del componente que no tiene que indicar cómo están programados los servicios o funcionalidades.
  Pueden existir varios ficheros de descripción usados por un mismo componente, pues el componente puede requerir la descripción de tipos de datos o estructuras de mensajes compartidas entre varios componentes o bien puede darse el caso de que un componente se derive en base a una jerarquía de otro componente.
- ❒ **Ficheros de implementación**: Estos ficheros indican cómo los servicios y operaciones definidas en el fichero de descripción del componente son implementadas. Este fichero contiene embebido código del lenguaje de alto nivel que se va a generar como parte de la implementación de las operaciones del componente. El objetivo de estos ficheros es la generación del código alto nivel, para su posterior compilación como un componente.

Estos ficheros son usados por varias herramientas de MOIRAE como entrada para la generación de código, validación o extracción de documentación. La sintaxis de estos ficheros se encuentra recogida el el Apéndice B.

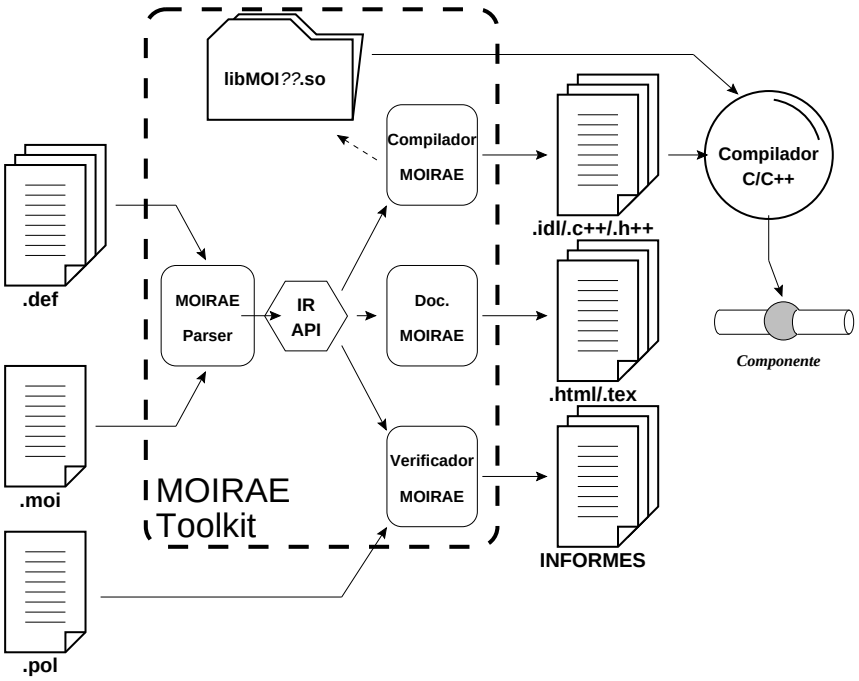


Figura 7.1: Herramientas del *toolkit* de MOIRAE

7.2.2 Herramientas MOIRAE

El conjunto de herramientas de soporte del *toolkit* MOIRAE se muestran en la figura 7.1. Estas herramientas son:

- ❒ Analizador (*MOIRAE Parser*).
- ❒ Compilador de Componentes.
- ❒ Verificador de Políticas.
- ❒ Generador de Documentación.

7.2.2.1 Analizador (*MOIRAE Parser*)

Este elemento no representa una herramienta de MOIRAE directamente, sino una librería de análisis léxico y sintáctico de los ficheros de descripción e implementación MOIRAE. Esta librería es capaz de leer cualquiera de estos ficheros y rellenar una estructura interna denominada **IR** (*Internal Representation*). Esta estructura interna está compuesta por una serie de tipos abstractos de datos (listas, colas, ...) que contendrán toda la información incluida en los ficheros. El resto de herramientas de MOIRAE incluirán esta librería y procesarán la IR que retorna para generar el tipo de salida asociada a dicha herramienta.

7.2.2.2 Compilador de Componentes

Esta primera herramienta tiene como objetivo la generación de una serie de ficheros en un lenguaje de alto nivel en base a la IR. Estos ficheros serán compilados directamente para generar un ejecutable que será un componente del entorno.

Para enlazar el ejecutable definitivo del componente es necesario incluir una serie de librerías adicionales tanto del *middleware* de comunicaciones como propias de MOIRAE, para ello el compilador de componentes indicará al compilador cual es la librería a enlazar. El conjunto de herramientas de MOIRAE incluirá múltiples librerías de enlace del tipo `libMOI??.so`.

En el caso de usar CORBA como *middleware* de comunicaciones, este compilador ha de generar no sólo el código del componente, sino ficheros, como las descripciones IDL de los interfaces del componente para que pueda ser invocado por otros componentes del sistema.

7.2.2.3 Verificador de Políticas

Esta segunda herramienta tiene como entrada la IR y un fichero de políticas y verifica que los componentes definidos por dicha IR puedan ser controlados correctamente por el conjunto de políticas indicadas en el fichero. Para realizar esta validación la herramienta comprueba una serie propiedades que tanto el componente como el conjunto de políticas deben cumplir. Este proceso de validación no detecta todos los posibles errores pero representa un filtro previo que asegura que ciertas circunstancias no se den a la hora de poner en ejecución los componentes y controlarlos por medio de estos conjuntos de políticas.

Las comprobaciones realizadas por este proceso se enfocan hacia:

① Verificación local de los componentes: Implementación de todas las funciones declaradas.

② Verificación global de los componentes: Referencias a funciones de otros componentes.

③ Cobertura del tratamiento de todos los eventos por medio de reglas de las políticas.

④ Detección de conflictos y bucles infinitos de control.

7.2.2.4 Generador de Documentación

La última herramienta del entorno MOIRAE es el generador de documentación. Esta utilidad genera una documentación en base a ficheros HTML o LaTeX, que describe los servicios, comandos y demás elementos del componente así como su implementación hasta el nivel de detalle descrito en los ficheros `.moi` y `.def`.

7.3 Motor de Políticas

El motor de políticas es el módulo central del agente que conforma el plano de control del componente. La función de este módulo es procesar las situaciones de conflicto para buscar la serie de acciones correctivas adecuada. La toma de decisiones realizada por este módulo se basa en información interna del motor y en la información que el plano de control tiene del estado del plano operacional así como de otros componentes. El procesamiento simbólico de este módulo debe contener todos estos elementos y además proporcionar algún tipo de mecanismo de inferencia que guíe el proceso de búsqueda de soluciones.

Las técnicas para implementar los motores de políticas deben tener en consideración dos factores fundamentales en los requisitos de funcionamiento de estos componentes:

- La flexibilidad y potencia en la búsqueda de soluciones, basada en un diseño de un motor que admita políticas complejas de control. Un motor de este estilo puede gestionar información incompleta del entorno o disponer de memoria en relación a pasados conflictos y soluciones. La evaluación de todos estos factores pueden permitir la elaboración de planes de acción óptimos para problemas complejos.
- La eficiencia en el tiempo de respuesta del motor. La evaluación de las opciones de respuesta a un conflicto debe de ser realizada en el menor tiempo posible. Este factor evalua el comportamiento a nivel de eficiencia de un componente MOIRAE en relación al elemento análogo que no separe planos operacional y de control.

Estos dos factores son fundamentales, cada uno en su faceta, para conseguir una solución aplicable de la arquitectura MOIRAE. La flexibilidad en las decisiones de control es la mayor ventaja en comparación con el diseño tradicional con funcionalidades de control y operacionales entremezcladas. Sin embargo, el plano de control no debe de ser un lastre a nivel de eficiencia, haciendo que operaciones relativamente simples requieran evaluaciones de acciones de control pesadas y lentas.

7.3.1 Tipos de Motores

Para satisfacer los requisitos antes citados existen dos estrategias. Por un lado, la búsqueda de un diseño de motor de políticas que equilibre dichos factores, debido a que cuanto más complejas son las políticas de control más lento es el proceso de evaluarlas. Por otro lado, en lugar de buscar un tipo de motor que encuentre un equilibrio de compromiso entre ambos factores, se pueden definir diferentes tipos de motores de control que acentúen más uno u otro factor. La opción tomada es la segunda, en base a la cual se definen los siguientes tipos de motores de políticas:

- ❒ Motores Reactivos.
- ❒ Motores Reactivos Condicionados.
- ❒ Motores Deliberativos.

7.3.1.1 Motores Reactivos

Orientados a la evaluación de políticas simples aplicables a casos sencillos. Definen una tabla estática de entradas del tipo estímulo–respuesta. Estos motores pueden ser implementados de forma muy eficiente aunque su aplicabilidad a problemas complejos es muy limitada.

7.3.1.2 Motores Reactivos Condicionados

Como extensión del tipo anterior, este modelo de motor de políticas añade condiciones de activación de las políticas. Las entradas de la tabla son elementos del tipo estímulo–condición–respuesta. La evaluación de las condiciones añade una mayor complejidad al motor y un retardo en su tiempo de respuesta, pero amplía el rango de problemas a los que se puede aplicar.

7.3.1.3 Motores Deliberativos

El tratamiento de conflictos complejos o la evaluación de planes de acción que afectan a múltiples componentes requieren la aplicación de otro tipo de políticas. El formato de estas políticas es mucho más complejo que en los casos anteriores y por lo tanto su articulación para la resolución de conflictos más difícil. La alternativa para el manejo de estas políticas son motores de control capaces de ejecutar algoritmos de búsqueda de soluciones más complejos que la simple equiparación estimulo–respuesta de los casos anteriores.

La alternativa optada para el diseño de este tipo de motor se basa en la representación de las políticas por medio de un lenguaje traducible a la representación de una WAM (*Warrem Abstract Machine*) [War83, AK91], como puede ser Prolog [ISO95]. El lenguaje Prolog tiene las siguientes ventajas:

- ❒ Los programas Prolog son mucho más compactos y pequeños que la misma solución en programación imperativa. Esta característica es fundamental para su transmisión por la red.
- ❒ Los programas Prolog son independientes de la arquitectura de la máquina que los ejecuta. A pesar de ello su ejecución es muy eficiente.
- ❒ La programación lógica permite la verificación formal de ciertas propiedades (completitud y corrección) de los programas.

- ❒ La programación en base a restricciones sobre dominios finitos está disponible para la gran mayoría de compiladores modernos de Prolog.
- ❒ Compiladores de Prolog como SICSTUS [SIC00], CIAO [HBC+99] o GNUProlog [Dia00] definen interfaces para interactuar con otros lenguajes de programación.

La principal ventaja de la definición de las políticas en Prolog es que las más simples de ellas pueden ser traducidas al formato estimulo–respuesta o estimulo–condición–respuesta, antes citado, de una forma sencilla. Esta funcionalidad permite que el mismo formato de las políticas sea cargable en todos los tipos de motores (siempre y cuando la complejidad de la política sea manejable por el motor).

7.3.2 Guías del Diseño de un Motor de Políticas en Prolog

La implementación del motor de políticas en Prolog implica la definición de una serie de predicados básicos de articulación de consultas y de recopilación de acciones de resolución, todos ellos como elementos básicos del motor. Estos predicados se complementan con las políticas, descritas también como predicados y cláusulas definidos en Prolog. Estas políticas son incluidas dentro del motor por medio del predicado `assert` que actualiza el contenido de la base de datos de políticas. La formulación de consultas se resuelve por medio del algoritmo general de unificación.

7.3.2.1 Formulación de Consultas

Se define un predicado `solve_conflict/2` cuya plantilla es:

FORMULACIÓN DE CONSULTAS:

```
solve_conflict(+conflict,-actions)
```

Dicho predicado evalúa las políticas para buscar el tratamiento adecuado al conflicto `conflict`. Las acciones correctivas definidas por las políticas se recogen en el término de salida `actions`.

La detección de eventos y las peticiones externas de otros agentes de control son las acciones que activan la formulación de consultas en base a este predicado. Por ejemplo, si el componente es un elemento de control de un grupo de componentes que ejecutan un proceso pesado y uno de los nodos se encuentra sobrecargado, se puede producir la siguiente consulta al plano de control del componente.

```
solve_conflict(overload(nodeA,2.3),As).

As=[move_component(component1,nodeA,nodeB),
    move_component(component2,nodeA,nodeC),
```

```
stop_component(component3)]
```

La secuencia de acciones `As` es el resultado de la evaluación de las políticas en base a dicho conflicto.

7.3.2.2 Formato de las Políticas

Las políticas se definen como formulas lógicas o sentencias de programación sobre dominios finitos que serán evaluadas por el motor de políticas. Por ejemplo, se puede disponer de una serie de restricciones de carga sobre los nodos de proceso del tipo:

```
ASSIGNATION=[PROC1,PROC2,PROC3,PROC4],
fd_domain_bool(ASSIGNATION),
calculate_process_load(LOAD1,LOAD2,LOAD3,LOAD4),
LOAD1*PROC1 + LOAD2*PROC2 + LOAD3*PROC3 + LOAD4*PROC4 #< MAXLOAD.
```

7.3.3 Otras Alternativas de Implementación

Como alternativa a la implementación del motor de políticas en Prolog, una alternativa adecuada puede ser su desarrollo en Java. Dicho lenguaje proporciona independencia de plataforma y capacidad para la carga dinámica y de forma remota de código.

Sin embargo, Java carece de alguna de las ventajas antes citadas sobre el diseño del motor en Prolog:

- ❒ Es un lenguaje procedimental y su validación y corrección no es aplicable.
- ❒ La políticas no son intercambiables entre motores reactivos y deliberativos. No es posible traducir de forma sencilla una clase Java a unas políticas del formato estímulo–respuesta.
- ❒ Las funcionalidades del lenguaje exceden de las necesidades del problema. Paquetes de clases, como AWT o las clases de comunicación no son necesarias dentro del motor de control y aun así forman parte del núcleo del lenguaje.
- ❒ La ejecución de una máquina virtual Java (JVM) dentro de un componente MOIRAE implica un consumo de recursos (CPU y sobre todo de memoria) excesivos.

7.4 Interacciones de Control

Las interacciones entre los planos de control de los diferentes componentes de la arquitectura se realiza por medio del interfaz de control. Este módulo permite que el agente de control de un componente coordine sus acciones con otros agentes de control. Como lenguaje de comunicación

entre estos agentes se propone KQML. Una implementación del lenguaje KQML es, por ejemplo COBALT [BDR98]. COBALT permite comunicación por medio de KQML sobre un *middleware* de comunicaciones CORBA.

7.4.1 Mensajes de Control en KQML

Las interacciones entre agentes de control pueden ser tanto las especificadas en el Modelo de Control, tales como la delegación, propagación y revocación de control; como las definidas por las propias políticas de control.

7.4.1.1 Propagación y Delegación de Control

El proceso de propagación de control consiste en la solicitud de la resolución de un conflicto que localmente no es posible resolver a un agente de control jerárquicamente superior. El resultado de esta operación puede ser tanto la intervención directa de otro agente en el conflicto como la delegación de un nuevo conjunto de políticas al agente que generó el conflicto (delegación de control).

La solicitud transmitida desde el agente generador del conflicto al agente de control superior se encapsula dentro de una *performative* de KQML de tipo `ask-one`.

PROPAGACIÓN DE CONTROL:

```
(ask-one
         :sender       origen
         :receiver     controlador
         :in-reply-to  id0
         :reply-with   id1
         :language     Prolog
         :ontology     MOIRAE
         :content      "solve_conflict(overload(nodeA,2.3),As)")
```

El agente de control puede responder a esta petición de tres formas diferentes:

① Por medio de un mensaje `insert` que contenga el nuevo conjunto de políticas de control que tratarán este tipo de conflictos. Esta respuesta se corresponde con una delegación de control.

DELEGACIÓN DE CONTROL:

```
(insert
        :sender       controlador
        :receiver     origen
        :in-reply-to  id1
        :reply-with   id2
        :language     Prolog
        :ontology     MOIRAE
        :content      "ASSIGNATION=[PROC1,PROC2,PROC3,PROC4],
                       fd_domain_bool(ASSIGNATION),
                       calculate_process_load(LOAD1,LOAD2,
                       LOAD3,LOAD4),
                       LOAD1*PROC1 + LOAD2*PROC2 +
                       LOAD3*PROC3 + LOAD4*PROC4 #< MAXLOAD.")
```

② Por medio de uno o varios mensajes `achieve` que implican la actuación directa del agente de control superior en las acciones del agente que originó el conflicto. Por medio de este mecanismo el agente de control superior activa los actuadores del componente inferior.

INTERVENCIÓN DIRECTA:

```
(achieve
        :sender       controlador
        :receiver     origen
        :in-reply-to  id1
        :reply-with   id2
        :language     Prolog
        :ontology     MOIRAE
        :content      "[move_component(component1,nodeA,nodeB),
                       move_component(component2,nodeA,nodeC),
                       stop_component(component3)]")
```

③ Un mensaje `sorry` si el agente de control no es capaz de resolver dicho conflicto. Por lo general este escenario se corresponde con algún problema de control más complejo.

ERROR:

```
(sorry
        :sender       controlador
        :receiver     origen
        :in-reply-to  id1
        :reply-with   id2)
```

7.5 Relaciones Operacionales entre Componentes

Como se vio a lo largo de la exposición del Modelo de Relación, los componentes del plano operacional pueden establecer diversos tipos de relaciones entre sí. Estas relaciones estaban clasificadas en diferentes tipos, dependiendo de sus características de visibilidad e identificación. En base a estas dos características se tenían relaciones públicas o privadas e identificadas o anónimas, respectivamente.

La implementación de esta tipología de relaciones está afectada, primordialmente, por el mecanismo de comunicación o *middleware* utilizado. En el caso de usar CORBA como sustrato de comunicación entre componentes, la implementación de estas relaciones sería el siguiente.

7.5.1 Relaciones Privadas–Identificadas

Se implementarán por medio de la definición de una variable local del componente que indique la referencia del objeto remoto apuntado por la relación.

7.5.2 Relaciones Públicas–Identificadas

Se podrán realizar por medio del Servicio de Relación de CORBA. Este servicio permite definir dependencias entre objetos de forma explícita. Dichas referencias son accesibles de forma externa para todo objeto que conozca el identificador de la relación. Dicho identificador es un nombre simbólico dado de alta en el Repositorio de Interfaces.

7.5.3 Relaciones Anónimas

Estas relaciones se establecen por medio del Servicio de Eventos o el Servicio de Notificación, dependiendo de las funcionalidades requeridas. Por medio de dicho servicio se puede definir un canal de eventos entre los componentes emisores y los receptores. Dependiendo del tipo de visibilidad de la relación el identificador del canal será público o sólo conocido por los componentes que intervienen en la relación.

7.6 Gestión de los Datos

La Federación de Procesamiento de Datos es la encargada de realizar las operaciones programadas por el resto de Federaciones sobre los datos gestionados por el sistema. Para ello, esta Federación dispone de dos tipos de componentes, componentes de proceso que representan operaciones sobre tablas de datos y componentes de datos. Estos últimos actúan como interfaz entre el resto

de componentes del sistema y los datos tal y como han de estar almacenados en el soporte físico.

7.6.1 Almacenamiento Físico

Los componentes de datos son los únicos componentes del sistema que acceden a la copia física de los datos del sistema. Esta copia física puede encontrase almacenada como un fichero dentro del sistema de ficheros local de un nodo o como una tabla relacional de una base de datos accesible por dicho nodo (por lo general también, local). La primera de las opciones es la que alcanza una mayor integración tal y como se encontraba enunciada en los objetivos del sistema. La otra es una solución más adecuada para operaciones de exportación de datos desde o hacia el sistema.

Desde el punto de vista del resto de componentes del sistema es completamente independiente el mecanismo de almacenamiento de los datos usado. De esta forma, es posible desarrollar las diferentes clases de componentes que deriven de la misma clase genérica de componente de datos pero que den acceso a diferentes mecanismos de almacenamiento físico.

7.6.2 Consulta de los Datos

Una vez que estos componentes cargan los datos, estos son accesibles al resto de componentes del sistema por medio de los diferentes servicios que proporcionan estos componentes. La principal característica de este tipo de interacción entre componentes es que el trasiego de información y el volumen total de los datos transmitidos es muy grande. El rendimiento final del sistema está determinado en gran medida por la eficiencia con la que se haga este proceso.

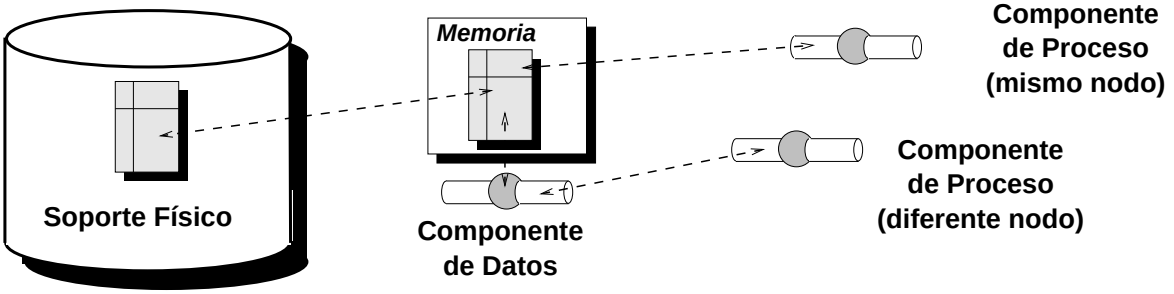


Figura 7.2: Interacción con los componentes de datos

A la hora de analizar los tipos de componentes que pueden acceder a los datos existe una división fundamental entre componentes ubicados en el nodo local y componentes remotos (Figura 7.2). Para el caso de los primeros es posible optimizar el proceso de consulta utilizando mecanis-

mos de memoria compartida entre el componente de datos y el componente de proceso que los utiliza. Esta posibilidad no es aplicable para componentes remotos, por lo cual en dichos casos las peticiones deben de ser canalizadas por el componente y retransmitidas por medio de mensajes. En este caso, la transmisión comprimida de datos puede dar una solución parcial al problema de la pérdida de rendimiento en relación a la interacción local.

7.7 Organización del Sistema

El diseño del sistema DI-DAMISYS presentado en el Capítulo 6 se desglosa a nivel de grupos de componentes denominados Federaciones. Estas Federaciones representan el nivel de organización de las diferentes tareas proporcionadas por el sistema. La descomposición a nivel de componentes de las tareas de cada Federación se realiza de forma sucesiva en diferentes grupos de componentes. La gestión de esta organización es fundamental para la administración y mantenimiento del sistema. Las pautas de diseño e implementación directamente ligadas a esta gestión son las siguientes:

- Esquema de organización de los grupos de componentes y Federaciones.
- Desarrollo de la base de datos de nodos de despliegue y el catálogo del sistema.
- Diferentes implementaciones de una misma clase de componente y diferentes instancias de una misma implementación.

7.7.1 Esquema de Organización

Cada uno de los componentes del sistema está identificado en base a dos nombres. Por un lado, se le asocia a un nombre de clase de componente que identifica a todos los componentes del mismo tipo, es decir con el mismo interfaz. Por otro lado, cada instancia, es decir cada componente, individualmente está identificado de forma única.

La gestión de estos esquemas de nombrado se puede realizar sobre el soporte del Servicio de Nombres de CORBA. Este servicio permite mantener una organización jerárquica de identificadores simbólicos. Cada una de las entradas de este servicio tiene asociado un campo *id* que contiene el nombre, así como un campo *kind* que identifica el tipo del componente. Este segundo campo puede ser usado para indicar la clase a la que pertenece el componente. Las entradas dentro de este servicio pueden corresponderse con objetos del entorno distribuido o con subdominios de nombres, que a su vez contendrán otros objetos o subdominios y así sucesivamente. Esta organización puede construirse de forma análoga al proceso de descomposición en grupos de componentes del sistema, como se muestra en la figura 7.3.

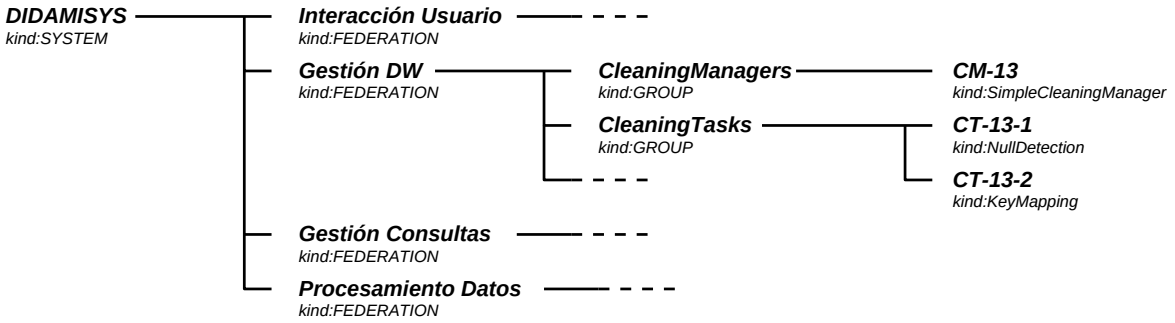


Figura 7.3: Esquema de nombrado de los componentes del sistema

7.7.2 Catálogo del Sistema

El Catálogo del Sistema contiene la información que el sistema necesita para realizar sus funciones. Intuitivamente, las funcionalidades de este elemento se pueden extrapolar a las de la estructura análoga dentro de un SGBD. Para un gestor de bases de datos, cuyas funcionalidades se centran en la manipulación de los datos, la información relativa a cuáles son los datos que contiene, los usuarios registrados y los permisos que dichos usuarios tiene para realizar operaciones es fundamental para su funcionamiento. En el caso de DI-DAMISYS dicha información es también imprescindible para poder realizar las funciones del sistema. Pero además, se requiere otro tipo de información que no es necesaria en el caso de los SGBD.

El Catálogo del sistema DI-DAMISYS es una extensión del definido para el sistema predecesor DAMISYS [Par98]. La información que necesita el sistema para operar, y que por lo tanto debe de estar almacenada en el Catálogo del Sistema, es:

- **Información sobre los datos**: Información referente a las tablas de datos gestionadas por el sistema y sobre las cuales se van a realizar las operaciones de *Data Mining*. Dichas tablas de datos, son tanto las tablas iniciales de datos como las derivadas a lo largo de diferentes procesos de consulta y preparación de datos. La información sobre estos datos es de dos tipos:
  - *Taxonomía de los datos*: Indica, cuáles son las tablas de datos, cuántos atributos tienen, cuántas tuplas, el dominio de cada uno de los atributos, así como cualquier información auxiliar que indique cualidades o características de los datos que sean relevantes para las operaciones de análisis.
  - *Origen de los datos*: Proporcionando información sobre las fuentes de datos utilizadas para recoger dicha información, así como las operaciones realizadas para su integración y carga dentro del sistema. Esta información útil para realzar procesos de actualización

de los datos según las fuentes de datos originales sean modificadas.

- **Información sobre los usuarios**: Cada uno de los usuarios del sistema que deseen utilizar el sistema deben de encontrarse registrados en el mismo. Esto es, su información relativa a nombre de usuario, mecanismo de autentificación (por lo general, clave de acceso) y permisos dentro del sistema deben de ser conocidos por el sistema. Para la gestión de permisos, estos pueden agruparse en roles o perfiles que fijen una serie de privilegios compartidos por varios usuarios.

- **Información sobre las operaciones**: Las operaciones que el sistema puede realizar sobre los datos se dividen en dos niveles:

  - *Operadores*: Operaciones elementales sobre las tablas de datos. Son operaciones simples que procesas un conjunto de tablas de entrada, generando un conjunto de tablas de salida, generadas o transformados por el operador. Un operador se invoca con una serie de parámetros de entrada que perfilan su funcionamiento. Un operador, a su vez, produce una serie de resultados simples (no tablas) que indican determinados tipos o condiciones de finalización de la operación.

  - *Algoritmos*: Operaciones más complejas, compuestas por varios operadores. Un algoritmo representa un proceso o subproceso completo. Al igual que un operador tiene tablas de entrada y salida, así como parámetros y resultados. La principal diferencia entre operador y algoritmo, el último puede generar diversos resultados intermedios a nivel de tablas usados como salidas de ciertos operadores internos y entradas de otros.

  Tanto los operadores como los algoritmos tienen numerosas características, tales como entradas y salidas, pre- y post-condiciones, así como otras restricciones y características necesarias para la planificación de las consultas, como pueden ser los privilegios necesarios para su utilización.

- **Información interna del sistema**: Por último, el catálogo del sistema debe dar soporte a otro tipo de información directamente utilizada por el propio sistema en sus procesos internos. Este tipo de información puede ser tanto registros de operaciones y *log* del sistema, información de auditoría o estadísticas internas del uso de recursos.
  Esta información puede ser usada para la optimización del sistema a largo plazo.

La información en el catálogo, salvo la relativa a los elementos más tradicionales, como usuarios y datos, es principalmente dinámica. Este dinamismo no sólo se centra en el contenido de las tablas del catálogo, sino que se extiende a la definición de nuevas tablas de catálogo. Por ejemplo, las funcionalidades proporcionadas por la Agenda de Tareas, para poder planificar la ejecución de trabajos *off-line* y otras operaciones periódicas, requiere ciertas estructuras para anotar las tareas

pendientes y las horas de realización de las mismas. Este tipo de información correspondería a una nueva tabla del catálogo asociada a la nueva funcionalidad.

El concepto de una estructura de catálogo estática carece de sentido en un sistema de características dinámicas. La definición e incorporación de nuevas funcionalidades debe permitir la extensión del catálogo de forma dinámica.

7.7.2.1 Almacenamiento y Consulta del Catálogo

El catálogo, como tablas de datos que son, se encontrará almacenado en la Federación de Procesamiento de Datos. Como todos los datos del sistema, para su utilización se activará un componente que actúe de interfaz entre el resto de componentes del sistema y el soporte físico que almacena dicha información. El proceso de consulta y manipulación del catálogo se canaliza por medio de estos componentes de interacción con los datos, siendo los encargados de garantizar la manipulación de los mismos sólo por usuarios privilegiados.

7.7.3 Diferentes Implementaciones de una Clase

El modelo de componente definido como parte de la arquitectura MOIRAE, define que cada uno de los elementos del sistema debe de estar definido por medio de una interfaz de acceso. Dicha interfaz de acceso, especifíca los servicios proporcionados al resto de componentes y se describe en una *Clase de Componente*. Estas Clases son implementadas de diferentes formas, para distintas arquitecturas hardware y/o sistemas operativos y en diferentes lenguajes de programación. Estas implementaciones concretas se denominan *Implementaciones de Componente*. Por último, cada una de las instancias o ejecuciones de estas implementaciones se denomina *Componente* que a su vez tendrá unos determinados argumentos o parámetros de funcionamiento que lo diferencian del resto de instancias de la misma implementación.

Para todo componente que está en ejecución, se conoce cual es la Clase a la que pertenece. Esta Clase está asociada a un interfaz IDL determinado y dicho interfaz es el registrado por el componente a la hora de darse de alta como objeto CORBA. Una vez que un componente está funcionando, el resto de información referente a la implementación e instancia del componente pueden resultar de interés o necesarios para determinadas operaciones, como por ejemplo son la planificación de consultas o la movilidad de componentes. Como solución a la publicación de información adicional por parte del componente una vez se encuentra en ejecución se ha recurrido al Servicio de Propiedades de CORBA. Dicho servicio permite la definición de pares identificador valor de forma dinámica por parte de todo objeto CORBA. Por medio de esta información se proporciona

información adicional sobre un componente mas allá de la Clase de Componente que lo define.

7.8 Tareas de otros Componentes

Junto a los casos antes citados de funcionalidades principales del sistema y su posible implementación, existen otras tareas menores cuyo detalle de implementación puede ser de interés.

- ❒ Conexión de usuarios al sistema.
- ❒ Registro de funcionalidades del sistema.

7.8.1 Conexión de Usuarios

Los usuarios acceden al sistema DI-DAMISYS por medio del componente `Connection Manager`. Las tareas de este componente son la verificación de las identidad del usuario y el control de las conexiones establecidas por el sistema. Debido al modelo usado para realizar dichas tareas pueden resultar de interés los siguientes Servicios de CORBA:

- ❒ Servicio de Seguridad: Para la autenticación de usuarios, transmisión de claves o gestión de certificados y claves de sesión.
- ❒ Servicio de Licencia: Para la gestión del numero de conexiones máximas del sistema, periodos de uso y cualquier otra restricción en la utilización del sistema.

7.8.2 Registro de Funcionalidades

Debido a la naturaleza dinámica del sistema, las funcionalidades del sistema pueden variar a lo largo de la ejecución del mismo. En el Capítulo 6 de explicación del sistema se indicó que los servicios del sistema, tales como tipos de consultas u operaciones de *Data Mining* así como las tareas de preprocesamiento y carga de datos se podían añadir dinámicamente. Para hacer posible que las aplicaciones de interfaz de usuario reconociesen las funcionalidades disponibles en el sistema existe un diálogo entre estos componentes y el representante de cada una de estos servicios del sistema. La opción de disponer de un componente que centralice las funcionalidades registradas y se encargue de proporcionárselas a las aplicaciones cliente cuando estas se conectan puede ser realizada también por medio del Servicio de Negociación de una forma más flexible y estandarizada.

Capítulo 8

ESCENARIOS EXPERIMENTALES

Índice General

8.1 Objetivo de los Experimentos

Para evaluar las capacidades de la arquitectura propuesta a la hora de resolver problemas de control, a lo largo de esta sección se exponen tres escenarios experimentales de trabajo. El objetivo de estos experimentos no es el de conseguir nuevos algoritmos de distribución de trabajos, equilibrado de carga o *Data Mining* distribuido. En su lugar se pretende demostrar cómo la arquitectura MOIRAE puede ser aplicable a dichos problemas de control, por medio de los siguientes pasos:

① Definición del problema en base a componentes MOIRAE, describiendo las situaciones a controlar y los parámetros de decisión disponibles.

② Desarrollo de diferentes políticas de control para las situaciones (conflictos) a tratar.

③ Modificación de las políticas de control de los componentes incluso durante su ejecución.

La evaluación de los experimentos será diferente para cada uno de los casos, pues dichos experimentos se han definido para evaluar las posibles repercusiones derivadas del diseño de sistemas distribuidos por medio de la arquitectura MOIRAE. Los factores que se pretenden evaluar por medio de los experimentos son:

❒ Efectos sobre el rendimiento y la eficiencia: Comparar la evaluación de políticas dinámicas en relación a las decisiones de control estáticas tradicionales.

❒ Monitorización de un sistema bajo diferentes condiciones de carga y entornos de trabajo: Estudiando la evolución del sistema con diferentes políticas de control y la modificación de éstas durante la ejecución.

❒ Generalización de un problema del entorno de *Data Mining*: Definición de diferentes soluciones al mismo por medio de un conjunto común de componentes y diferentes políticas de control.

❒ Verificación de políticas: Procesos automáticos de verificación de un conjunto de políticas para comprobar que cubren todos los casos que pueden presentarse de un determinado conflicto.

En resumen, no se pretende hallar nuevas soluciones a los problemas sobre los cuales se ha experimentado, sino que se apunta hacia la redefinición de dichos problemas por medio de una arquitectura que facilite su descomposición. De esta forma, se pueden desarrollar las soluciones existentes y se deja la puerta abierta a nuevas soluciones a los problemas planteados de esta forma. La ventaja radica en el campo de investigación, en la facilidad para desarrollar esta nuevas soluciones y dentro del campo de aplicación, en la posibilidad de disponer de diferentes políticas de control aplicables a un mismo problema dependiendo de determinados factores de utilización del sistema o entorno.

8.1.1 Descripción de los Experimentos

La evaluación de la arquitectura en base a los criterios antes citados se ha realizado por medio de tres escenarios experimentales:

- Problema de los lectores escritores: Un escenario sencillo con un único componente a controlar. Permite evaluar la eficiencia de los diferentes modelos de control para una implementación concreta de la arquitectura.
- Distribuición de trabajos: Un entorno en varios niveles con elementos de control que interactúan entre sí. Este escenario permite experimentar con diferentes políticas de control y con diferentes baterías de trabajos. El objeto de este segundo experimento es doble, por un lado mostrar la flexibilidad de un entorno basado en componentes MOIRAE y en segundo lugar evaluar el efecto del cambio de políticas de control durante la ejecución del sistema. Adicionalmente se incluye un ejemplo de un proceso de validación de políticas para un caso concreto.
- Algoritmos de *Data Mining*: Dentro del campo de *Data Mining* se ha reformulado una familia de algoritmos distribuidos para extracción de reglas de asociación por medio de: (i) un conjunto de componentes común y (ii) un reducido conjunto de reglas de control características de cada algoritmo. Este último escenario muestra cómo un algoritmo de *Data Mining* puede ser dividido en funcionalidades operacionales y de control para su implementación en la arquitectura MOIRAE.

8.2 Escenario 1: Lectores–Escritores

Este experimento abarca un escenario sencillo de control en el cual se pueden evaluar aspectos de rendimiento más difíciles de extraer en escenarios más complejos. El problema a modelizar por medio de componentes MOIRAE es el de la comunicación de lectores y escritores por medio de un *buffer* de almacenamiento. Dicho elemento es un recurso compartido entre los elementos que lo usan y su gestión es el factor a controlar por medio de las políticas.

8.2.1 Componentes del Experimento

Como se muestra en la figura 8.1, este escenario está compuesto por tres tipos de elementos:

- **Escritores**: Procesos que generan datos de forma continua.
- **Lectores**: Consumidores de datos. Procesan los datos generados por los elementos anteriores.
- ***Buffer***: Almacenamiento intermedio entre lectores y escritores. Este elemento mantiene una cola de datos alimentada por los procesos escritores y consumida por los lectores.
  Este elemento se modeliza como un componente MOIRAE. Para ello se deben definir sus sensores, actuadores y eventos:

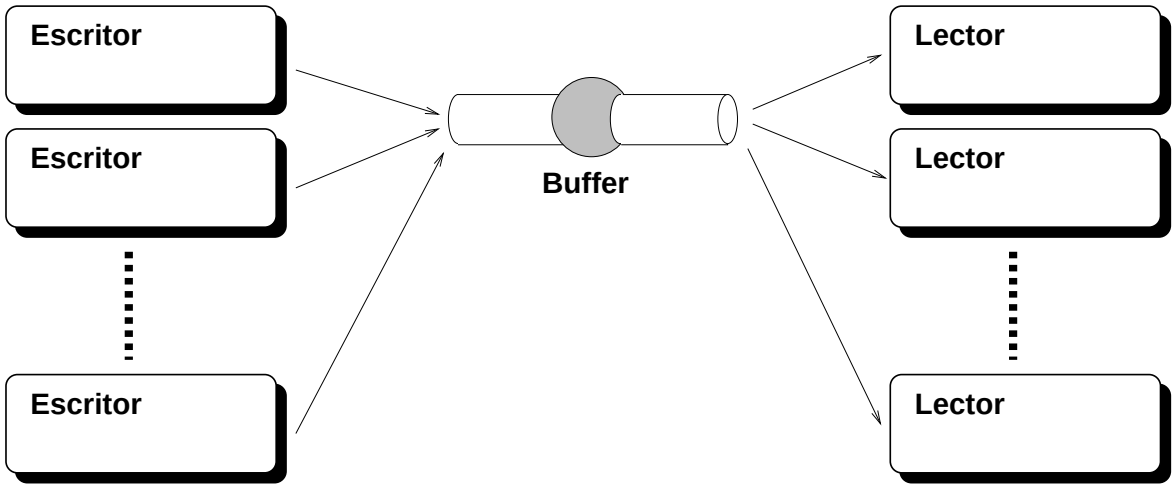


Figura 8.1: Componentes del Experimento 1

- **Sensor** `buffer_size`: Tamaño reservado para el *buffer*. Determina cual es la memoria actualmente usada para almacenar los datos.
- **Sensor** `used_size`: Posiciones actualmente ocupadas por los datos almacenados.
- **Actuador** `resize_buffer(n)`: Redimensiona el *buffer* a tamaño *n*.
- **Actuador** `reset_buffer()`: Vacía el *buffer* de todos los datos almacenados.
- **Actuador** `block_operation()`: Bloque la operación realizada hasta que la condición apropiada la despierte. Espera pasiva.
- **Actuador** `redo_operation()`: Reintenta la última operación realizada. Es de suponer que ciertas acciones de control se han realizado para resolver los problemas que de ella se derivasen.
- **Evento** `buffer_is_full`: Producido al intentar realizar una inserción sobre un *buffer* lleno.

El escenario definido para esta serie de experimentos permite parametrizar los siguientes factores:

- Número de lectores y de escritores.
- Distribución a lo largo del tiempo de las peticiones.
- Tiempos medios de proceso (de lectores y escritores) sin realizar peticiones al *buffer*.
- Tamaño máximo del *buffer*.
- Número de iteraciones del experimento.

8.2.2 Políticas de Control

El objetivo de este experimento es comparar la ejecución de una decisión de control simple (*buffer* de escritura lleno) para tres modelos de control diferente:

① Control programado: Incluyendo dentro del código del *buffer* el criterio de redimensionamiento del mismo. Solución tradicional.

② Políticas de control estáticas: Siguiendo la filosofía de control de la arquitectura MOIRAE el plano de control será activado al producirse un evento `buffer_is_full`. Las políticas de control estarán cargadas estáticamente en el código del plano de control.

③ Políticas de control dinámicas: Al igual que en el caso anterior el conflicto será procesado por el plano de control del componente. La principal diferencia es que las políticas son cargadas dinámicamente por el plano de control al comenzar la ejecución.

La implementación de los prototipos usados en el desarrollo de este experimento se ha realizado en C y en el caso de los planos de control, sus funcionalidades de evaluación se han implementado usando un motor *Prolog*, en concreto *GNU-Prolog*. La principal diferencia entre las soluciones ② y ③ radica en la eficiencia a la hora de la evaluación. El código estático *Prolog* se evalúa de forma más eficiente que el código dinámico, sin embargo las funcionalidades de carga y descarga dinámica de políticas exigen esta segunda funcionalidad. En el caso de *Prolog* la gestión dinámica de código puede realizarse por medio de predicados `assert/1` o `retract/1` o por medio de mecanismos auxiliares que proporcione el entorno utilizado. Por lo general estos otros mecanismos son mucho más eficientes aunque no son estándares.

> POLÍTICA DE REDIMENSIONAMIENTO: Las políticas de control a aplicar en este experimento son muy simples. Si el espacio reservado para el *buffer* está lleno, entonces se redimensionará su tamaño en ampliaciones de un tamaño fijo, hasta un máximo permitido. Si dicho valor límite es alcanzado los procesos de escritura se bloquean hasta que haya espacio libre.

8.2.2.1 Resultados Experimentales

El parámetro central a evaluar por medio de este escenario es el rendimiento de los procedimientos de control. Para este experimento los procesos productores y consumidores implementan un bucle infinito de operaciones que mediante un parámetro de distribución dividen entre operaciones internas y lecturas/escrituras del *buffer*. Este parámetro determina el tiempo de consumo o generación de un datos frente al tiempo de acceso al recurso compartido y la consiguiente acción de control derivada.

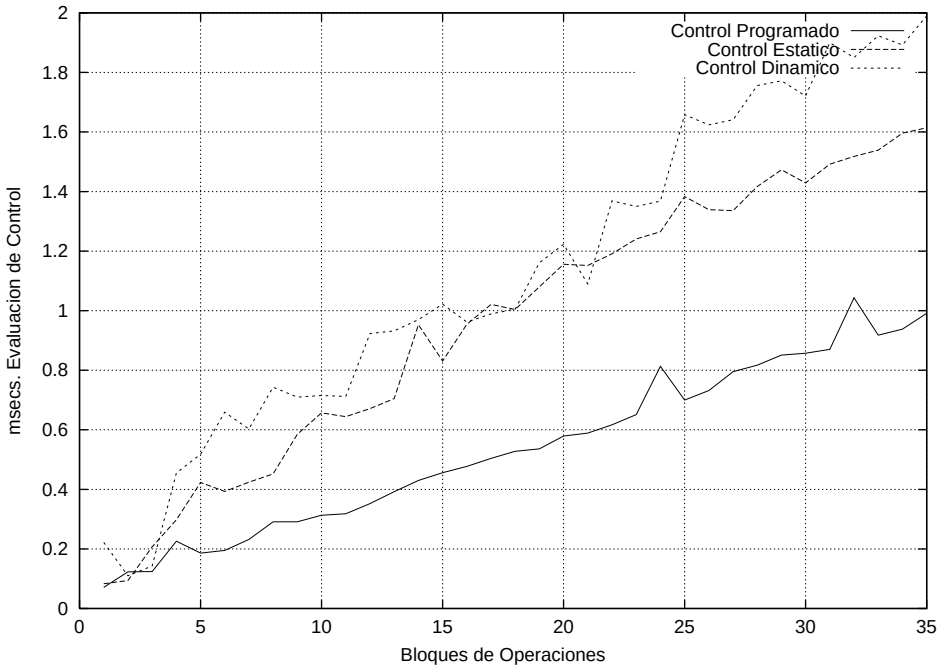


Figura 8.2: Evaluación de políticas vs iteraciones del experimento

La gráfica 8.2 muestra la progresión del tiempo acumulado de evaluación de las políticas de control para series de experimentos de diferente tamaño. La distribución de operaciones internas y accesos al *buffer* se mantiene constante así como el número de lectores y escritores. En esta gráfica se aprecia la diferencia entre la evaluación de control programada y el modelo de políticas tanto estáticas como dinámicas. Asimismo, se puede observar que bajo este parámetro todos los modelos de control escalan de forma lineal.

La figura 8.3 por su parte muestra el tiempo medio de evaluación de control para el caso de un incremento en el número de procesos lectores y escritores. El valor mostrado es el tiempo promediado que ha invertido cada proceso que accede al *buffer* en operaciones de control. Como se puede observar dicho tiempo promedio se mantiene incluso al incrementar los procesos que usan dicho recurso.

Por último en la gráfica 8.4 se pueden comparar la evolución de los diferentes modelos de control al aumentar la frecuencia de operaciones de acceso al *buffer* por ciclo de operación de los procesos. La diferente pendiente de las líneas está relacionada con los tiempos de evaluación promedio vistos en la gráfica anterior.

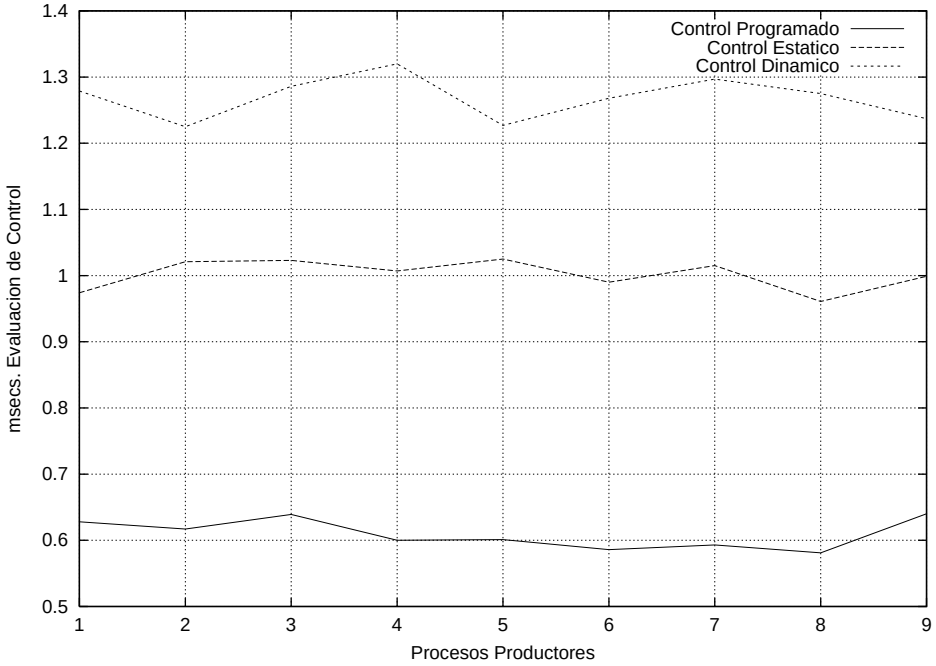


Figura 8.3: Evaluación de políticas vs número de procesos productores

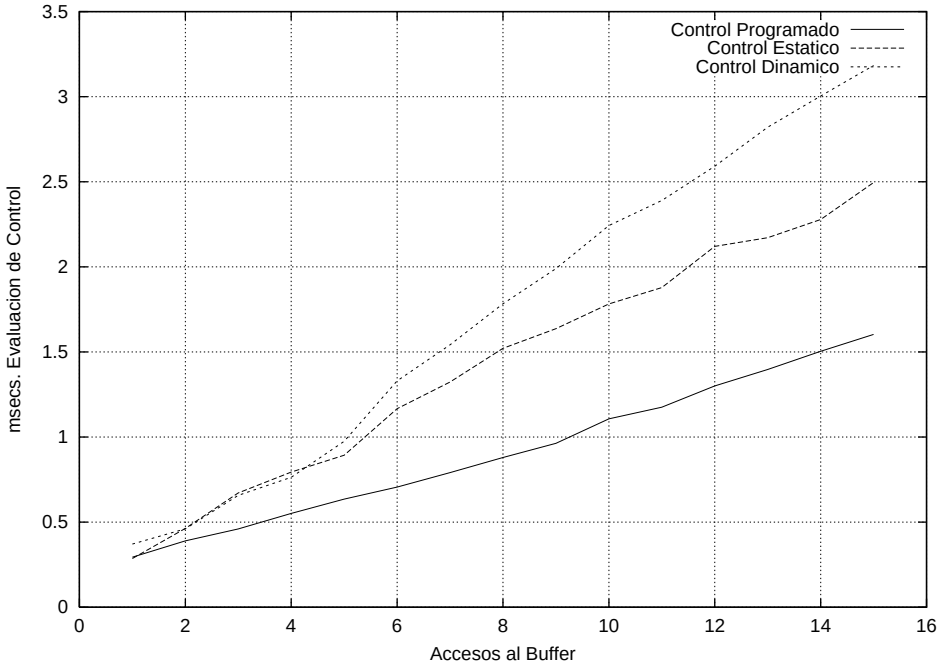


Figura 8.4: Evaluación de políticas vs frecuencia de operaciones de acceso a *buffer*

8.2.3 Evaluación de Resultados

La experimentación realizada sobre el escenario propuesto demuestra la merma en eficiencia anunciada al introducir un mecanismo de evaluación de situaciones de control dinámico. El ejem-

plo propuesto es el caso extremo en el que se aprecia esta diferencia en el rendimiento. La decisión de control a interpretar es fácilmente programable dentro del componente (*buffer*) y su tiempo de ejecución es mínimo. Todas las soluciones comparten un mismo retaro común, derivado del bloqueo de la operación de escritura y de la posterior realización de la decisión tomada.

El tiempo de evaluación de las decisiones de control se ve incrementado del orden de un 64% para el caso de políticas estáticas y de un 108% en el caso de políticas dinámicas. Esta diferencia como ya se comentó es debida a los mecanismos que proporciona el entorno de programación de las políticas de control (Prolog en este caso) para incluir código. Este incremento es asumible en los casos en los cuales las ventajas ganadas por medio de este entorno compensen dicha pérdida de eficiencia. Un factor a tener en cuenta es el porcentaje de tiempo invertido en operaciones de control en relación a procesamiento operacional. En el escenario propuesto esto sería asimilable a la relación entre el tiempo de acceso al *buffer* y el de producción o consumo de los datos transferidos.

Más adelante se muestran políticas de control más complejas en las cuales la diferencia entre la versión tradicional y la programada por medio de políticas de control (estáticas o dinámicas) es mucho menor. Esta convergencia es debida a que la sobrecarga computacional asociada al propio entorno de evaluación de las políticas se atenúa en el momento en el cual las propias operaciones de decisión se hacen más complejas. En sí, el proceso costoso es el de invocación del entorno de evaluación de las políticas.

8.3 Escenario 2: Distribución de Trabajos

Este segundo escenario experimental plantea un supuesto distribuido más complejo que el visto anteriormente. En este caso se pretende modelizar un sistema de distribución de trabajos que reparta la carga asociada a la realización de una serie de procesos de cálculo entre diferentes nodos de computación.

El objeto de este segundo escenario es el evaluar la flexibilidad de un diseño en dos niveles de componentes MOIRAE desde la perspectiva de modificación estática y dinámica de las políticas. Se pretende definir un esquema de componentes para la resolución de un problema y experimentar el uso de diferentes políticas de control para dicho entorno. Estas políticas se irán definiendo a medida que se introduzcan nuevos criterios de control. Se han definido tres diferentes sub-escenarios dependiendo de las consideraciones de control:

- ❒ Distribución de carga sin prioridades.

- ❒ Distribución de carga con prioridades.
- ❒ Distribución de carga basada en restricciones de recursos.

Los dos primeros casos ilustran las capacidades de flexibilizar en control alcanzables por medio de la arquitectura MOIRAE. El último de los casos está enfocado hacia la verificación automática de políticas.

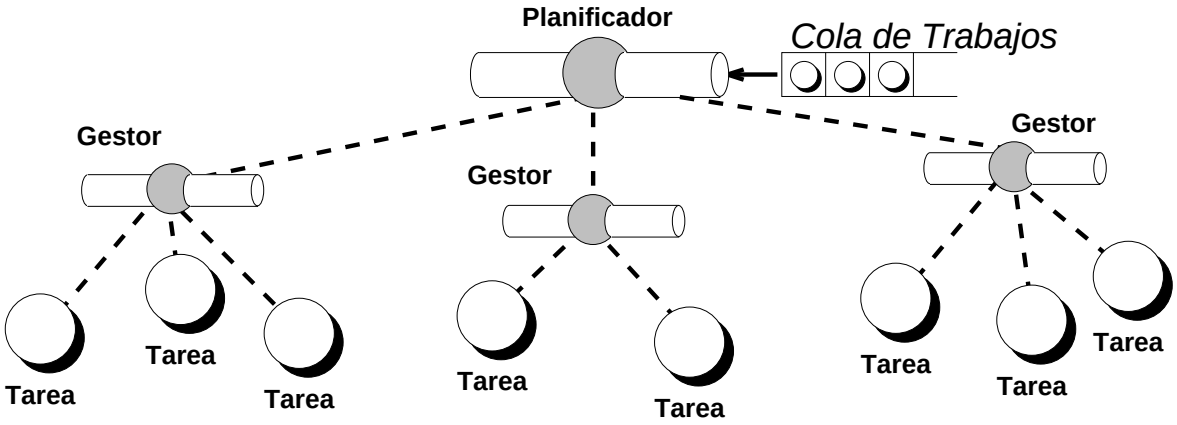


Figura 8.5: Componentes del Experimento 2

8.3.1 Componentes del Experimento

La figura 8.5 muestra los componentes que intervienen en el diseño del experimento. El escenario propuesto dispone de dos niveles de control: (i) un nivel superior con un planificador de la distribución de trabajos y (ii) un segundo nivel formado por gestores locales de tareas. Los componentes que conforman el entorno son:

- ❒ **Planificador**: Componente que gestiona un cola de trabajos pendientes y que decide la distribución de los mismos entre los diferentes nodos de trabajo. Las decisiones aplicables a este nivel deben permitir equilibrar la carga de los nodos y satisfacer restricciones de ejecución y privilegios de los trabajos a realizar.
  Este componente no tiene sensores directos sino que mantiene una información aproximada del estado de los nodos registrados (estado del nodo, disponibilidad de la conexión, etc.). Esta información es actualizada por medio de mensajes recibidos periódicamente o bajo demanda.
  Los actuadores de este componente le permiten asignar y retirar tareas a los gestores, así como emitir una serie de mensajes de control hacia estos.
- ❒ **Gestores**: Componentes asociados a la monitorización y control de cada nodo de proceso.

Estos componentes conocen el estado del nodo en relación a la carga actual así como otras restricciones de uso aplicables a los trabajos a ejecutar en él. Son encargados de gestionar los trabajos asignados a dicho nodo, decidiendo si se lanza o no su ejecución, así como si dicha ejecución se congela momentáneamente.

- **Tareas**: Procesos del sistema encargados de la realización de un trabajo dentro de un nodo. Estos procesos no son componentes MOIRAE y carecen de plano de control. Representan el trabajo computacional de la resolución del problema.

El problema de la asignación de trabajos, tal y como se encuentra planteado en este escenario, se resuelve en base a decisiones de control tomadas en cualquiera de los dos niveles. El componente **planificador** representa un nivel de control global de la carga del sistema. Este nivel no conoce información instantánea del estado de los nodos, en su lugar dispone de datos periódicamente actualizados de dicho estado. El nivel de control representado por los **gestores** de nodos dispone de mucha más información local, pero no tiene responsabilidades sobre la planificación general de trabajos.

8.3.2 Políticas de Control Sin Prioridades

El objeto de este primer sub-escenario es mostrar cómo se pueden definir varias políticas de control para el mismo problema (distribución de trabajos) de forma que, dependiendo de la taxonomía de la batería de trabajos a realizar y del rendimiento esperado del sistema, se mejoren los resultados. Un factor a considerar en la evaluación de las diferentes políticas es determinar cuál es el parámetro a optimizar, por ejemplo el tiempo total consumido en la ejecución de trabajos, el sumatorio del tiempo de ejecución de cada trabajo o incluso factores ponderados en base a diferentes criterios de prioridad. Debido a que dichos criterios pueden ser diferentes, la posibilidad de modificar las políticas permite asignar el modelo de distribución de trabajos más apropiado para alcanzar la meta planteada.

La política más simple que se ha aplicado a este experimento consiste en un proceso de asignación de tareas de forma ponderada teniendo en cuenta la carga y potencia de ejecución de los nodos. Mediante un proceso de calibración previo y una función de monitorización se dispone de una métrica de la capacidad de procesamiento disponible en cada nodo. Dicho factor es usado para definir la política DBFC (Distribución *Batch* con Factor de Carga):

> DBFC: Los trabajos son procesados según su orden en la cola de trabajos (sin prioridades). La asignación se realiza al nodo más descargado de los disponibles.

Una mejora a esta última a política de control se ha denominado *DBFCC* (Distribución *Batch* con Factor de Carga y Corte):

> DBFCC: Los trabajos son procesados según su orden en la cola de trabajos (sin prioridades). La asignación se realiza al nodo más descargado de los disponibles. Si la carga de proceso de dicho nodo excede un valor de corte la asignación de tareas se bloquea hasta que la carga de algún nodo esté por debajo de dicho valor.

Por último se propone una variante de la política anterior que incluye un factor de corrección asignando por el planificador de trabajos. Esta política se ha denominado DBFCCA (Distribución *Batch* con Factor de Carga y Corte Ajustado):

> DBFCCA: Ajuste sobre la política DBFCC incluyendo un factor de corrección a la carga estimada del nodo inmediatamente después de la asignación de una tarea. Dicho factor modifica el factor de carga hasta que éste sea actualizado por el gestor del nodo con un nuevo valor real.

8.3.2.1 Resultados Experimentales

La evaluación de estas políticas se ha realizado por medio de un *cluster* de 6 nodos, tres de ellos de elevada potencia de cálculo y otros 3 de menores prestaciones. Los trabajos que se desean realizar son tareas de cálculo masivo sin entrada/salida.

La aplicación de la política DBFC con baterías de trabajos de gran tamaño causa la saturación de los nodos de proceso al ejecutar un elevado número de tareas de forma simultánea. Su ejecución sobre un *cluster* heterogéneo de nodos hace que los procesadores de menor potencia se colapsen a pesar de la ponderación de carga usada en la distribución de trabajos.

El uso de la política DBFCC anula el factor de saturación de los nodos, deteniendo el reparto de tareas en base al estado de carga. Sin embargo esta política usa como parámetro de calibración un factor de carga que, como ya se indicó, es aproximado. Dicho valor es refrescado por el gestor del nodo con cierta frecuencia, pero pueden darse casos en los cuales la planificación de tareas se realice sobre un factor de carga bajo y debido a que su actualización no se produce a tiempo y como concecuencia a dicho nodo sean asignadas varias tareas. Esta situación es mejorada por medio de la política DBFCCA la cual es menos proclive a situaciones de este estilo.

La gráfica 8.6 muestra la evolución del número de tareas que uno de los nodos de altas prestaciones ha realizado con cada una de las políticas. Como se puede observar en el caso de DBFC este parámetro se incrementa constantemente hasta que no se disponen de más tareas a realizar. En el

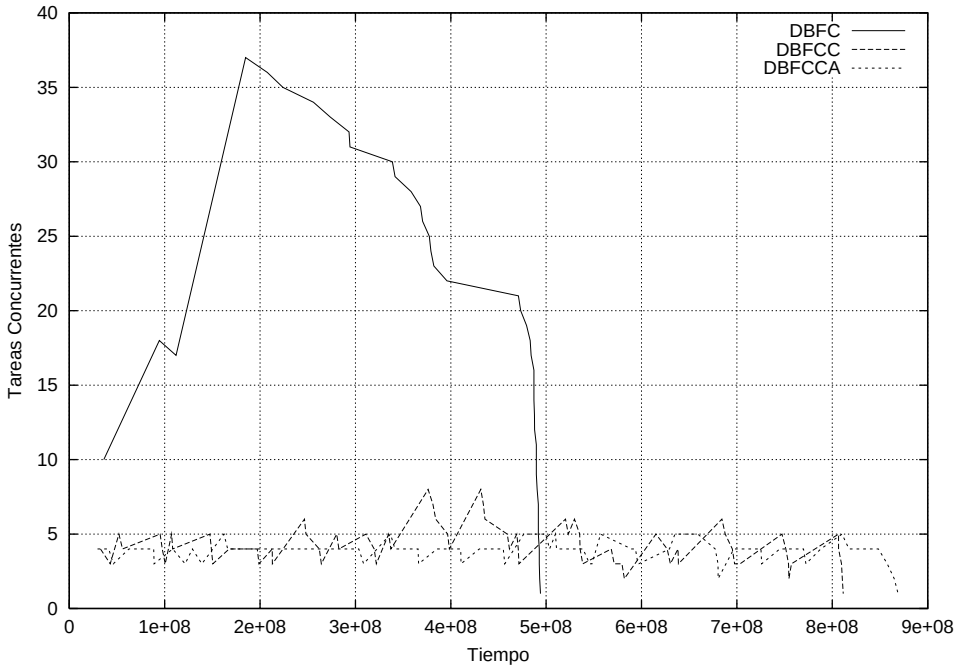


Figura 8.6: Nodo de mayor potencia: Número de tareas ejecutadas concurrentemente

caso de DBFCC y DBFCCA el número de tareas en ejecución permanece estable. Se puede observar como en el caso de DBFCCA, al incorporar el ajuste estimado de carga, el número de tareas no sufre cambios tan bruscos. En la gráfica 8.7 se puede observar la evolución del mismo parámetro para uno de los nodos de menor potencia[1]. En este caso se ve con mayor claridad los problemas de planificación debidos a que se dispone de un valor de carga no actualizado en tiempo real.

Las gráficas 8.8 y 8.9 muestran la comparación con el tiempo de ejecución de cada tarea. Como se puede observar según se incrementa al número de tareas ejecutadas simultáneamente, este factor aumenta rápidamente. En el caso de nodos de menor potencia se aprecia con mayor calidad la diferencia entre las políticas DBFCC y DBFCCA.

La gráfica 8.10 incorpora una variación de la política DBFCCA denominada DBFCCA-mgr que delega en el gestor del nodo parte del control. Para esta nueva política el gestor del nodo es capaz de bloquear la ejecución de ciertas tareas asignadas si el limite de carga actual excede el valor de corte. Como se puede observar los resultados obtenidos con DBFCCA y DBFCCA-mgr son equiparables. La política DBFCCA-mgr usa un modelo de control en dos niveles: un nivel de asignación global de tareas y un nivel de control directo sobre la carga del nodo.

[1]No se dispone de gráfica para el caso DBFC puesto que el nodo se colapsó debido al número de tareas

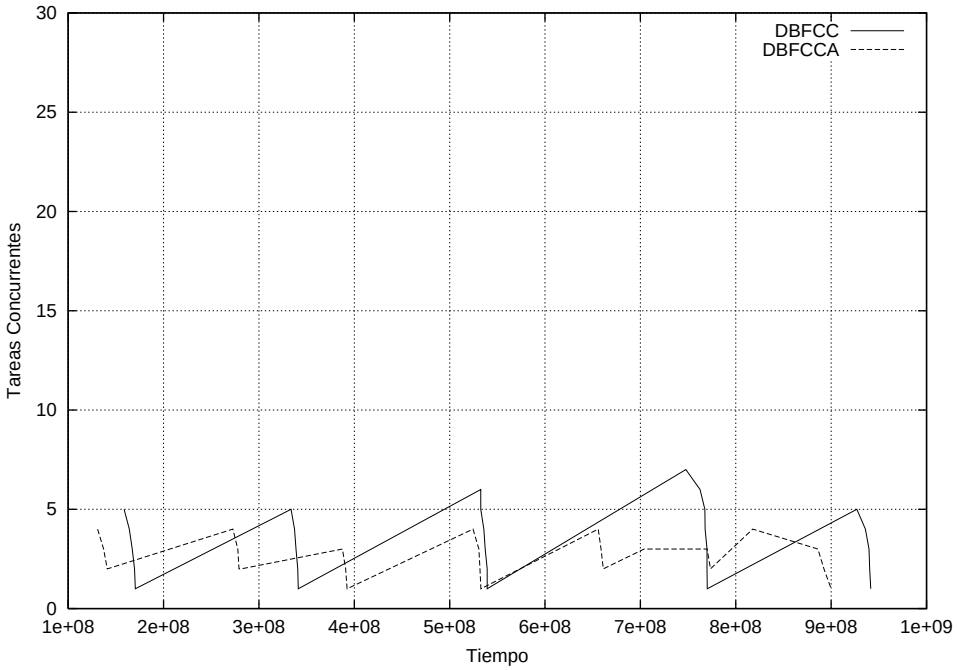


Figura 8.7: Nodo de menor potencia: Número de tareas ejecutadas concurrentemente

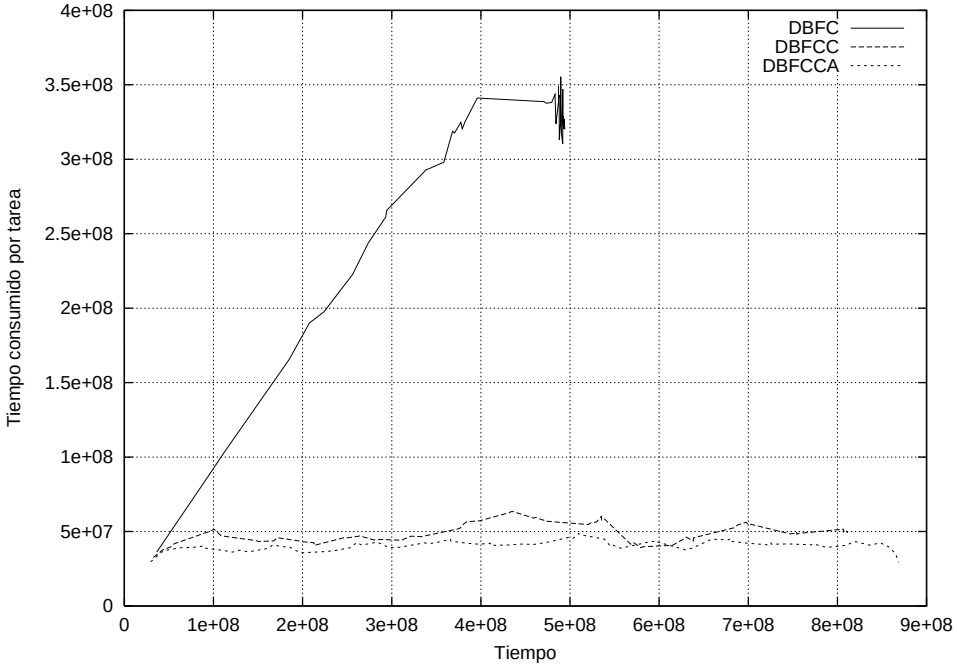


Figura 8.8: Nodo de mayor potencia: Tiempo de ejecución por tarea

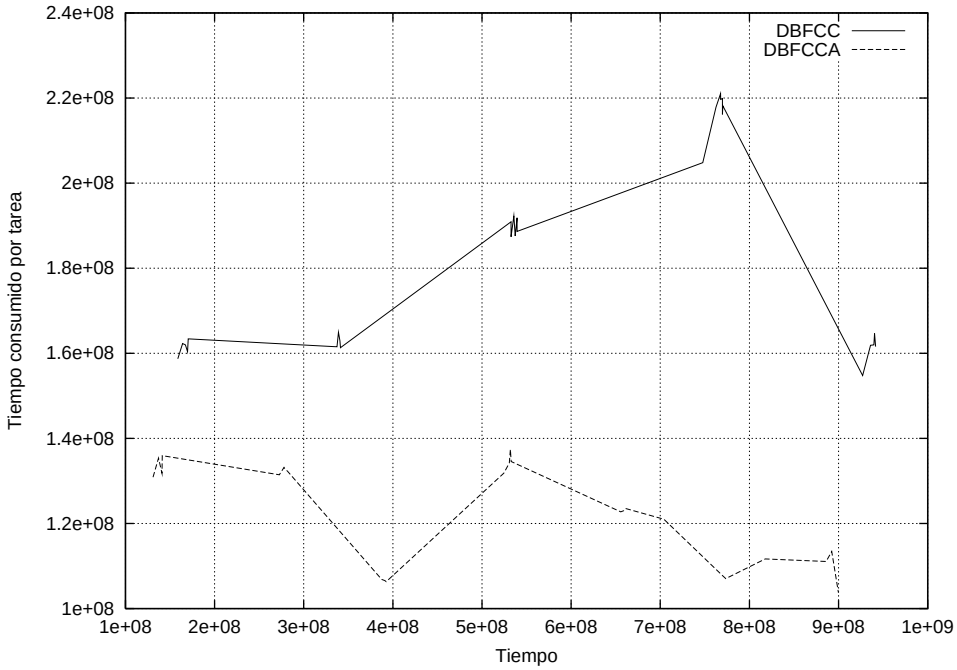


Figura 8.9: Nodo de menor potencia: Tiempo de ejecución por tarea

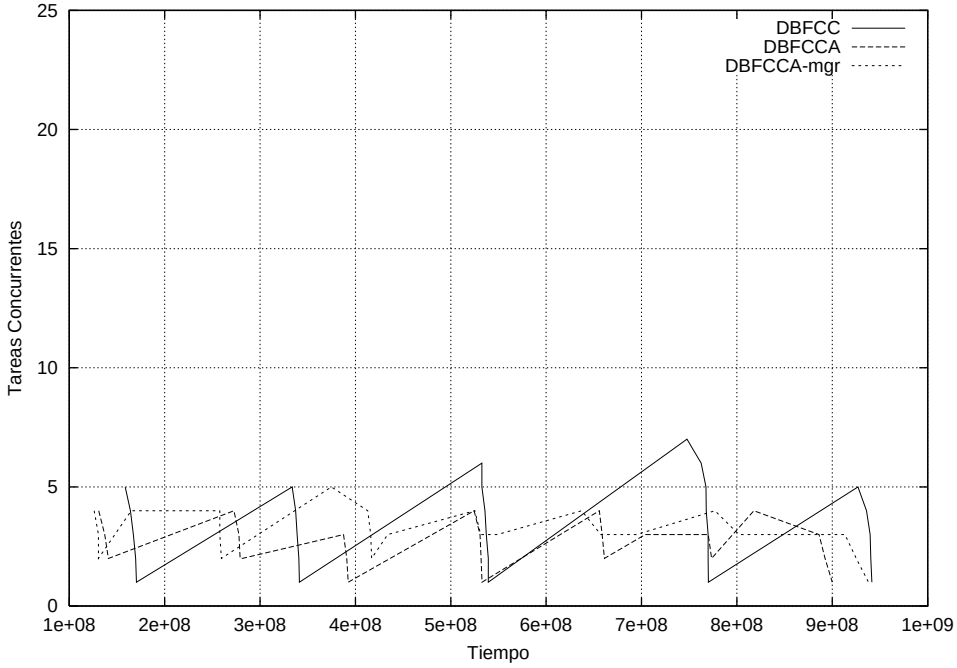


Figura 8.10: Nodo de menor potencia: Comparativa con la política DBFCAA-mgr

8.3.3 Políticas de Control Con Prioridades

Si en este escenario se desean incluir un tratamiento para tareas con diferentes prioridades, esta modificación puede realizarse directamente sobre las políticas de control.

Se proponen dos modelos de tratamiento de las prioridades:

① A nivel únicamente de planificador: El planificador encargado de distribuir los trabajos asigna el trabajo más prioritario de los aun pendientes en el momento en el que se cumplen las condiciones de asignación de trabajos (dependiendo de la carga). El gestor de cada nodo no gestiona prioridades y ejecuta todas las tareas equitativamente.

② A nivel de planificador y gestor: Al igual que en el caso anterior en planificador ordena los trabajos pendientes según prioridad. En este caso el gestor aborta todos los trabajos de baja prioridad en el momento en el cual un trabajo de alta prioridad se está ejecutando.

8.3.3.1 Resultados Experimentales

Para la realización de este experimento se ha preparado una batería de trabajos de baja prioridad como punto de partida del experimento añadiendo tareas de más alta prioridad en diferentes instantes:

❶ Se han insertado tareas de **prioridad media** en los instantes: $10segs.$, $32segs.$ (dos tareas) y $47segs.$ (otras 2 tareas).

❷ Se han insertado tareas de **prioridad alta** en los instantes: $12segs.$, $46segs.$, $82segs.$ (dos tareas) y $87segs.$.

Como se puede observar en la gráfica 8.11 la gestión de prioridades usando únicamente el planificador permite que las tareas de más alta prioridad sean iniciadas inmediatamente después de haber sido introducidas. Si se compara esta gráfica con 8.12 se puede comprobar que con la colaboración del gestor residente en el nodo es posible no sólo planificar dichos trabajos prioritarios antes que el resto de trabajos, sino que además su ejecución se realiza minimizando su tiempo de proceso, a expensas de tareas con menor prioridad.

El caso del procesamiento de prioridades en el planificador y en el gestor (gráfica 8.12) puede realizarse con diferentes estrategias. En el caso expuesto se ha optado por minimizar el tiempo de ejecución de las tareas de más alta prioridad. Esta estrategia garantiza que en ejecución no habrá nunca dos tareas de diferente prioridad. Si una tarea de prioridad más alta es remitida al nodo, éste aborta la ejecución de todas las tareas actuales y ejecuta únicamente la tarea prioritaria. Esta estrategia también incluye una modificación a nivel de las decisiones del planificador pues se permite

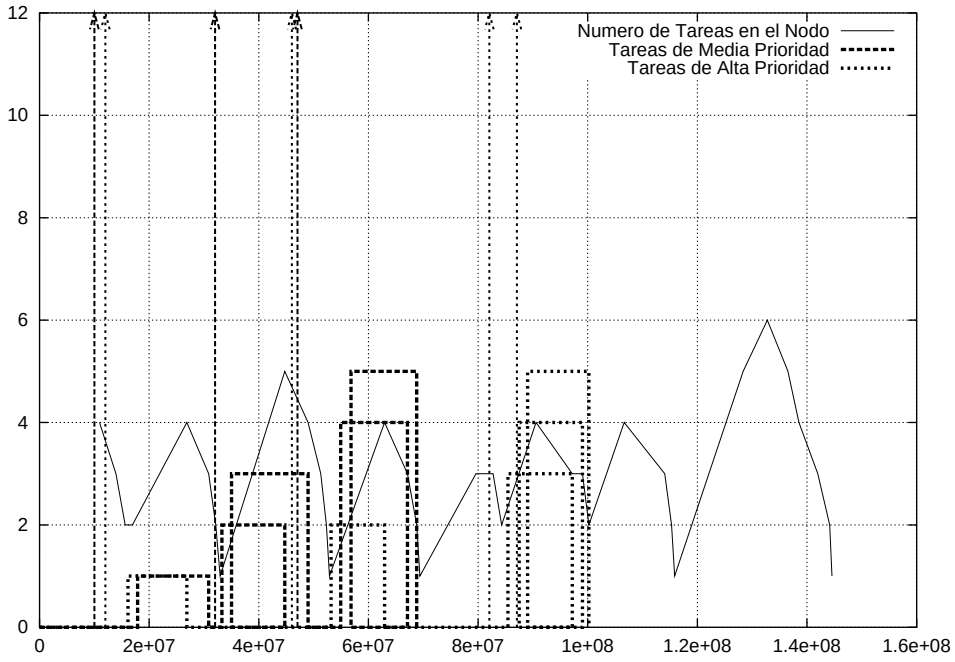


Figura 8.11: Procesamiento con prioridades en el planificador

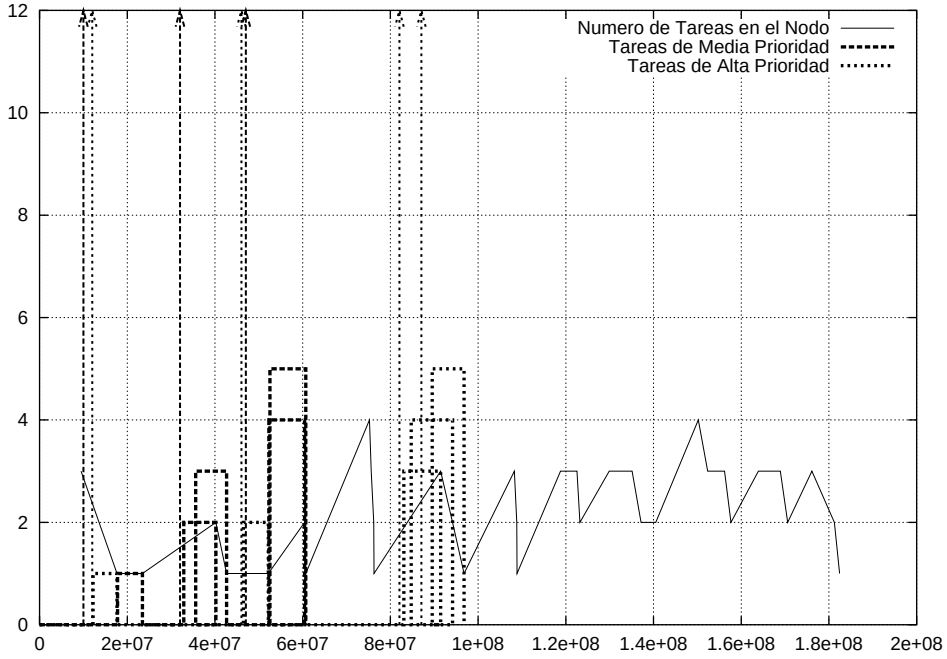


Figura 8.12: Procesamiento con prioridades en planificador y gestor

que se asignen trabajos a un nodo que tiene una carga superior al límite definido si dichos trabajos tiene una prioridad mayor que los actualmente en ejecución en el nodo. Esta modificación impide que se quede bloqueado un trabajo prioritario porque todos los nodos estén ejecutando muchas tareas de baja prioridad.

El tiempo medio de ejecución de cada tipo de tarea se puede contrastar en la tabla 8.1.

Tarea	**Planificador ①**	**Planificador y Gestor ②**	**Porcentaje de Reducción**
Tareas de baja prioridad	$11698.069ms$	$10570.102ms$	90.357%
Tareas de media prioridad	$12505.239ms$	$7274.232ms$	58.169%
Tareas de alta prioridad	$10993.462ms$	$7239.884ms$	65.856%

Tabla 8.1: Tiempo medio de procesamiento de cada tarea

8.3.4 Políticas de Control Restringidas a Recursos

El último caso abordado en este escenario viene a ilustrar otra de las propiedades deseadas de las políticas de control, los **mecanismos de validación**. Se desea resolver una versión ampliada del supuesto de distribución de trabajos consistente en las siguientes modificaciones:

- ❒ Las tareas llevan asociada una lista de recursos a utilizar.
- ❒ Los nodos disponen de un número finito de recursos de cada tipo.
- ❒ La ejecución de una tarea en un nodo implica la reserva de los recursos necesarios para dicha ejecución y su posterior liberación una vez finalizada.

Bajo estos supuestos se ha diseñado una política capaz de asignar un conjunto de tareas a unos nodos determinados. Dicha política se ha denominado **RES**.

> RES: Los trabajos son procesados en el orden definido por la cola de espera. Un trabajo es asignado a un nodo cuando este nodo dispone de los recursos necesarios para ejecutar dicho trabajo.

Bajo esta perspectiva es posible modelizar la carga de la máquina como un recurso más del nodo y estimar las características de la ejecución del trabajo como indicador del consumo de dicho recurso.

8.3.4.1 Resultados Experimentales

Si se aplica la política de control **RES** a un conjunto de trabajos, ésta garantiza que la asignación de trabajos a nodos no viola las restricciones de uso de recursos. Sin embargo esta política puede

plantear problemas en una serie de circunstancias:

- ❒ Si un trabajo requiere un número de recursos tal que no es posible encontrar un nodo que disponga de tantos recursos (entre libres y ocupados).
- ❒ Si un trabajo solicita un recurso que no se proporciona en ningún nodo.

Cualquiera de estos dos factores hacen que el planificador se bloquee y la distribución de trabajos se detenga. En ambos casos se trata de situaciones de error, pues dichos trabajos no podrán ser realizados. El problema no radica en los trabajos que no se puedan realizar sino en que un tipo de error de este estilo bloquea el sistema. La solución adecuada es eliminar dichas tareas o moverlas a una cola de trabajos no realizables y la implementación de dicha resolución no es compleja. Sin embargo la detección de este tipo de circunstancias no siempre es sencilla.

Para este caso se ha diseñado un proceso de verificación de las condiciones de aplicación de las políticas. Este mecanismo de validación realiza un análisis de las posibles entradas al agente de control y estudia las salidas generadas. El análisis de la validez de un conjunto de políticas de control require:

- ❒ La definición del dominio de los posibles datos de entrada. Indicando qué tipo de datos son y qué relación hay entre ellos. Esta definición debe ser exhaustiva y minuciosa.
- ❒ Indicación de cualquier restricción que debe ser cumplida por las acciones tomadas por el agente de control.
- ❒ Exactamente la misma política que se va a incluir en el sistema en ejecución.

En base a estas entradas el proceso de validación debe detectar las condiciones para las cuales los parámetros de entrada del sistema provocan que el agente de control no es capaz de generar un conjunto de acciones como respuesta. En el caso de generar acciones se debe verificar que éstas no violan las restricciones indicadas.

La definición de un mecanismo general de validación de políticas no es un asunto trivial debido a que está sujeto a multitud de factores complejos, tales como agentes de control con estado o memoria interna o dominios de entrada o salida infinitos. En el caso enunciado en este escenario se han formalizado las expresiones capaces de generar todo el dominio de datos de entrada (tareas y nodos) y de una forma particular para este caso se ha realizado la validación del conjunto de políticas de control. Las salidas obtenidas en este caso han sido del tipo:

```
UNRESOLVED CONFLICT
Task List: [[_#3(5..7)|_188]]
Resource List: [_#47(1..4),_#69(1..4)]
```

Dicha salida se interpreta como una condición de bloqueo del planificador, es decir como la condición para la cual el proceso de control da un error. Si la lista de tareas contiene una tarea que solicita algún recurso del 5 al 7 y los recursos disponibles van del 1 al 4, la política de control no asigna ninguna acción asociada. Si modificamos la política de control para incluir los casos ya citados y asignamos para tales casos la eliminación de la tarea, el proceso de verificación no encontrará ningún error.

8.3.5 Evaluación de Resultados

El conjunto de experimentos realizados como base el proceso de distribución de tareas han proporcionado tres diferentes resultados:

- ❒ Exploración de la flexibilidad de definición de políticas de control.
- ❒ El uso de políticas de control a varios niveles.
- ❒ Un estudio de la posibilidad de validar políticas de control.

8.3.5.1 Flexibilidad de las Políticas

En el caso de la distribución de trabajos sin factor de prioridad se han evaluado cuatro diferentes algoritmos (DBFC, DBFCC, DBFCCA y DBFCCA-mgr). Todos estos algoritmos son aplicables sobre los mismos componentes operacionales y su uso depende únicamente de las políticas de control cargadas. La modificación de las políticas de control automáticamente haría que el comportamiento del sistema sea diferente. Esta flexibilidad es adecuada para casos en los cuales el rendimiento esperado del sistema sea diferente y el entorno de uso sea distinto. Por ejemplo, la primera política (DBFC) para el caso de nodos de altas prestaciones concluye la ejecución antes que cualquier otra, sin embargo la carga a la que somete al nodo hace que no sea posible ejecutar cualquier otro tipo de proceso en el mismo nodo, además es una política que no es aplicable para ciertos nodos de menor rendimiento que pueden saturarse con el número de tareas. De forma análoga el resto de políticas u otras aplicables tiene ventajas e inconvenientes.

La aplicación de un modelo diferente de control, como sucede en el caso de soportar prioridades de trabajos, es una modificación que en los casos de sistemas tradicionales implican el rediseño de los mismos. En el caso de componentes MOIRAE, una modificación de este tipo sólo afecta a los algoritmos propios de control, es decir a las políticas, las cuales son externas al propio sistema en sí. Esto se ha comprobado al incluir gestión de prioridades al escenario original.

8.3.5.2 Políticas de Control Multinivel

La existencia de diferentes componentes de control de un sistema permite que las estrategias usadas sean aplicables en diferentes puntos. Éste es un factor más de flexibilidad y en algunos casos (como el algoritmo DBFCCA-mgr) una posibilidad de resolver problemas no aplicables desde modelos de control centralizado.

La existencia de agentes de control a diferentes niveles no restringe que cada uno de ellos sea diseñado de una forma diferente. Como se comentó en la arquitectura del sistema, se presentaron diferentes modelos de agentes de control, desde agentes reactivos hasta agentes con mayores capacidades de control. En este experimento se ha trabajado con un agente *Prolog* capaz de gestionar políticas complejas en el planificador y con un agente reactivo en los gestores de cada nodo.

8.3.5.3 Validación de Políticas

Las políticas de control son un elemento crítico en el rendimiento del sistema. La arquitectura MOIRAE permite que los componentes carguen dinámicamente sus políticas de control. Esta capacidad puede fomentar que las políticas de control incluidas dentro del sistema tengan algún tipo de error. Un error a ese nivel puede ser fatal para el sistema. La existencia de un mecanismo de validación de las políticas de control es un factor muy útil para solventar estos problemas. Sin embargo estos mecanismos de validación está restringidos a lenguajes de programación de las políticas de tipo declarativo y aun en esos casos plantea una alta complejidad.

8.4 Escenario 3: Algoritmos Distribuidos de Reglas de Asociación

Para finalizar se presenta un tercer escenario de experimentación en el cual se evalúa la aplicación de la arquitectura MOIRAE al diseño de una familia de algoritmos de *Data Mining*. El problema a resolver es el cálculo de reglas de asociación sobre bases de datos transaccionales. Una base de datos transaccional está compuesta por una serie de registros denominados transacciones. Una transacción es un conjunto de elementos denominados *items* acontecidos a la vez en una determinada operación. El caso mas habitual de bases de datos transaccionales son los registros de compras, por ejemplo de unos grandes almacenes.

Los conceptos relacionados con reglas de asociación se encuentran en [AIS93a] y son:

- Sea $\mathcal{I}$ el conjunto de todos los *items* y sea $\mathcal{T}$ el conjunto de todas las transacciones.
- Se dice que una transacción t contiene un conjunto de items (denominado *itemset*) X si X es

un subconjunto de los elementos de t.

- Un *itemset* X se dice que tiene un soporte s, indicándolo como $supp(X) = s$ si el $s\%$ de las transacciones de $\mathcal{T}$ contienen a X.
- Una regla de asociación es una implicación de la forma $X \Longrightarrow Y$, donde $X, Y \subset \mathcal{I}$ y $X \cap Y = \emptyset$.
- El soporte de una regla de asociación $X \Longrightarrow Y$ se calcula como el porcentaje de transacciones que contiene los *itemsets* de ambos lados de la regla. $supp(X \Longrightarrow Y) = supp(X \cup Y)$.
- La confianza c de una regla se define como el porcentaje de transacciones que incluyendo el *itemset* de la parte izquierda de la regla también contiene el de la parte derecha. $c = supp(X \cup Y)/supp(X)$.

El uso habitual de las reglas de asociación, en casos como el mencionado anteriormente de transacciones comerciales de unos grandes almacenes, es para determinar dependencias entre productos comprados por los clientes. Por ejemplo, si un cliente compra a y b también comprará c con un 85% de probabilidad.

8.4.1 Componentes del Experimento

El cálculo de reglas de asociación por medio de algoritmos centralizados se ha visto limitado por el tamaño de las bases de datos transaccionales. Retomando el caso de unos grandes almacenes el tamaño de sus bases de datos transaccionales es enorme tanto en número de registros como en el de *items* presentes en dichas transacciones. Los métodos genéricos de extracción de reglas de asociación se basan en el algoritmo *Apriori* [AIS93a], consistente en varias pasadas iterativas en las cuales son calculados los *itemsets* más frecuente de tamaño $1, 2, 3, \ldots$ consecutivamente. Una vez obtenidos dichos *itemsets* se combinan para conformar las reglas. El algoritmo *Apriori* tiene como fase computacionalmente más costosa el cálculo de los *itemsets* debido a dos factores:

① Cada iteración consiste en el recorrido sistemático de la base de datos transaccional completa. En casos reales dicha base de datos no puede ser residente en memoria debido a su tamaño.

② En cada iteración cada una de las transacciones es comparada con con los *itemset* candidatos. En la primera iteración los candidatos son los conjuntos de *items* tomados de uno en uno, en las sucesivas iteraciones estos candidatos se construyen por la combinación de *itemsets* mayoritarios de transacciones anteriores.
Dependiendo de las características de los datos de entrada (distribución y número de *items*) en ciertas iteraciones se produce una explosión del número de *itemsets* candidatos. En problemas reales el número de estos candidatos es suficientemente grande para no poder ser almacenados en memoria.

Shintani y Kitsuregawa [SK99] proponen diferentes algoritmos distribuidos para realizar el proceso de cálculo de reglas de asociación sobre un *cluster* de estaciones de trabajo. Las fases en las que se divide el cálculo de los *itemsets* con soporte superior a un valor determinado son las siguientes [AIS93a]: (i) Generación de *itemsets* candidatos, (ii) Recorrido de la tabla de datos y cuenta, (iii) Selección de los *itemsets* con mayor soporte. Estas fases son repetidas en cada iteración, para el cálculo de los *itemsets* de tamaño $T+1$. Para resolver los dos problemas (① y ②) antes citados los autores plantean varios algoritmos:

❶ **H–HPGM (hash)** (*Hierarchical Hash Partitioned Generalized association rule Mining*): Por medio de una función *hash* se asocia un determinado nodo de proceso para cada uno de los *itemsets* candidatos. Durante la fase de recorrido de los datos cada nodo procesa parte de la base de datos transaccional y se intercambian mensajes para actualizar los contadores de los *itemsets* que no están residentes en cada nodo.

❷ **H–HPGM (stat)** (*H–HPGM with stats*): El mecanismo de distribución por medio de la función *hash* reparte homogéneamente los candidatos entre todos los nodos, sin embargo la frecuencia de dichos candidatos no es igual. El algoritmo **H–HPGM (hash)** es proclive a generar nodos que sean cuellos de botella al concentrar un gran número de mensajes de otros nodos, debido a que gestionan varios de candidatos con alta frecuencia de aparición.
El algoritmo **H–HPGM (stat)** propone un mecanismo de distribución de candidatos ponderado por una estimación estadística de la frecuencia de aparición de dicho *itemset*.

❸ **H–HPGM–TGD (stat+)** (*H–HPGM with Tree Grain Duplication*): El proceso de asignación de candidatos a cada nodo del algoritmo **H–HPGM (stat)** por lo general no usa toda la memoria disponible en cada uno de estos nodos. Como mejora al algoritmo anterior se puede usar el espacio de memoria disponible en cada nodo para replicar los *itemsets* más frecuentes. Esta medida permite reducir el número de mensajes intercambiados entre los nodos.

Para una descripción completa de los algoritmos se refiere al lector al articulo [SK99]. Shintani y Kitsuregawa incorporan dos versiones adicionales del último algoritmo: **H–HPGM–PGD (stat+)** (*H–HPGM with Path Grain Duplication*) y **H–HPGM–FGD (stat+)** (*H–HPGM with Fine Grain Duplication*) aplicables al cálculo de reglas de asociación sobre jerarquías conceptuales.

8.4.2 Políticas de Control Usadas

Para la resolución de esta tarea se ha planteado un esquema de componentes idéntico al presentado en la sección 6.6. La figura 8.13 muestra de nuevo la distribución de componentes usada en la federación de procesamiento de datos.

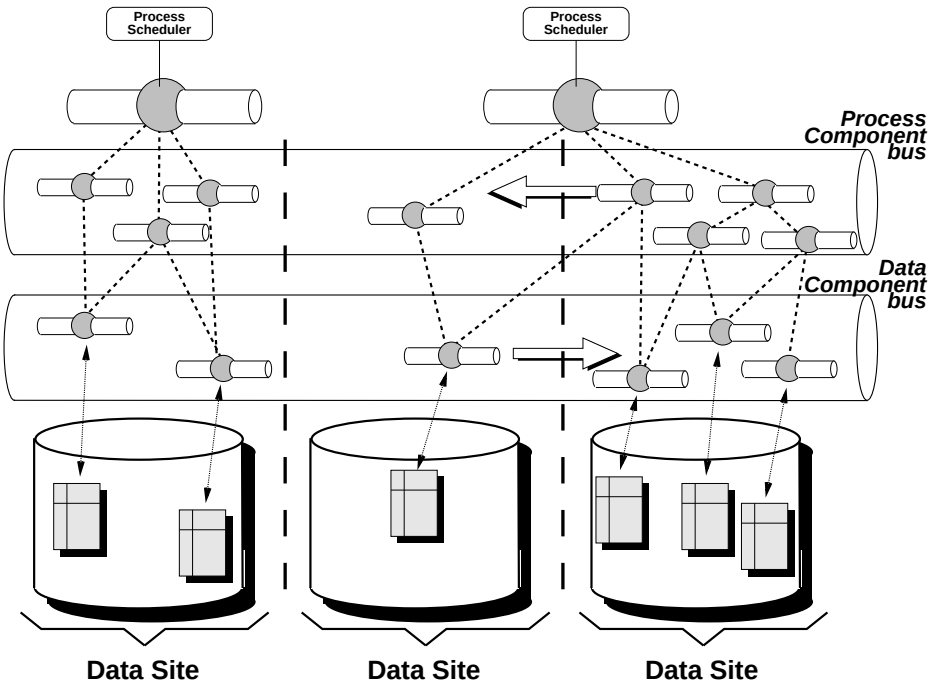


Figura 8.13: Componentes de la Federación de Procesamiento de Datos

El componente *Process Scheduler* hará las veces de elemento de coordinación del algoritmo. Su tarea principal será la asignación de *itemsets* candidatos a los diferentes nodos en cada iteración. Este componente seguirá el esquema propuesto en cada uno de los algoritmos a la hora de realizar la distribución. Por su parte, los componentes de cálculo (uno por nodo de proceso) serán los encargados de recorrer el fragmento de tabla asignado al nodo y realizar el proceso de cuenta del soporte de los *itemsets*.

Las políticas desarrolladas para este experimento son la versión para el componente de coordinación (*Process Scheduler*) y para el componente de proceso asociado a cada nodo. En este experimento se podrán comprobar los retardos asociados a la delegación y propagación de control enunciados en las funcionalidades de la arquitectura MOIRAE. Asimismo se podrá comparar a nivel de eficiencia políticas de control más complejas que las usadas en el primer experimento.

8.4.2.1 Resultados Experimentales

Para evaluar las implementaciones de estos algoritmos se han usado unos datos sintéticos generados automáticamente. Estos datos se han construido en base a unos parámetros de forma que a priori se conoce su distribución. Como en los casos reales la explosión combinatoria de candidatos ocurre entre la segunda y la tercera iteración. Son estas iteraciones las que se ven prin-

cipalmente afectadas por los algoritmos antes mencionados, pues en el resto de casos el conjunto de candidatos es reducido y puede ser almacenado en la memoria de cada nodo.

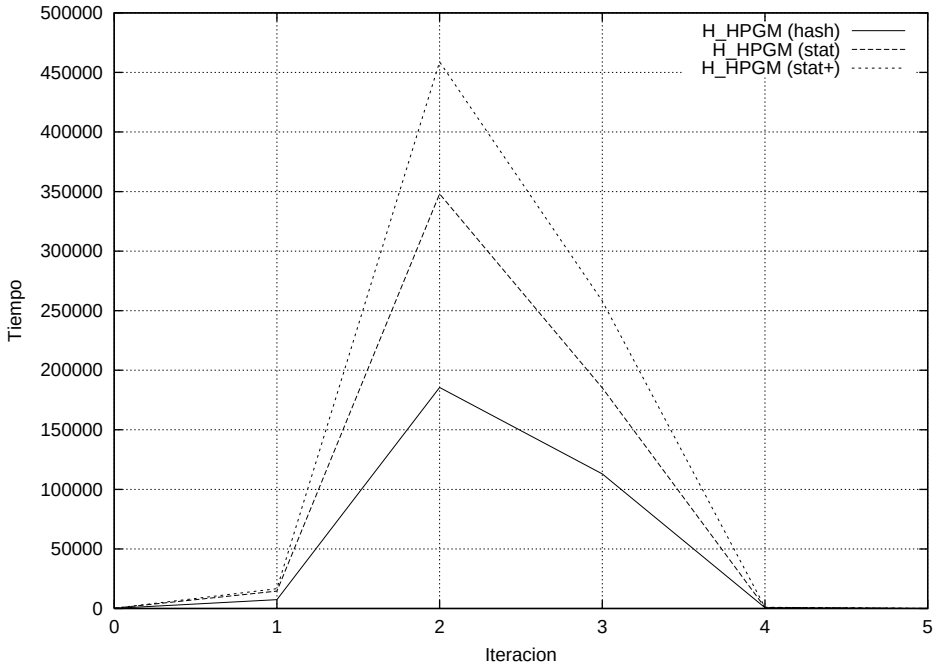


Figura 8.14: Tiempo de distribución de candidatos: Programado

En las gráficas 8.14 y 8.15 muestran el tiempo de distribución de candidatos requerido por el nodo de coordinación. El tiempo computado se restringe únicamente al de la toma de la decisión no al de transmisión del candidato al nodo. Como se puede observar tanto en el caso de distribución programada (gráfica 8.14) como en el de políticas dinámicas de asignación (gráfica 8.15), el tiempo de evaluación de los algoritmos mantiene la misma progresión.

Si se compara algoritmo a algoritmo cada una de las implementaciones, se tiene que en el caso de **H–HPGM (hash)** (figura 8.16) la diferencia entre el algoritmo programado y las políticas de control es superior al 240%. En el caso del algoritmo **H–HPGM (stat)** (figura 8.17) esta diferencia es de un 107%, mientras que para el último caso, **H–HPGM (stat+)** (figura 8.18), tan sólo es de un 95%.

En cualquier caso, el tiempo consumido en cada iteración por el proceso de distribución de candidatos es despreciable en relación al proceso de recorrer la base de datos transaccional de cada uno de los nodos. Una vez distribuidos los candidatos cada nodo procesa su fragmento de la base de datos transaccional. Para cada registro leído actualiza los contadores de los *itemsets* presentes en dicha transacción. Si el *itemset* se almacena localmente dicho valor se acumula, en otro caso se

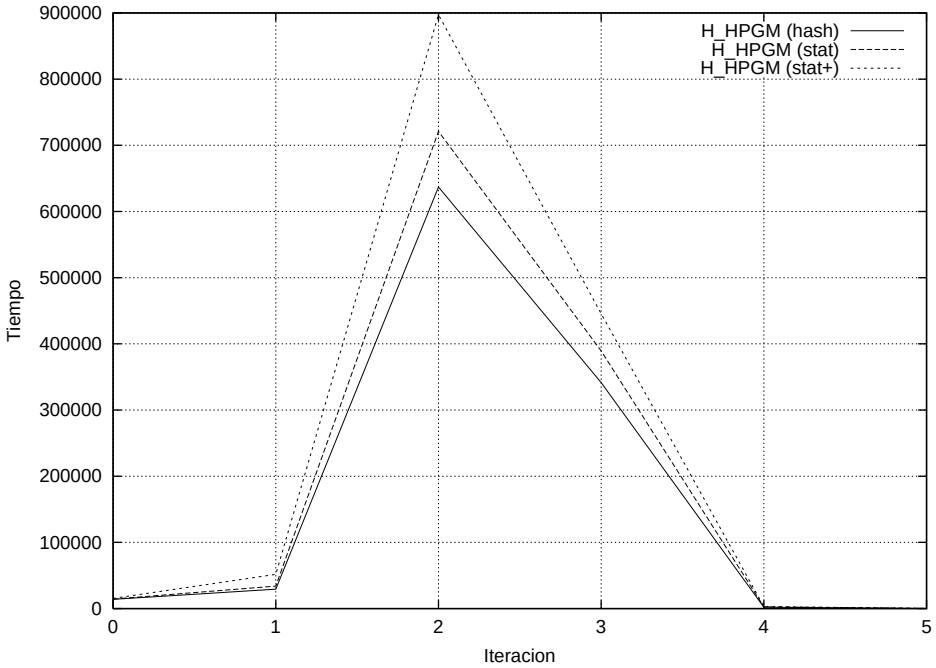


Figura 8.15: Tiempo de distribución de candidatos: Políticas Dinámicas

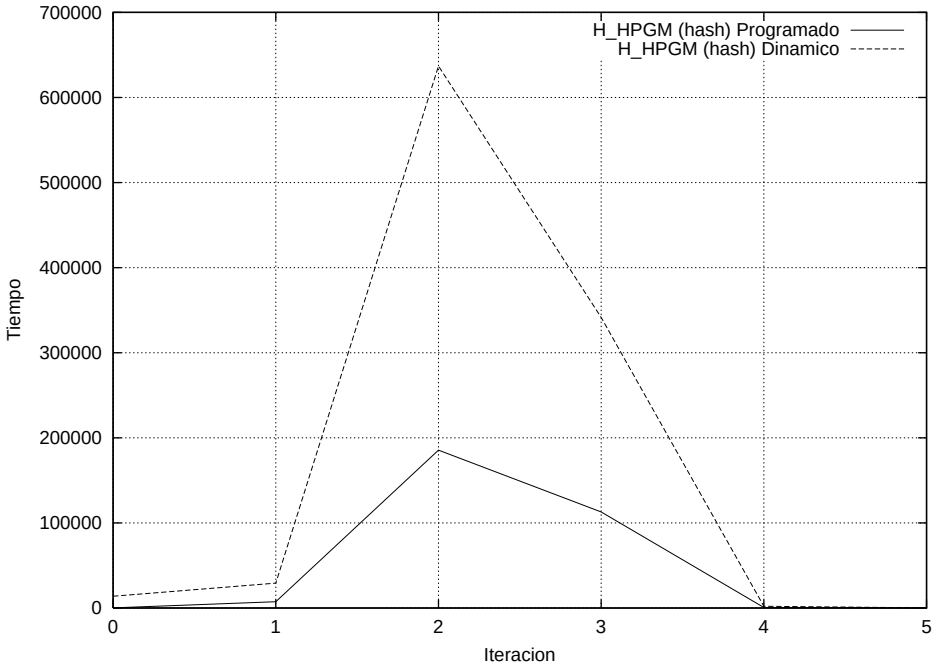


Figura 8.16: Tiempo de distribución de candidatos: Comparativa para H-HPGM (hash)

almacena en un *buffer* de actualizaciones pendientes. Cuando este *buffer* está lleno se transmite un mensaje *broadcast* a todos los nodos para que actualicen los contadores de dichos *itemsets*.

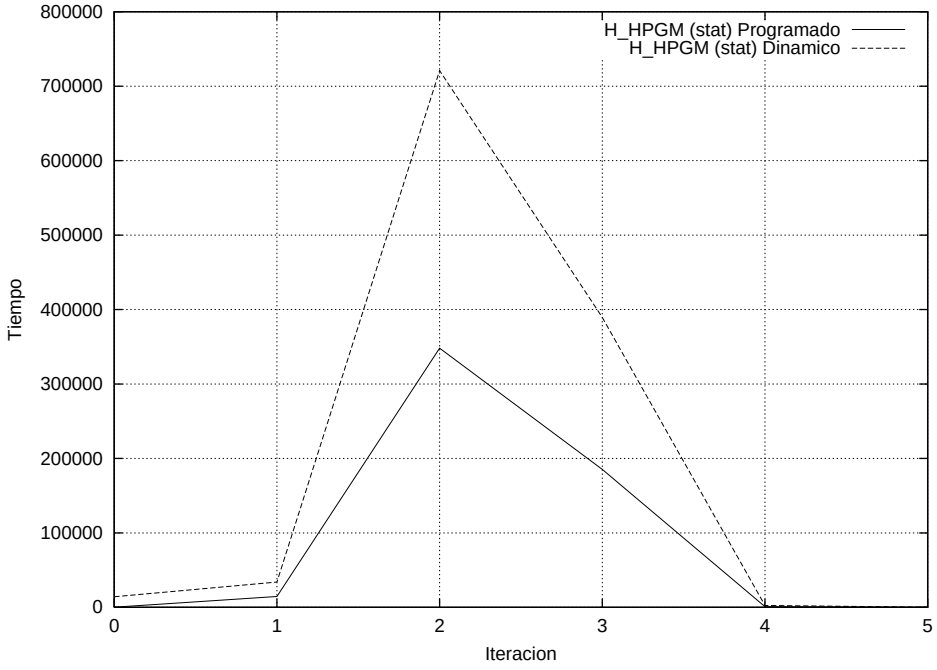


Figura 8.17: Tiempo de distribución de candidatos: Comparativa para H-HPGM (stat)

Fase	**Control Programado**	**Política Reactiva**	**Porcentaje de Incremento**
Primer fallo de localización del *itemset* (incluye propagación–delegación de control)	$12.11ms$	$198.12ms$	1536.0033%
Siguientes fallos de localización (sin propagación–delegación de control)	$12.17ms$	$12.98ms$	6.6557%

Tabla 8.2: Tiempo acumulado en fallos de localización de *itemsets*

Las decisiones de control de los elementos de cálculo se han implementado por medio de agentes de control reactivos. Estos agentes disponen de una tabla de conflictos (eventos que activan el plano de control). A cada conflicto se le asocia una función de tratamiento. En este caso, al conflicto derivado de no encontrar el *itemset* entre los candidatos locales (`itemset_not_found`) se asocia la operación de insertar en el *buffer*. Si el *buffer* está lleno (`buffer_is_full`) se activa la función de transmisión del mensaje *broadcast*. La tabla 8.2 recoge el tiempo acumulado en estas tareas de control para los casos de control programado y de políticas reactivas.

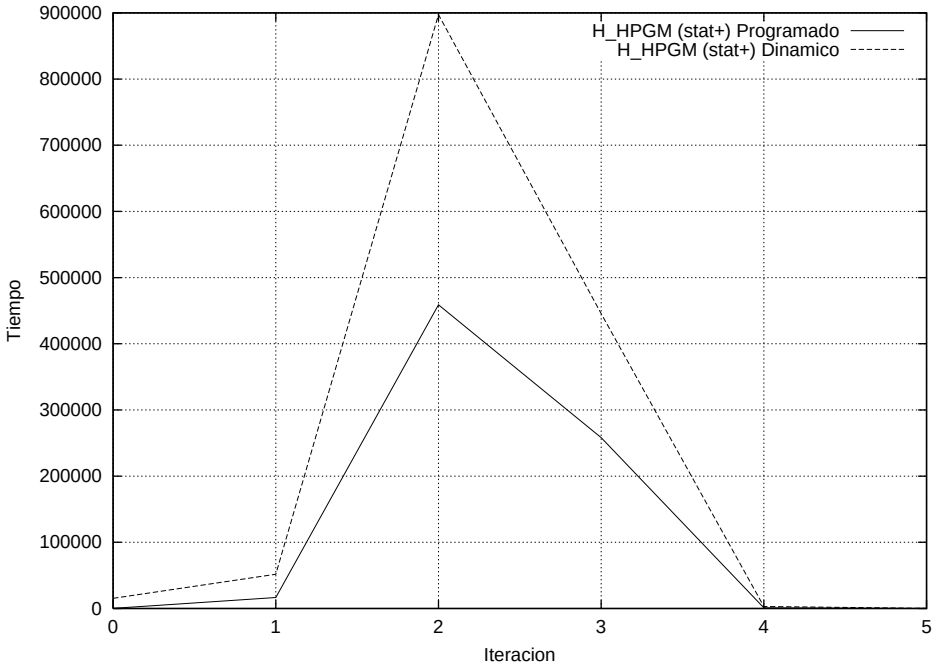


Figura 8.18: Tiempo de distribución de candidatos: Comparativa para H-HPGM (stat+)

Una alternativa a los casos de algoritmos vistos en este escenario puede realizarse por medio de modelos mixtos. En dichos modelos se usa para ciertas iteraciones un algoritmo de distribución y en otras se utiliza uno diferente. Esta alternativa es perfectamente realizable por medio del modelo de control de la arquitectura MOIRAE.

8.4.3 Evaluación de Resultados

En un escenario más complejo que el presentado en el primer experimento se pueden comparar los tiempos de control para dos niveles diferentes:

① La distribución de candidatos es la principal diferencia entre los distintos algoritmos planteados. Los procesos de asignación de *itemsets* candidatos a un nodo u otro son muy variados en relación a los parámetros de decisión tomados y al orden en el cual los candidatos son asignados. Las políticas de control definidas en *Prolog* resultan útiles debido a la flexibilidad en la definición de algoritmos, su pérdida de eficiencia en la ejecución es asumible en un proceso que se repite una vez por iteración (considerando que el tiempo medio de una iteración es de varios minutos).

② El tratamiento de los fallos producidos al no encontrar un *itemset* entre los candidatos loca-

les es mucho más frecuente que el caso anterior. La evaluación de esta acción de control es como consecuencia mucho más sensible a reducciones de eficiencia. La opción de un mecanismos de control tradicional es válida aunque una solución basada en un agente de control reactivo (no implementado en *Prolog* sino como una tabla estímulo–respuesta) no degrada el rendimiento y proporciona un cierto factor de flexibilidad.

Este experimento da una buena visión del ámbito de aplicación de la arquitectura MOIRAE a sistemas complejos. La decisión del agente de control a usar: reactivo o deliberativo (*Prolog*), o de incluir o no agente de control debe llegar del análisis de decisiones tales como:

- Si la eficiencia en la toma de dicha decisión es clave para el rendimiento del sistema.
- Las posibles diferencias entre los modelos de control actuales y planteamiento futuros. Si el grado de libertad a la hora de definir políticas de control es suficientemente ámplio para comprometer la eficiencia.
- Si en alguna circunstancia puede resultar útil cambiar las políticas de control dinámicamente
- Las acciones que se pueden realizar y la información que necesitan los algoritmos de control planteados.

Parte III

CONCLUSIONES Y LÍNEAS FUTURAS

Capítulo 9

CONCLUSIONES

Índice General

Los objetivos fijados como meta del trabajo de investigación han sido alcanzados y se ha definido con éxito una solución para un sistema de *Data Mining* integrado, distribuido, extensible y de configuración flexible. Estos objetivos se han conseguido en base a la definición de MOIRAE, una arquitectura genérica para sistemas distribuidos de control flexible y el diseño general de un sistema de *Data Mining* sobre dicha arquitectura.

Las aportaciones extraidas tras la realización de este trabajo pueden enmarcarse en dos grandes grupos. Por un lado, la arquitectura MOIRAE con sus características han proporcionado una serie de funcionalidades al desarrollo de sistemas distribuidos flexibles. Por otro lado, el diseño del sistema DI-DAMISYS verifica las ventajas de la arquitectura MOIRAE y agrupa una serie de características beneficiosas para un sistema de *Data Mining*.

9.1 Aportaciones de la Arquitectura MOIRAE

La arquitectura de control distribuida MOIRAE permite experimentar con diferentes estrategias para la resolución de problemas derivados de un sistema distribuido complejo, tales como:

- El equilibrio de carga entre tareas y nodos de computación.

- ❒ La gestión de recursos compartidos entre procesos que compiten por ellos.
- ❒ El arbitraje de prioridades entre tareas.
- ❒ Los factores de seguridad derivados de la autenticación delegada y los ataques por denegación de servicios.
- ❒ La tolerancia a fallos por medio de servicios redundantes y reconfiguración dinámica del sistema.
- ❒ La eliminación y adición de funcionalidades y servicios al sistema de forma dinámica.
- ❒ La administración y auditoría de sistemas complejos.

Estos problemas son sólo un reducido ejemplo de los abordables por medio de arquitecturas de control dinámicas, y en concreto por MOIRAE. La separación entre funcionalidades de control y las tareas operacionales del sistema permiten que las decisiones relativas, tanto al comportamiento general del sistema como a componentes particulares del mismo sean modificables de forma fácil y flexible sin rediseñar ni el sistema ni los componentes.

La arquitectura MOIRAE no es en sí una estrategia para resolver ninguno de estos problemas, es únicamente un marco de desarrollo que permite que un sistema desarrollado convenientemente sobre ella pueda modificar sus políticas de control para plantear diferentes soluciones a cualquiera de estas problemáticas.

Como todo modelo de diseño, éste tiene sus limitaciones y sus áreas apropiadas de aplicación. Las limitaciones se circunscriben casi por completo a cuestiones de eficiencia. El modelo de evaluación de políticas dinámicas es ligeramente menos eficiente que el de decisiones estáticas y para problemas en los cuales tal factor sea decisivo ésta es una decisión a tener en cuenta. Como alternativa se pueden definir modelos dinámicos con menores funcionalidades pero mayor eficiencia. Tales soluciones son los ya mencionados agentes reactivos.

En el estudio de las áreas de aplicación de la arquitectura hay que tener en cuenta el nivel de detalle en el cual es apropiado hacer un análisis de este tipo. En los experimentos se muestra que la aplicación de esta arquitectura a un problema implica analizar el mismo en la parte operativa y en la parte de control. Este análisis no se debe realizar para todos y cada uno de los elementos de un sistema, sino sólo para aquellos cuya responsabilidad, la capacidad de cambiar su funcionamiento o su ámbito de aplicación lo hagan apropiado. Como en muchos otros casos, ésta es una decisión de diseño.

Junto con las aportaciones prácticas de esta arquitectura, resulta imprescindible desatacar la representación formal desarrollada. Se ha extendido el modelo de autómata de entrada/salida definido por Tuttle para incluir como un concepto diferente las decisiones de control, separadas de las funciones operacionales. La formalización de la arquitectura incluye el mecanismo para traducir todo autómata de entrada/salida a un autómata de control MOIRAE. Permitiendo de esta forma aprovechar los mecanismos formales de análisis y validación de sistema distribuidos asíncronos.

9.2 Aportaciones del Sistema DI-DAMISYS

En primer lugar, como resultado aplicable de este trabajo se han extraído las características de los sistemas actuales de *Data Mining* distribuido. Clasificando dichos sistemas en base a una serie de funcionalidades proporcionadas. Dicho análisis siempre es aplicable a cualquier trabajo derivado de esta línea que pretenda extender las capacidades de estos sistemas.

Como resultado de este análisis y en base a la arquitectura MOIRAE de control se ha diseñado un sistema de *Data Mining* distribuido. Dicho sistema tiene las características planteadas al comienzo del trabajo:

- *Integración* con los sistemas gestores de bases de datos, al poder incorporar las funcionalidades que un SGBD proporciona junto con las funciones de análisis de datos de un sistema de *Data Mining*. Esta característica permite que el mismo sistema actúe tanto como un sistema de gestión como de análisis de datos.
- *Extensibilidad* de sus funcionalidades, requerida por la creciente investigación en el campo de nuevas técnicas de *Data Mining* que desarrollan nuevos algoritmos de análisis, preparación o procesamiento de los datos. Todas estas nuevas funcionalidades pueden ser incluidas en el sistema, sin requerir el rediseño del mismo e incluso sin detener la ejecución del mismo.
- *Distribución* del sistema, necesaria para el mejor aprovechamiento de los recursos de computación de un grupo de computadoras interconectadas en red. Las necesidades de potencia de cálculo son, de esta forma, alcanzadas por medio del procesamiento distribuido y en paralelo de operaciones con gran consumo de CPU y memoria. La distribución en el campo de *Data Mining* también permite la utilización de fuentes de datos originalmente distribuidas y cuya centralización no es abordable por problemas tecnológico o por consideraciones administrativas o de privacidad.
- *Flexibilidad* de control, proporcionada por medio de su diseño sobre la arquitectura MOIRAE. Esto permite que el sistema disfrute de todas las ventajas derivadas de las funcionalidades de control definidas por la arquitectura.

Capítulo 10

LÍNEAS FUTURAS

Índice General

Si bien todos los objetivos fijados han sido alcanzados, el trabajo realizado a lo largo de esta Tesis ha dejado abiertas nuevas líneas de investigación y desarrollo para futuros trabajos situados dentro de este mismo campo. En primer lugar, se pueden resaltar las líneas de desarrollo dirigidas hacia la consecución tanto de un prototipo de evaluación del sistema como el desarrollo del sistema completo. En segundo lugar, nuevas aportaciones dentro del marco de investigación de arquitecturas de control dinámicas pueden involucrarse como continuación de esta Tesis.

10.1 Líneas de Desarrollo

10.1.1 Herramientas de Desarrollo MOIRAE

La definición y descripción de componentes en MOIRAE se realiza por medio de la sintaxis de dos lenguajes, uno de definición del componente y otro de implementación. El desarrollo de una herramienta automática capaz de utilizar la descripción del componente en dichos términos y la generación posterior del esqueleto del componente, a falta de la implementación concreta de

ciertas operaciones, permitiría el desarrollo de sistemas en base a la arquitectura MOIRAE de una forma productiva.

Dicha aplicación y el conjunto de herramientas asociadas para depuración, verificación y documentación constituiría un entorno de desarrollo rápido de sistemas distribuidos.

10.1.2 Implementación del Sistema

Una vez desarrolladas las herramientas de la arquitectura MOIRAE, el desarrollo completo del sistema sería fácilmente abordable. El desarrollo y explotación del sistema permitiría evaluar su rendimiento y funcionalidades contra otros sistemas tanto comerciales como experimentales dentro del campo de *Data Mining*.

10.1.3 Aplicación de la Arquitectura MOIRAE a otros Entornos

MOIRAE es una arquitectura genérica para sistemas distribuidos. Si bien, en este trabajo se ha orientado su aplicación hacia el campo de sistemas de *Data Mining*, su aplicabilidad es extrapolable a otros campos. Cualquier sistema distribuido complejo que esté compuesto por elementos que se añaden o eliminan de forma dinámica del sistema puede ser desarrollado en base a la arquitectura MOIRAE. Dentro de este tipo de aplicaciones se pueden destacar sistemas de control industrial o sistemas automáticos de monitorización, tales como centrales térmicas, cadenas de producción, etc. Otros sistemas a los que se pueden aplicar estas técnicas son entornos complejos en que los factores como la seguridad, la tolerancia a fallos o la operatividad ante diferentes tipos de circunstancias esté garantizada, por ejemplo sistemas transaccionales de grandes compañías o sistemas de información de las organizaciones.

10.2 Líneas de Investigación

10.2.1 Definición Dinámica de Políticas

En la arquitectura propuesta, las políticas de control son suministradas de forma externa al sistema. La arquitectura MOIRAE es la encargada de gestionar las políticas, distribuirlas entre los componentes, aplicarlas y mantenerlas actualizadas. Este proceso se basa en la definición por parte de un administrador de las políticas que considere adecuadas para el sistema, en base a criterios tales como el comportamiento requerido del sistema o el criterio de utilización de ciertos recursos.

Una interesante línea de investigación sería la asociada a la definición de nuevas políticas o la variación de las existentes por parte del sistema. De esta forma el sistema sería capaz de regular

automáticamente su propio comportamiento. Estas modificación de las decisiones de control del sistema debería encontrarse restringida dentro de unos límites definidos externamente, para impedir que el sistema se excediera en el uso de ciertos recursos o que determinadas operaciones quedaran bloqueadas. Un escenario de aplicación de esta nueva línea sería la configuración del sistema dentro de unos parámetros determinados, dejando que el sistema se auto-configurara en otros aspectos dinámicamente.

10.2.2 Experimentación de ciertas Estrategias de Control

Como se enunció en las conclusiones de este trabajo, la arquitectura MOIRAE no representa una estrategia para resolver conflictos concretos de los sistemas distribuidos sino que actúa como un lecho de desarrollo y experimentación de nuevas estrategias para resolver dichos problemas. Un investigador interesado en algoritmos, por ejemplo, de equilibrado de carga, puede usar la arquitectura MOIRAE para evaluar diferentes estrategias de reparto o gestión de tareas en un sistema desarrollado sobre dicha arquitectura. Se considera que dicha aportación podría ser de gran interés para multitud de investigadores en diferentes campos dentro del área de sistemas distribuidos.

Bibliografía

[AC87] P. Agre and D. Chapman. PENGI: An implementation of the theory of activity. In *P*roceedings of the Sixth National Conference on Artificial Intelligence, pages 268–272, 1987.

[ACF+94a] R. Agrawal, M. Carey, C. Faloutson, S. Ghosh, A. Houtsma, T. Imielinski, B. Iyer, A. Mahboob, H. Miranda, R. Srikant, and A. Swami. Quest: A project on database mining. *S*IGMOD Record (ACM Special Interest Group on Management of Data), 23(2):514–514, June 1994.

[ACF+94b] Rakesh Agrawal, Michael J. Carey, Christos Faloutsos, Sakti P. Ghosh, Maurice A. W. Houtsma, Tomasz Imielinski, Balakrishna R. Iyer, A. Mahboob, H. Miranda, Ramakrishnan Srikant, and ArunÑ. Swami. Quest: A project on database mining. In Richard T. Snodgrass and Marianne Winslett, editors, *P*roceedings of the 1994 ACM SIGMOD International Conference on Management of Data, page 514, Minneapolis, Minnesota, 24–27 May 1994.

[ACM98] ACM, editor. *SC*'98: High Performance Networking and Computing: Proceedings of the 1998 ACM/IEEE SC98 Conference: Orange County Convention Center, Orlando, Florida, USA, November 7–13, 1998. ACM and IEEE, 1998.

[AIS93a] Rakesh Agrawal, Tomasz Imielinski, and Arun Swami. Mining association rules between sets of items in large databases. *S*IGMOD Record (ACM Special Interest Group on Management of Data), 22(2):207–216, June 1993.

[AIS93b] Rakesh Agrawal, Tomasz Imielinski, and ArunÑ. Swami. Mining association rules between sets of items in large databases. In Peter Buneman and Sushil Jajodia, editors, *P*roceedings of the 1993 ACM SIGMOD International Conference on Management of Data, pages 207–216, Washington, D.C., 26–28 May 1993.

[AK91] H. Aït-Kaci. *W*arrem's Abstract Machine, A Tutorial Reconstrution. Logical Programming Series. AAAI Press/MIT Press, 1991.

[AMS+96] Rakesh Agrawal, Manish Mehta, John Shafer, Ramakrishnan Srikant, Andreas Arning, and Toni Bollinger. The quest data mining system. In Simoudis et al. [SHF96], page 244.

[ANS93] ANSI. Database language – SQL. American National Standard X3.135–1992, January 1993.

[ANS98] ANS. Knowledge interchange format. Draft Proposed American National Standard (dpANS),NCITS.T2/98-004, 1998.

[AP95] R. Agrawal and G. Psaila. Active data mining. In *P*roceedings 1st International Conference on Knowledge Discovery in Data Bases and Data Mining, 1995.

[ARP93] ARPA. Specification of the kqml agent-communication language. The DARPA Knowledge Sharing Initiative External Interfaces Working Group, 1993. Defined by Tim Finin and others.

[AS96a] R. Agrawal and J. C. Shafer. Parallel mining of association rules. *I*eee Trans. On Knowledge And Data Engineering, 8:962–969, 1996.

[AS96b] Rakesh Agrawal and Kyuseok Shim. Developing tightly-coupled data mining applications on a relational database system. In Simoudis et al. [SHF96], page 287.

[Aus62] J.L. Austin. *H*ow to Do Things with Words. Harvard University Press, Cambridge, Massachusetts, 1962.

[BCG+99] S. Bailey, E. Creel, R. Grossman, S. Gutti, and H. Sivakumar. A high performance implementation of the data space transfer protocol (DSTP). In Zaki and Ho [ZH99]. in conjunction with ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD99).

[BDKJT97] F. M. T. Brazier, B. M. Dunin-Keplicz, N. R. Jennings, and J. Treur. DESIRE: Modelling multi-agent systems in a compositional formal framework. *I*nt Journal of Cooperative Information Systems, 6(1):67–94, 1997.

[BDR98] D. Benech, T. Desprats, and Y. Raynaud. COBALT: An architecture for intelligent agent-based management. In *P*roceedings IFIP/IEEE NOMS'98,New Orleans, USA, February 1998.

[BIP88] M.E. Bratman, D.J. Israel, and M.E. Pollack. Plans and resource-bounded practical reasoning. *C*omputational Intelligence, 4:349–355, 1988.

[BKK97] Cliff Brunk, James Kelly, and Ron Kohavi. Mineset: An integrated system for data mining. In Heckerman et al. [HMPU97], page 135.

[Blu82] Robert L. Blum. *D*iscovery and Representation of Causal Relationships from a Large Time-Oriented Clinical Database: The RX Project, volume 19 of *L*ecture Notes in Medical Informatics. Spinger-Verlag, 1982.

[BN84] A. Birrel and B.J. Nelson. Implementing remote procedure calls. *A*CM Transactions on Computer Systems, 2(1):39–59, February 1984.

[Box97a] D. Box. Q&A ActiveX/COM. *M*icrosoft Systems Journal, pages 93–105, March 1997.

[Box97b] D. Box. Q&A ActiveX/COM. *M*icrosoft Systems Journal, pages 93–108, July 1997.

[Bra97a] Jeffrey M. Bradshaw, editor. *S*oftware Agents. AAAI Press/MIT Press, 1997.

[Bra97b] Jeffrey M. Bradshaw. *S*oftware Agents, chapter Introduction to Software Agents, pages 3–48. AAAI Press/MIT Press, 1997. Chapter belonging to [Bra97a].

[Bro86] R.A. Brooks. A robust layered control system for a movile robot. *IEEE* Journal of Robotics and Automation, 2(1):14–23, 1986.

[BS92] B. Burmesiter and K. Sundermeyer. *D*ecentralized AI 3 – Proceedings of the Third European Workshop on Modelling Autonomous Agents and Multi-Agent Worlds, chapter Cooperative Problem Solving Guided by Intention and Perception, pages 77–92. Elsevier, Amsterdam, The Nederlands, 1992.

[CA86] D. Chapman and P. Agre. *R*easoning About Actions and Plans - Proceedings of the 1986 Workshop, chapter Abstract Reasoning as Emergence from Concrete Ability, pages 411–424. Morgan Kaufmann, San Mateo, California, 1986.

[Cat94] R.G.G. Cattell, editor. *T*he Object Database Standard – ODMG93 v 1.2. Morgan Kaufmann, San Mateo, California, 1994.

[CCK+00] P. Chapman, J. Clinton, R. Kerber, T. Khabaza, T. Reinartz, C.Shearer, and R. Wirth. CRISP-DM 1.0 step by step data mining guide. CRISP Consortium Standard, August 2000.

[CGH+95] Davis Chess, Benjamin Grosof, Colin Harrison, David Levine, Colin Parris, and Gene Tsudik. Itinerant Agents for Mobile Computing. *IEEE* Computer Society Press, 2(5):34–49, October 1995.

[CGHH89] P.R. Cohen, M.L. Greenbergm, D.M. Hart, and A.E. Hoewe. Trial by fire: Understanding the design requirements for agents in complex environments. *A*I Magazine, 10(3):32–48, 1989.

[CGS99] J. Chattratichat, Y. Guo, and J. Syed. A visual language for internet-based data mining and data visualisation. In *I*EEE Symposium on Visual Languages (VL'99), Tokyo, Japan, 1999.

[CJ96] D. Cockburn and Nick R. Jennings. ARCHON: A distributed artificial intelligence system for industrial applications. In Greg O'Hare and Nick Jennings, editors, *F*oundations of Distributed Artificial Intelligence, chapter 12. John Wiley and Sons, 1996.

[CL90a] P.R. Cohen and H.J. Levesque. Intention is choice with commitment. *A*rtificial Intelligence, 42:213–261, 1990.

[CL90b] P.R. Cohen and H.J. Levesque. *I*ntentions in Communication, chapter Rational Interactions as the Basis of Communication, pages 221–256. AAAI Press/MIT Press, Cambridge, Massachusetts, 1990.

[CL97] P.R. Cohen and H.J. Levesque. Comunicative actions for artificial agents. In *F*irst International Conference on Multi-Agent Systems, pages 65–74, Cambridge, Massachusetts, 1997. AAAI Press.

[CNFF96] D. W. Cheung, V. T. Ng, A. W. Fu, and Y. J. Fu. Efficient mining of association rules in distributed databases. *I*eee Trans. On Knowledge And Data Engineering, 8:911–922, December 1996.

[COM95] COM. *T*he Component Object Model Specification. Microsoft, msdn library, specifications edition, 1995.

[CS93a] Philip K. Chan and Salvatore J. Stolfo. Experiments in multistrategy learning by meta-learning. In *P*roceedings of the second international conference on information and knowledge management, pages 314–323, Washington, DC, 1993.

[CS93b] D. E. Comer and D. L. Stevens. *I*nternetworking with TCP/IP, Volume III: Client-Server Programming and Applications, BSD Socket Version. Prentice Hall, Engelwood Cliffs, NJ, 1993.

[CS96] Philip K. Chan and Salvatore J. Stolfo. Sharing learned models among remote database partitions by local meta-learning. In Simoudis et al. [SHF96], page 2.

[Day99] Umeshwar Dayal. Large-scale data mining applications: Requirements and architectures. In Zaki and Ho [ZH99]. in conjunction with ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD99).

[Dia00] Daniel Diaz. *GNU* Prolog, edition 1.4 for gnu prolog version 1.2.0 edition, July 2000.

[DKR97] Mark Derthick, John Kolojejchick, and Steven F. Roth. An interactive visualization environment for data exploration. In Heckerman et al. [HMPU97], page 2.

[DM99] I. S. Dhillon and D. S. Modha. A data clustering algorithm on distributed memory machines a data clustering algorithm on distributed memory machines a data clustering algorithm on distributed memory machines. In Zaki and Ho [ZH99]. in conjunction with ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD99).

[DMTH95] G. DiMarzo, M. Muhugusa, C. Tschidin, and J. Harms. The messenger paradigm and its implications on distributed systems. In *P*roceedings ICC'95 Workshop on Intelligent Computer Communication, 1995.

[DP96] S. Dao and B. Perry. Information mediation in cyberspace: Scalable methods for declarative information networks. *J*ournal of Intellingent Information Systems, 6(2/3):131–150, 1996.

[EJ95] Stephen G. Eick and Brian S. Johnson. Interactive data visualization at AT&T bell labs. In *P*roceedings of ACM CHI'95 Conference on Human Factors in Computing Systems, volume 2 of *D*emonstrations: Visualization, pages 17–18, 1995.

[ELS94] O. Etzioni, N. Lesh, and R. Segal. Building softbots for UNIX. In *S*oftware Agents – Papers from 1994 Spring Symposiun, pages 9–16. AAAI Press, 1994.

[Far98] Jim Farley. *J*ava Distributed Computing. O'Reilly, 1998.

[Fer92] I.A: Ferguson. *TouringMachines*: An Architecture for Dynamic, Rational, Movile Agents. PhD thesis, Clare Hall, University of Cambridge, 1992.

[Fer95a] I.A: Ferguson. *I*ntelligent Agents: Theories, Architectures and Languages, volume 890 of *L*ectute Notes in Artificial Intelligence, chapter Integrated Control and Coordinated Behaviour A Case for Agent Models. Springer-Verlag, Heidelberg, Germany, January 1995.

[Fer95b] I.A: Ferguson. On the role of BDI modeling for integrated control and coordenated behaviour in autonomous agents. *A*pplied Artificial Intelligence, 9(4):421–448, November 1995.

[FFPR95] A. Farquhar, R. Fikes, W. Pratt, and J. Rice. Collaborative ontology construction for information integration. Technical Report KSL-95-63, Knowledge Systems Laboratory Department of Computer Science, 1995.

[FFR96] A. Farquhar, R. Fikes, and J. Rice. The ontolingua server: A tool for collaborative ontology construction. Technical Report KSL-96-26, Knowledge Systems Laboratory Department of Computer Science, 1996.

[FFR97] A. Farquhar, R. Fikes, and J. Rice. Tools for assembling modular ontologies in Ontolingua. Technical Report KSL-97-03, Knowledge Systems Laboratory Department of Computer Science, 1997.

[FG96] S. Franklin and A. Graesser. Is it an agent, or just a program?: A taxonomy for autonomous agents. In *P*roceedings of the Third International Workshop on Agent Theories, Architectures, and Languages. Springer-Verlag, 1996.

[Fis94] M. Fisher. *T*emporal Logic – Proceedings of the First International Conference, volume 827 of *L*NAI, chapter A Survey of Concurrent MetateM – The Language and its Applications, pages 480–505. Springer–Verlag, Heidelverg, Germany, 1994.

[FKZ97] Ronen Feldman, Willi Klösgen, and Amir Zilberstein. Visualization techniques to explore data mining results for document collections. In Heckerman et al. [HMPU97], page 16.

[FMP96] Klaus Fischer, Jörg P. Müller, and Markus Pischel. A pragmatic BDI architecture. In Michael Wooldridge, Jörg P. Müller, and Milind Tambe, editors, *P*roceedings on the IJCAI Workshop on Intelligent Agents II : Agent Theories, Architectures, and Languages, volume 1037 of *L*NAI, pages 203–218, Berlin, 19–20 1996. Springer Verlag.

[FMP+00] Covadonga Fernández, Ernestina Menasalvas, José M. Peña, Juan F. Martínez, Óscar Delgado, and J. Ignacio López. Rough sets as a foundation of data mining queries in DAMISYS. In *P*roceedings of IPMU'2000, 2000.

[FN71] R.E. Fikes and N.A. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *A*rtificial Intelligence, 5(2):189–208, 1971.

[FPM+99] Covadonga Fernández, José M. Peña, Juan F. Martínez, Óscar Delgado, J. Ignacio López, M. Ángeles Luna, and J.F. Borja Pardo. DAMISYS: An overview. In *P*roceedings of DAWAK'99, Florencia, Italy, pages 224–230, August 1999.

[FW93] M Fisher and M. Wooldridge. *S*pecifying and Verifying Distributed Intelligent Systems, volume 727 of *L*NAI, chapter Specifying and Verifying Distributed Intelligent Systems, pages 13–28. Springer-Verlag, Heidelberg, Germany, 1993.

[GBR+99] R. L. Grossman, S. Bailey, A. Ramu, B. Malhi, P. Hallstrom, I. Pulleyn, and X. Qin. The management and mining of multiple predictive models using the predictive modeling markup language (PMML). *I*nformation and Software Technology, 1999.

[GBST99] Robert L. Grossman, Stuart M. Bailey, Harinath Sivakumar, and Andrei L. Turinsky. Papyrus: A system for data mining over local and wide-area clusters and superclusters. In ACM, editor, *SC*'99: Oregon Convention Center 777 NE Martin Luther King Jr. Boulevard, Portland, Oregon, November 11–18, 1999, New York, NY 10036, USA and 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1999. ACM Press and IEEE Computer Society Press.

[GC99] S. Goil and A. Choudhary. Efficient parallel classification using dimensional aggregates. In Zaki and Ho [ZH99]. in conjunction with ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD99).

[ge92] G. Piatetsky-Shapiro (guest editor), editor. *IJIS* Special issue on Knowledge Discovery in Databases and Knowledge Bases, volume 7. September 1992. Special issue on Knowledge Discovery in Databases and Knowledge Bases, edited selection of best papers from AAAI KDD-91 workshop.

[GF92] M. R. Genesereth and R. E. Fikes. Knowledge Interchange Format, Version 3.0 Reference Manual. Technical Report Logic-92-1, Computer Science Department, Stanford University, June 1992.

[GG99] Robert Grossman and Yike Guo. Communicating data mining: Issues and challenges in wide area distributed data mining communicating data mining: Issues and challenges in wide area distributed data mining a data clustering algorithm on distributed memory machines. In Zaki and Ho [ZH99]. in conjunction with ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD99).

[GGH+95] M.R. Genesereth, D. Gunning, R. Hull, L. Kerschberg, R. King, B.Ñeches, and G. Wiederhold. I^3 reference architecture. Intelligent Integration of Information Program, Advanced Research Projects Agency, Draft, 1995.

[GH92] F.J. Garijo and D. Hoffmann. A multiagent architecture for operation and maintenance of telecomunication networks. In *P*roceedings 12th International Conference on Artificial Intelligence, Expert Systems and Natural Language (Avignon'92), Vol. 2, pages 427–436, 1992.

[GHJV93] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. A catalog of object-oriented design patterns, 1993. draft manuscript.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *D*esign Patterns. Addison Wesley, Reading, MA, 1995.

[Gin91] Matthew L. Ginsberg. Knowledge Interchange Format: The KIF of Death. *A*I Magazine, 12(2):57–63, 1991.

[GK94] M.R. Genesereth and S.P. Ketchpel. Software agents. *C*ommunications of the ACM, 37(7):48–53, 94.

[GKM+98] R. L. Grossman, S. Kasif, D. Mon, A. Ramu, and B. Malhi. The preliminary design of Papyrus: A system for high performance, distributed data mining over clusters, meta-clusters and super-clusters. In *P*roceedings of the KDD-98 Workshop on Distributed Data Mining. AAAI Press, 1998.

[GKM+99] R. Grossman, S. Kasif, R. Moore, D. Rocke, and J. Ullman. Data mining research: Opportunities and challenges, January 1999. Report of three NSF Workshops on Mining Large, Massive and Distributed Data.

[GL87] M.P. Georgeff and A.L. Lansky. Reactive reasoning and planing. In *P*roceedings Sixth National Conference on Artificial Intelligence (AAAI-87), pages 677–682, 1987.

[GL94] R.V. Guha and D.B. Lenat. Enabling agents to work together. *C*ommunications of the ACM, 37(7):127–142, 1994.

[GM95] James Gosling and H. McGilton. *T*he Java Language Environment. Sun Microsystems Computer Company, May 1995.

[GMI95a] GMI. *T*he TDE Developer Libraries Reference. General Magic Inc., telescript development environment edition, October 1995.

[GMI95b] GMI. *T*elescript Language Reference. General Magic Inc., telescript development environment edition, October 1995.

[GMI96a] GMI. *T*he TDE User Guide. General Magic Inc., telescript development environment edition, January 1996.

[GMI96b] GMI. *T*elescript Programming Guide. General Magic Inc., telescript development environment edition, January 1996.

[GPP+99] M. Georgeff, B. Pell, M. Pollack, M. Tambe, and M. Wooldridge. The Belief-Desire-Intention model of agency. In *P*roceedings of Agents, Theories, Architectures and Languages (ATAL), 1999.

[GQ94] R. L. Grossman and X. Qin. Ptool: A scalable persistent object manager. *S*IGMOD Record (ACM Special Interest Group on Management of Data), 23(2):510–510, June 1994.

[Gra95] R. S. Gray. Agent Tcl: A transportable agent system. In *P*roceedings of the CIKM Workshop on Intelligent Information Agents, Fourth International Conference on Information and Knowledge Management (CIKM 95), Baltimore, Maryland, December 1995.

[Gra97] R. S. Gray. *Agent Tcl*: A Flexible and Secure Mobile-Agent System. PhD thesis, Dept. of Computer Science, Dartmouth College, June 1997. Available as Dartmouth Computer Science Technical Report TR98-327.

[Gri96] Udo Grimmer. Clementine: Data mining software. In Hans-Joachim Mucha and Hans-Hermann Bock, editors, *C*lassification and Multivariate Graphics: Models, Software and Applications, number 10 in Weierstrass-Institut für Angewandte Analysis und Stochastik, pages 25–31. Berlin, Germany, 1996.

[Gro96] Robert Grossman. The terabyte challenge: An open, distributed testbed for managing and mining massive data sets. In *C*D-ROM Proceedings of Supercomputing'96, Pittsburgh, PA, November 1996. IEEE.

[Gru93] T.R. Gruber. A translation approach to portable ontology specification. *K*nowledge Acquisition, 5(2):199–220, 1993.

[GS99] Y. Guo and J. Sutiwaraphun. Probing knowledge in distributed data mining. In *P*roceedings PAKDD, Beijing, China, 1999.

[GSSW92] G. Grinstein, J. C. Sieg, S. Smith, and M. G. Williams. Visualization for knowledge discovery. In ijis-special-issue:92 [ge92], pages 637–648. Special issue on Knowledge Discovery in Databases and Knowledge Bases, edited selection of best papers from AAAI KDD-91 workshop.

[Had94] A. Haddadi. A hybrid architecture for multi-agent systems. In *P*roceedings of the 1986 Conference of Theoretical Aspects of Reasoning About Knowledge, pages 1–18, San Mateo, California, 1994. Morgan Kaufmann.

[HBC+99] M. Hermenegildo, F. Bueno, D. Cabeza, M. Carro, M. García de-la Banda, P. López-García, and G. Puebla. *P*arallelism and Implementation of Logic and Constraint Logic Programming, chapter The CIAO Multi-Dialect Compiler and System: An Experimentation Workbench for Future (C)LP Systems, pages 65–85. Nova Science, April 1999.

[HCBK99] L.O. Hall, N. Chawla, K.W. Bowyer, and W. P. Kegelmeyer. Learning rules from distributed data. In Zaki and Ho [ZH99]. in conjunction with ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD99).

[Hew76] Carl Hewitt. Viewing control structures as patterns of passing messages. Memo 410, MIT Artificial Intelligence Laboratory, December 1976.

[Hew77] Carl Hewitt. Viewing control structures as patterns of passing messages. *A*rtificial Intelligence, 8(3):323–364, Fall 1977.

[HFW+96a] Jiawei Han, Yongjian Fu, Wei Wang, Jenny Chiang, Wan Gong, Krzystof Koperski, Deyi Li, Yijun Lu, Amynmohamed Rajan, Nebojsa Stefanovic, Betty Xia, and Osmar R. Zaiane. DBMiner: A system for mining knowledge in large relational databases. In Simoudis et al. [SHF96], page 250.

[HFW+96b] Jiawei Han, Yongjian Fu, Wei Wang, Jenny Chiang, Osmar R. Zaïane, and Krzysztof Koperski. DBMiner: Interactive mining of multiple-level knowledge in relational databases. In H. V. Jagadish and Inderpal Singh Mumick, editors, *P*roceedings of the 1996 ACM SIGMOD International Conference on Management of Data, page 550, Montreal, Quebec, Canada, 4–6 June 1996.

[HK97] Markus Horstmann and Mary Kirtland. DCOM architecture. Whitepaper, Microsoft Corporation, July 1997.

[HKK97] E.H. Han, G. Karypis, and V. Kumar. Scalable parallel data mining for association rules. In *P*roceedings ACM SIGMOD Conference for Management of Data, May 1997.

[HMPU97] David Heckerman, Heikki Mannila, Daryl Pregibon, and Ramasamy Uthurusamy, editors. *P*roceedings of the Third International Conference on Knowledge Discovery and Data Mining (KDD-97). AAAI Press, 1997.

[Hub95] J. Huber. PPFS: An experimental filesystem for high performance parallel input/output. Master's thesis, Department of Computer Science, University of Illinois at Urbana-Champaing, 1995.

[Hut93] James M. Hutchinson. A radial basis function approach to financial time series analysis. Technical report, Artificial Intelligence Laboratory, Massachusetts Institute of Technology (MIT), Cambridge, Massachusetts, December 1993.

[IBM99] IBM. *A*pplication programming interface and utility reference. IBM DB2 Intelligent Miner for Data. IBM, September 1999.

[ISL95] ISL. *C*lementine User Guide, volume Version 5. ISL, Integral Solutions Limited, July 1995.

[ISL99] ISL. Clementine servcer distributed architecture. White Paper, 1999. Integrates Solution Limited, SPSS Group.

[ISO95] ISO. Information technology – programming languages – Prolog – part 1: General core. Standard ISO/IEC 13211–1, 1995.

[JCL+95] N. R. Jennings, J. M. Corera, L. Laresgoiti, E. H. Mamdani, F. Perriollat, P. Skarek, and L. Z. Varga. Using ARCHON to develop real-world DAI applications for electricity transportation management and particle accelerator control. *IEEE* Expert, 1995.

[Jen93] Nicholas R. Jennings. Specification and implementation of a belief desire joint–intention architecture for collaborative problem solving. *J*ournal of Intelligent and Cooperative Information Systems, 2(3):289–318, 1993.

[JHKK00] Mahesh V. Joshi, Eui-Hong (Sam) Han, George Karypis, and Vipin Kumar. chapter Parallel Algorithms for Data Mining. Morgan Kaufmann, 2000.

[JKK98] M. Joshi, G. Karypis, and V. Kumar. Scalparc: A scalable and parallel classification algorithm for mining large datasets. In *P*roceedings International Parallel Processing Symposium, 1998.

[JvRS95a] Dag Johansen, Robbert van Renesse, and Fred B. Schneider. An Introduction to the TACOMA Distributed System Version 1.0. Technical Report 95–23, University of Tromsø, Norway, June 1995.

[JvRS95b] Dag Johansen, Robbert van Renesse, and Fred B. Schneider. Operating System Support for Mobile Agents. In *P*roceedings of the Fifth Workshop Hot Topics in Operating Systems (HotOS), pages 42–45, Washington, USA, May 1995.

[JvRS96] Dag Johansen, Robbert van Renesse, and Fred B. Schneider. Supporting Broad Internet Access to TACOMA. In *P*roceedings of the 7th SIGOPS European Workshop, pages 55–58, Connemara, Ireland, September 1996.

[JW98a] Nicholas R. Jennings and Michael J. Wooldridge, editors. *A*gent Technology: Foundations, Applications and Markets. UNICOM-Seminars. Spinger-Verlag, 1998.

[JW98b] Nicholas .R. Jennings and Michael J. Wooldridge. *A*pplication of Intelligent Agents, chapter Application of Intelligent Agents. UNICOM-Seminars. Spinger-Verlag, 1998. Chapter belonging to [JW98a].

[Kae86] L.P. Kaelbling. *R*easoning About Actions and Plans - Proceedings of the 1986 Workshop, chapter An Architecture for Intelligent Reactive Agents, pages 395–410. Morgan Kaufmann, San Mateo, California, 1986.

[Kae91] L.P. Kaelbling. A situated automata approach to the design of embedded agents a situated automata approach to the design of embedded agentsdesigning autonomous agents of the 1986 workshop. *S*IGART Bulletin, 2(4):85–88, 1991.

[Ken99a] Kensingston, Enterprise Data Mining. Kensington: New generation enterprise data mining. White Paper, 1999. Parallel Computing Research Centre, Department of Computing Imperial College, (Contact Martin Khler).

[Ken99b] Kensingston, Enterprise Data Mining. Kensington: The open infrastucture for enterprise data mining. White Paper, 1999. Parallel Computing Research Centre, Department of Computing Imperial College, (Contact Martin Khler).

[KHS97a] Hillol Kargupta, Ilker Hamzaoglu, and Brian Stafford. Scalable, distributed data mining-an agent architecture. In Heckerman et al. [HMPU97], page 211.

[KHS97b] Hillol Kargupta, Ilker Hamzaoglu, and Brian Stafford. Web based parallel/distributed medical data mining using software agents. In *E*xtended Proceedings of the 1997 Fall Symposium, American Medical Informatics Association, 1997.

[KJL+94] R. Kohavi, G. John, R. Long, D. Manley, and K. Pfleger. MLC++: A machine learning library in C++. In *T*ools with Artificial Intelligence, pages 740–743, 1994. Available by anonymous ftp from: starry.stanford.edu:pub/ronnyk/mlc/toolsmlc.ps.

[Koh91] J.T. Kohl. The evolution of the Kerberos authentication service. In *P*roceedings EurOpen Spring'91, pages 295–313, 1991.

[Koh95] Ronny Kohavi. MLC++ tutorial a machine learning library of c++ classes, November 1995.

[Koh97] Ron Kohavi. Data mining with mineset: What worked, what did not, and what might. In *P*roceedings of the Workshop on Knowledge Discovery in Databases, 1997. Workshop on the Commercial Success of Data Mining.

[KR91] L.P. Kaelbling and S.J. Rosenchein. *D*esigning Autonomous Agents Designing Autonomous Agents of the 1986 Workshop, chapter Action and Planning in Embedded Agents, pages 35–48. AAAI Press/MIT Press, Cambridge, Massachusetts, 1991.

[Kri95] Cheryl D. Krivda. Data-mining dynamite — supercharge your data-mining projects with data cleansing, data warehouses, parallel processing, and mega-storage. *B*yte Magazine, 20(10):97–??, October 1995.

[KS95] Tom Khabaza and Colin Shearer. Data mining with clementine, February 1995.

[KS96] Ron Kohavi and Dan Sommerfield. SGIMLC++ utilities 2.0, 1996.

[KSD96] Ron Kohavi, Dan Sommerfield, and James Dougherty. Data mining using MLC++: A machine learning library in C++. In *T*ools with Artificial Intelligence, pages 234–245. IEEE Computer Society Press, 1996. Received the best paper award.

[Lan97] Danny B. Lange. Java Aglet Application Programming Interface (J-AAPI) White Paper – Draft 2. White Paper IBM Tokyo Research Laboratory, February 1997.

[LC96] Danny B. Lange and Daniel T. Chang. Programming Mobile Agents in Java – A White Paper. White paper, IBM Corp., 1996.

[Len95] D.B. Lenat. Cyc: A large-scale investment in knowledge infrastructure. *C*ommunications of the ACM, 38(11):33–38, 1995.

[LF97] Yannis Labrou and Tim Finin. A proposal for a new KQML specification. Technical Report TR CS–97–03, Computer Science and Electrical Engineering Department, University of Maryland Baltimore County, February 1997.

[LOKK97] Danny B. Lange, Mitsuru Oshima, Gunter Karjoth, and Kosaka Kazuya. Aglets: Programming Mobile Agents in Java. In T. Masuda, Y. Masunaga, and M. Tsukamoto, editors, *P*roceedings of Worldwide Computing and Its Applications (WWCA'97), Lecture Notes in Computer Science, volume 1274, pages 253–266, Tsukuba, Japan, March 1997.

[Lon94] R. Long. Overview of MLC++ library classes, 1994. Internal document.

[Lóp99] J. Ignacio López. Diseño e implementación del nivel de data warehouse para RSDM. Master's thesis, DLSIIS, FI, Universidad Politécnica de Madrid, Spain, Julio 1999.

[LT87a] Nancy A. Lynch and Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, April 1987. Published also as technical report [LT87c].

[LT87b] Nancy A. Lynch and Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *P*roceedings of the Sixth Annual ACM SYmposium on Principles of Distributed Computing, pages 137–151, August 1987. Abbreviated version of [LT87c].

[LT87c] Nancy A. Lynch and Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. Technical Report MIT/LCS/TR–387, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, April 1987.

[LT89] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI*-Quaterly, (2):219–246, September 1989.

[Lyn97] Nancy A. Lynch. *D*istributed Algorithms. Series in Data Management Systems. Morgan Kaufmann, 1997.

[Mac91] R. MacGregor. *P*rinciples of Semantic Networks: Explorations in the Representation of Knowledge, chapter The Envolving Technology of Classification-Base Knowledge Representation Systems. Morgan Kaufmann, San Mateo, California, 1991.

[Mae91a] Pattie Maes. The agent network architecture. In *P*roceedings of AAAI Spring Symposium on Integrated Intelligent Architectures, Stanford, California, 1991. AAAI Press.

[Mae91b] Pattie Maes, editor. *D*esigning Autonomous Agents: Theory and Practice from Biology to Engineering and Back. AAAI Press/MIT Press, 1991.

[Mal97] B. S. Malhi. Data Mining Query Language : A perspective. Technical report, Data Mining Group at Laboratory of Advance Computing, 1997.

[Man97] H. Mannila. Methods and problems in data mining. In *P*roceedings International Conference of Data Base Theory, 1997.

[Mar99] Juan F. Martínez. Diseño e implementación del subsistema motor de DAMISYS. Master's thesis, DLSIIS, FI, Universidad Politécnica de Madrid, Spain, Septiembre 1999.

[Men98] Ernestina Menasalvas. *I*ntegración del Proceso de Inferencia de Conocimiento con las Bases de Datos Relacionales: Formalización Matemática de la Fase de Análisis. PhD thesis, DLSIIS, FI, Universidad Politécnica de Madrid, Spain, Septiembre 1998.

[MH99] Richard Monson-Haefel. *E*nterprise JavaBeans. O'Reilly, 1999.

[Mic98] Microsoft Corporation. Micorsoft component services. Whitepaper, Microsoft Corporation, August 1998.

[Mic99a] Microsoft Corporation. Active directory service technical overview. Whitepaper, Microsoft Corporation, June 1999.

[Mic99b] Microsoft Corporation. Internet information server technical overview. Whitepaper, Microsoft Corporation, September 1999.

[Mic00a] Microsoft Corporation. *B*uilding Enterprise Active Directory Services: Notes from the Field. Microsoft Press, January 2000.

[Mic00b] Microsoft Corporation. Message queue server reviewer's guide. Whitepaper, Microsoft Corporation, 2000.

[Mic00c] Microsoft Corporation. Microsoft transaction server-transactional component services. Whitepaper, Microsoft Corporation, 2000.

[Mil85] D. L. Mills. RFC 958: Network time protocol (NTP), September 1985. Obsoleted by RFC1059, RFC1119, RFC1305 [Mil88, Mil89b, Mil92]. Status: UNKNOWN.

[Mil88] D. L. Mills. RFC 1059: Network time protocol (version 1) specification and implementation, July 1988. Obsoleted by RFC1119, RFC1305 [Mil89b, Mil92]. Obsoletes RFC0958 [Mil85]. Status: UNKNOWN.

[Mil89a] D. Mills. STD 12: Network Time Protocol, September 1989. See also RFC1119 [Mil89b].

[Mil89b] D. L. Mills. RFC 1119: Network time protocol (version 2) specification and implementation, September 1989. Obsoleted by RFC1305 [Mil92]. Obsoletes RFC0958, RFC1059 [Mil85, Mil88]. See also STD0012 [Mil89a]. Status: STANDARD.

[Mil92] David L. Mills. RFC 1305: Network time protocol (version 3) specification, implementation, March 1992. Obsoletes RFC0958, RFC1059, RFC1119 [Mil85, Mil88, Mil89b]. Status: DRAFT STANDARD.

[Mil95] G.A. Miller. WorldNet: A lexical database for english. *C*ommunications of the ACM, 38(11):39–41, 1995.

[ML94] A.E. Milewski and S.M. Lewis. Design of intelligent agent user interfaces: Delegation issues. Technical report, AT&T Information Technologies Services, October 1994.

[MLF96] J. Mayfield, Y. Labrou, and T. Finin. *I*ntelligent Agents Volume II – Proceedings of the 1995 Workshop on Agent Theories, Architectures, and Languages, chapter Evaluating KQML as an Agent Communication Language, pages 347–360. Lecture Notes in Artificial Intelligence. Springer-Verlag, 1996.

[MN95] K. Mehlhorn and S.D. Nher. LEDA, a platform for combinatorial and geometric computing. In *C*ACKM, volume 38, pages 96–102, 1995.

[MN99a] K. Mehlhorn and S.D. Nher. *T*he LEDA Platform of Combinatorial and Geometric Computing. Cambridge University Press, 1999.

[MN99b] S. Morishita and A.Ñakaya. Parallel branch-and-bound graph search for correlated association rules. In Zaki and Ho [ZH99]. in conjunction with ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD99).

[MNSU96] K. Mehlhorn, S.D. Nher, M. Seel, and C. Uhrig. LEDA user manual, version 3.7.1. Internal Document, 1996.

[Moo90] R.C. Moore. *R*eadings in Planning, chapter A Formal Theory of Knowledge an Action, pages 480–519. Morgan Kaufmann, San Mateo, California, 1990.

[Moo99] Reagan Moore. Collection-based data management. In Zaki and Ho [ZH99]. in conjunction with ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD99).

[MP93] Jörg P. Müller and Markus Pischel. The agent architecture inteRRaP: Concept and application. Research Report RR-93-26, Deutsches Forschungszentrum für Künstliche Intelligenz, Deutsches Forschungszentrum für Künstliche Intelligenz GmbH
Erwin-Schrödinger Strasse
Postfach 2080
67608 Kaiserslautern
Germany, 1993.

[MPT95] Jörg P. Müller, Markus Pischel, and Michael Thiel. Modelling reactive behaviour in vertically layered agent architectures. In Michael J. Wooldridge and Nicholas R. Jennings, editors, *P*roceedings of the ECAI-94 Workshop on Agent Theories, architectures and languages: Intelligent Agents I, volume 890 of *L*NAI, pages 261–276. Springer-Verlag: Heidelberg, Germany, August 1995.

[MR92] J. A. Major and D. R. Riedinger. Efd - a hybrid knowledge statistical-based system for the detection of fraud. In ijis-special-issue:92 [ge92], pages 687–703. Special issue on Knowledge Discovery in Databases and Knowledge Bases, edited selection of best papers from AAAI KDD-91 workshop.

[MTF98] M. Mulder, J. Treur, and M. Fisher. Agent modelling in MetateM and DESIRE. *L*ecture Notes in Computer Science, 1365:193–??, 1998.

[Mül96] Jörg P. Müller. *A*n Architecture for Dynamically Interacting Agents. PhD thesis, Universität des Saarlandes, Saarbrucken, 1996.

[MZ00] William A. Maniatty and Mohammed J. Zaki. A requirements analysis for parallel kdd systems. In Jose Rolim et al., editor, *3*rd IPDPS Workshop on High Performance Data Mining, pages 358–265, May 2000.

[Nec94] R.D. Neches. Overview of the ARPA knowledge sharing effort. Edited by Gruber, April 1994.

[NN98] H.S. Nwana and D.T. Ndumu. *A* Brief Introduction to Software Agent Technology, chapter Application of Intelligent Agents. UNICOM-Seminars. Spinger-Verlag, 1998. Chapter belonging to [JW98a].

[NS76] A.J. Newell and H.A. Simon. Computer science as empirical enquiry. *C*ommunications of the ACM, 19:113–126, 1976.

[Nwa96] H.S. Nwana. Software agents: An overview. *T*he Knowledge Engineering Review, 11(3), 1996.

[OMG95] Object Management Group. Compound presentation and compound interchange facilities – part. Document: 95–12–30, December 1995.

[OMG97a] Object Management Group. CORBA 2.0 IIOP specification. Document: formal/97–02–25, February 1997.

[OMG97b] Object Management Group. CORBA 2.1 without change bars. Document: formal/97–11–03, November 1997.

[OMG97c] Object Management Group. CORBA/IIOP 2.1 - chapter 12 (GIOP) (w/changebars). Document: formal/97–10–17, October 1997.

[OMG97d] Object Management Group. CORBA/IIOP 2.1 - chapter 8 (boa) (w/changebars). Document: formal/97–10–13, October 1997.

[OMG97e] Object Management Group. CORBA/IIOP 2.1 specification - full version. Document: formal/97–09–01, September 1997.

[OMG97f] Object Management Group. CORBAservices - collection service (chapter 17). Document: formal/97–12–24, December 1997.

[OMG97g] Object Management Group. CORBAservices - concurrency service (chapter 7). Document: formal/97–12–14, December 1997.

[OMG97h] Object Management Group. CORBAservices - event management service (chapter 4). Document: formal/97–12–11, December 1997.

[OMG97i] Object Management Group. CORBAservices - licensing service (chapter 12). Document: formal/97–12–19, December 1997.

[OMG97j] Object Management Group. CORBAservices - lifecycle service (chapter 6). Document: formal/97–12–13, December 1997.

[OMG97k] Object Management Group. CORBAservices - persistent object service (chapter 5). Document: formal/97–12–12, December 1997.

[OMG97l] Object Management Group. CORBAservices - property service (chapter 13). Document: formal/97–12–20, December 1997.

[OMG97m] Object Management Group. CORBAservices - query service (chapter 11). Document: formal/97–12–18, December 1997.

[OMG97n] Object Management Group. CORBAservices - relationships service (chapter 9). Document: formal/97–12–16, December 1997.

[OMG97o] Object Management Group. CORBAservices - time service (chapter 14). Document: formal/97–12–21, December 1997.

[OMG97p] Object Management Group. CORBAservices - trader service (chapter 16). Document: formal/97–12–23, December 1997.

[OMG97q] Object Management Group. CORBAservices - transactions service (chapter 10). Document: formal/97–12–17, December 1997.

[OMG97r] Object Management Group. CORBAservices specification - full book - updated december 1997. Document: formal/97–12–02, December 1997.

[OMG97s] Object Management Group. DCE/CORBA interworking RFP, draft 2 and final version. Document: orbos/97–02–03, February 1997.

[OMG97t] Object Management Group. Overview of CORBA geographical information systems (CORBAgis). Document: c4i/97–06–06, June 1997.

[OMG98a] Object Management Group. Complete CORBAservices book. Document: formal/98–07–05, July 1998.

[OMG98b] Object Management Group. Component framework for telecoms. Document: telecom/98–06–12, June 1998.

[OMG98c] Object Management Group. Components white paper. Document: omg/98–09–01, September 1998.

[OMG98d] Object Management Group. CORBA 2.2 specification. Document: formal/98–07–01, July 1998.

[OMG98e] Object Management Group. CORBA 2.3 full specification. Document: formal/98–12–01, December 1998.

[OMG98f] Object Management Group. CORBA telecoms book. Document: formal/98–06–01, June 1998.

[OMG98g] Object Management Group. CORBA telecoms specification. Document: formal/98–07–12, July 1998.

[OMG98h] Object Management Group. CORBA/IIOP 2.2 specification - full version. Document: formal/98–02–01, February 1998.

[OMG98i] Object Management Group. Corbamed roadmap version 1.0. Document: corbamed/98–12–02, December 1998.

[OMG98j] Object Management Group. CORBAmed roadmap version 2.0. Document: corbamed/98–10–01, October 1998.

[OMG98k] Object Management Group. CORBAservices externalization. Document: formal/98–12–16, December 1998.

[OMG98l] Object Management Group. Coseventchanneladmin. Document: formal/98–10–05, October 1998.

[OMG98m] Object Management Group. Cosnaming. Document: formal/98–10–19, October 1998.

[OMG98n] Object Management Group. Interoperable naming service IDL. Document: orbos/98–03–07, March 1998.

[OMG98o] Object Management Group. Security service v1.2 OMG IDL. Document: formal/98–12–20, December 1998.

[OMG99a] Object Management Group. Ada language mapping OMG IDL text file. Document: formal/99–08–02, August 1999.

[OMG99b] Object Management Group. C language mapping OMG IDL text file. Document: formal/99–08–03, August 1999.

[OMG99c] Object Management Group. C++ language mapping OMG IDL text file. Document: formal/99–08–04, August 1999.

[OMG99d] Object Management Group. Cobol language mapping OMG IDL text file. Document: formal/99–08–05, August 1999.

[OMG99e] Object Management Group. CORBA 2.3 - chapter 10 - interface repository. Document: formal/99–07–14, July 1999.

[OMG99f] Object Management Group. CORBA 2.3 - chapter 11 - portable object adaptor. Document: formal/99–07–15, July 1999.

[OMG99g] Object Management Group. CORBA 2.3 - chapter 3 - IDL syntax and semantics. Document: formal/99–07–07, July 1999.

[OMG99h] Object Management Group. CORBA 2.3 chapter 1 - object model. Document: formal/99–07–05, July 1999.

[OMG99i] Object Management Group. CORBA component specification. Document: orbos/99–02–01, February 1999.

[OMG99j] Object Management Group. CORBA electronic commerce domain specification v1.0. Document: dtc/99–07–03, July 1999.

[OMG99k] Object Management Group. CORBAmed roadmap. Document: corbamed/99–01–13, January 1999.

[OMG99l] Object Management Group. The CORBAmed roadmap v1.0b, 3rd february 1999. Document: corbamed/99–11–01, November 1999.

[OMG99m] Object Management Group. CORBAmed roadmap, version 1.0b...latest draft. Document: corbamed/99–02–01, February 1999.

[OMG99n] Object Management Group. IDL to java language mapping OMG IDL text file. Document: formal/99–08–06, August 1999.

[OMG99o] Object Management Group. IDL to PL/I language mapping document. Document: orbos/99–11–01, November 1999.

[OMG99p] Object Management Group. Java to IDL language mapping OMG IDL text file. Document: formal/99–08–07, August 1999.

[OMG99q] Object Management Group. Smalltalk language mapping OMG IDL text file. Document: formal/99–08–08, August 1999.

[OMG00a] Object Management Group. Collection service stand-alone document. Document: formal/00–06–13, June 2000.

[OMG00b] Object Management Group. Concurrency service stand-alone document. Document: formal/00–06–14, June 2000.

[OMG00c] Object Management Group. Discussion of the object management architecture (OMA) guide. Document: formal/00–06–41, June 2000.

[OMG00d] Object Management Group. Event service stand-alone document. Document: formal/00–06–15, June 2000.

[OMG00e] Object Management Group. Externalization service stand-alone document. Document: formal/00–06–16, June 2000.

[OMG00f] Object Management Group. Licensing service stand-alone document. Document: formal/00–06–17, June 2000.

[OMG00g] Object Management Group. Life cycle service stand-alone document. Document: formal/00–06–18, June 2000.

[OMG00h] Object Management Group. Lisp mapping specification. Document: formal/00–06–02, June 2000.

[OMG00i] Object Management Group. Naming service stand-alone document. Document: formal/00–06–19, June 2000.

[OMG00j] Object Management Group. Notification service stand-alone document. Document: formal/00–06–20, June 2000.

[OMG00k] Object Management Group. Persistent object service stand-alone document. Document: formal/00–06–21, June 2000.

[OMG00l] Object Management Group. Property service stand-alone document. Document: formal/00–06–22, June 2000.

[OMG00m] Object Management Group. Query service stand-alone document. Document: formal/00–06–23, June 2000.

[OMG00n] Object Management Group. Relationship service stand-alone document. Document: formal/00–06–24, June 2000.

[OMG00o] Object Management Group. Security service v1.5 stand-alone document. Document: formal/00–06–25, June 2000.

[OMG00p] Object Management Group. Time service stand-alone document. Document: formal/00–06–26, June 2000.

[OMG00q] Object Management Group. Trading object service stand-alone document. Document: formal/00–06–27, June 2000.

[OMG00r] Object Management Group. Transaction service (v1.1) stand-alone document. Document: formal/00–06–28, June 2000.

[OSF94] Open Software Foundation. OSF DCE application development guide, 1994. Distributed Computer Environment document, Revision 1.0, update 1.0.2.

[OSF95] Open Software Foundation. AES/distributed computing – remote procedure call, 1995. Distributed Computer Environment document, Revision B.

[Par98] J.F. Borja Pardo. Diseño del catálogo del data warehouse para soportar las operaciones de RSDM. Master's thesis, DLSIIS, FI, Universidad Politécnica de Madrid, Spain, Septiembre 1998.

[Paw82] Z. Pawlak. Rough sets. *I*nternational Journal of Computer and Information Sciences, 11(5), 1982.

[PCS00] A. Prodromidis, P. Chan, and S. Stolfo. chapter Meta-learning in distributed data mining systems: Issues and approaches. AAAI/MIT Press, 2000.

[PDO99] S. Parthasarathy, S. Dwarkadas, and M. Ogihara. Active data mining in a distributed setting. In Zaki and Ho [ZH99]. in conjunction with ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD99).

[Peñ00] José María Peña. Lenguaje de definición de componentes MOIRAE. Technical Report (Data Mining Group) RSDM-22, DLSIIS, FI, UPM, January 2000.

[PFL00] J.M. Peña, F. Famili, and S. Létourneau. Data mining to detect abnormal behavior in aerospace data component failure. In *P*roceedings of the ACM-KDD 2000, Boston, 2000.

[PFPS+92] R. S. Patil, R. E. Fikes, P. F. Patel-Schneider, D. McKay, T. Finin, T. Gruber, and R.D. Neches. The DARPA knowledge sharing effort: Progress report. In C. Rich, W. Swartout, and B.D. Nebel, editors, *P*roceedings of Knowledge Representation and Reasoning (KR&R-92), pages 777–788, 1992.

[PLF99] J.M. Peña, S. Létourneau, and F. Famili. Application of rough sets algorithms to prediction of aircraft component failure. In *P*roceedings of the Third International Symposium on Intelligent Data Analysis. Springer-Verlag, 1999.

[PR90] M.E. Pollack and M. Riguette. Introducing Tileworld: Experimentally evaluating agent architectures. In *P*roceedings of the Eighth National Conference on Artifical Intelligene (AAAI–90), pages 183–189, 1990.

[PS91a] G. Piatetsky-Shapiro. *K*nowledge Discovery in Databases. AAAI Press, 1991.

[PS91b] G. Piatetsky-Shapiro. Report on the AAAI-91 workshop on knowledge discovery in databases. *I*EEE Expert, 6(5):74–76, 1991.

[PSBK+96] Gregory Piatetsky-Shapiro, Ron Brachman, Tom Khabaza, Willi Kloesgen, and Evangelos Simoudis. An overview of issues in developing industrial data mining and knowledge discovery applications. In Simoudis et al. [SHF96], page 89.

[Qui86] J. Ross Quinlan. Induction of decision trees. *M*achine Learning, 1:81–106, 1986.

[Qui92] J. Ross Quinlan. *C4*.5: Programs for Machine Learning. Morgan Kaufmann, 1992.

[Ram98] A. T. Ramu. Incorporating transportable software agents into a wide area high performance distributed data mining system. Master's thesis, University of Illinois at Chicago, 1998.

[Ree95] S. Reese Hedberg. Parallelism speeds data mining. *I*EEE parallel and distributed technology: systems and applications, 3(4):3–6, Winter 1995.

[RG91a] A.S. Rao and M.P. Georgeff. Asymetric thesis and side-effect problems in linear time and branching time intention logics. In *P*roceedings of the Eighth National Conference on Artificial Intelligence (AAAI-90), pages 183–189, 1991.

[RG91b] A.S. Rao and M.P. Georgeff. Modeling rational agents within a BDI-architecture. In R. Fikes and E. Sandewall, editors, *P*roceedings of Knowledge Representation and Reasoning, pages 473–484, San Mateo, California, 1991. Morgan Kaufmann.

[RG92a] A.S. Rao and M.P. Georgeff. An abstract architecture for rational agents. In C. Rich, W. Swartout, and B.D. Nebel, editors, *P*roceedings of Knowledge Representation and Reasoning, pages 439–449, San Mateo, California, 1992. Morgan Kaufmann.

[RG92b] A.S. Rao and M.P. Georgeff. Social plans: Preliminary report. In E. Werner and Y. Demazeau, editors, *P*roceedings of the Third European Workshop on Modelling Autonomous Agents and Multi-Agent Worlds, pages 57–76, Amsterdam, The Nederlands, 1992. Elsevier.

[RV98] D. Rathjens and H. Vanderberg. Mineset userś guide, 1998. Silicon Graphics, Inc., #007-3214-004.

[RW91] S.J. Russell and E. Wefald. *D*o the Right Thing – Studies in Limited Reasoning. AAAI Press/MIT Press, Cambridge, Massachusetts, 1991.

[SAM96a] J. Shafer, R. Agrawal, and M. Mehta. Sprint: A scalable parallel classifier for data mining. In *P*roceedings International Conference in Very Large Databases, March 1996.

[SAM96b] John C. Shafer, Rakesh Agrawal, and Manish Mehta. SPRINT: A scalable parallel classifier for data mining. In T. M. Vijayaraman, Alejandro P. Buchmann, C. Mohan, and Nandlal L. Sarda, editors, *P*roc. 22nd Int. Conf. Very Large Databases, VLDB, pages 544–555. Morgan Kaufmann, 3–6 September 1996.

[Sea69] J.R. Searle. *S*peech Acts. Harvard University Press, Cambridge, Massachusetts, 1969.

[SFL+00] S. Stolfo, W. Fan, W. Lee, A. Prodromidis, and P. Chan. Cost-based modeling for fraud and instrusion detection: Results from the JAM project,. In *DARPA* Information Survivability Conference and Exposition, pages 130–144. IEEE Computer Press, 2000.

[SG99] Joãn Pedro Sousa and David Garlan. Formal modeling of the enterprise JavaBeans component integration framework. In *P*roceedings of FM'99, LNCS 1709, Toulouse, France, September 1999. Springer Verlag.

[She96] Colin Shearer. User driven data mining, 1996. Unicom Data Mining Conference. London.

[SHF96] Evangelos Simoudis, Jia Wei Han, and Usama Fayyad, editors. *P*roceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD-96). AAAI Press, 1996.

[Sho90] Yoav Shoham. Agent-oriented programming. Technical report, Computer Science Department, Stanford University, Stanford, California, 1990.

[Sho93] Yoav Shoham. Agent-oriented programming. *A*rtificial Intelligence, 60(1):51, 1993.

[Sho97] Yoav Shoham. Agent-oriented programming. Technical report, Computer Science Department, Stanford University, Stanford, California, 1997.

[SIC00] SICS. *SICStus* Prolog. University of Marseilles, sicstus prolog 3.8.4 edition, 2000. May.

[Sin94] M.P. Singh. *M*ultiagent Systems: A Theoretical Framework for Intentions, Know-How and Communications, volume 799 of *L*ecture Notes in Computer Science (Lecture Notes in Artificial Intelligence). Springer-Verlag, Heidelberg, Germany, 1994.

[SK98] T. Shintani and M. Kitsuregawa. Mining algorithms for sequential patterns in parallel: Hash based approach. In *2*nd Pacific-Asian Conference on Knowledge Discovery and Data Mining, Appril 1998.

[SK99] T. Shintani and M. Kitsuregawa. Parallel algorithms for mining association rule mining on large scale PC cluster. In Zaki and Ho [ZH99]. in conjunction with ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD99).

[Ski98] D. Skillicorn. Strategies for parallelizing data mining. In *P*roceedings Workshop on High-Performance Data Mining, 1998. in association with IPPS/SPDP.

[Ski99] D. Skillicorn. Strategies for parallelizing data mining. *I*EEE Concurrency, 7, October–November 1999.

[SMHP+97a] Rational Software, Microsoft, Hewlett-Packard, Oracle, Sterling Software, MCI Systemhouse, Unisys, ICON Computing, IntelliCorp, i Logix, IBM, ObjecTime, Platinum Technology, Ptech, Taskon, Reich Technologies, and Softeam. *U*nified Modeling Language (version 1.1). Rational Software Corporation, September 1997.

[SMHP+97b] Rational Software, Microsoft, Hewlett-Packard, Oracle, Sterling Software, MCI Systemhouse, Unisys, ICON Computing, IntelliCorp, i Logix, IBM, ObjecTime, Platinum Technology, Ptech, Taskon, Reich Technologies, and Softeam. *U*nified Modeling Language - UML Semantics (version 1.1). Rational Software Corporation, September 1997.

[SNS88] J.G. Steiner, B.C. Neuman, and J.I. Schiller. Kerberos: An authentication service for open network systems. In *P*roceedings USENIX 1988, pages 191–202, 1988.

[Soh97] Yoav Soham. *S*oftware Agents, chapter An Overview of Agent-Oriented Programming, pages 271–290. AAAI Press/MIT Press, 1997. Chapter belonging to [Bra97a].

[Som96] F. Somers. HYBRID: Intelligent agents for distributed atm network management. In *P*roceedings ECAI Workshop on Intelligent Agennts for Telecom Applications, IATA'96, August 1996.

[SPT+97] Salvatore Stolfo, Andreas L. Prodromidis, Shelley Tselepis, Wenke Lee, Dave W. Fan, and Philip K. Chan. JAM: Java agents for meta-learning over distributed databases. In Heckerman et al. [HMPU97], page 74.

[SS94] R. Shortland and R. Scarfe. Data mining applications in bt. *B*t Technology J., 12:17–22, 1994.

[SS95] R.M. Soley and C.M. Stone. *T*he Object Management Architecture Guide. John Wiley & Sons, Inc., 3rd edition edition, June 1995.

[Ste90] L. Steels. *D*ecentralized AI – Proceedings of the First European Workshop on Modelling Autonomous Agents and Multi-Agent Worlds, chapter Cooperation Between Distributed Agents through Self Organization, pages 175–196. Elsevier, Amsterdam, The Nederlands, 1990.

[Sun98a] Sun Microsystems. Enterprise JavaBeans 1.1 specification. Whitepaper, Sun Microsystems, 1998.

[Sun98b] Sun Microsystems. Java Message Service (JMS). Whitepaper, Sun Microsystems, 1998.

[Sun98c] Sun Microsystems. RMI specification. Whitepaper, Sun Microsystems, 1998.

[Sun99a] Sun Microsystems. Enterprise JavaBeans 2.0 specification. Whitepaper, Sun Microsystems, 1999.

[Sun99b] Sun Microsystems. Java Naming and Directory Interface (JNDI). Whitepaper, Sun Microsystems, 1999.

[Sun99c] Sun Microsystems. Java RMI-IIOP documentation. Whitepaper, Sun Microsystems, 1999.

[Sun99d] Sun Microsystems. Java Transation Service (JTS). Whitepaper, Sun Microsystems, 1999.

[Sun99e] Sun Microsystems. Jini technology and emerging network technologies. Whitepaper, Sun Microsystems, January 1999.

[SZ96] K. Sycara. and D. Zeng. Coordination of multiple intelligent software agents. *I*nternational Journal of Cooperative Information Systems, 1996.

[Szo95] P. Szolovits. Uncertainty and decisions in medical informatics. *M*ethods Of Information In Medicine, 34:111–121, 1995.

[TDMH94] C. Tschidin, G. DiMarzo, M. Muhugusa, and J. Harms. Messenger-based operating systems. Technical report, University of Geneve, Switzerland, 1994.

[The97] The Data Mining Research Group. *D*BMiner User Manual. Simnon Fraser University, Intelligent Database Systems Laboratory, December 1997.

[Tho93] S.R. Thomas. *PLACA*, an Agent Oriented Programming Language. PhD thesis, Computer Science Department, Stanford University, Stanford, California, 1993.

[Tka98] D.S. Tkach. Information mining with the ibm intelligent miner family, February 1998. IBM Software Solutions White Paper.

[Tsc97] Christian Tschudin. The messenger environment M∅ – A condensed description. In *M*obile Object Systems: Towards the Programmable Internet, number 1222 in Lecture Notes in Computer Science, pages 149–156. Springer-Verlag, April 1997.

[TT96] Shusaku Tsumoto and Hiroshi Tanaka. Automated discovery of medical expert system rules from clinical databases based on rough sets. In Simoudis et al. [SHF96], page 63.

[VB90] S. Vere and T. Bickmore. A basic agent. *C*omputational Intelligence, 6:391–395, 1990.

[Wal99] Jim Waldo. Jini technology architectural overview. Whitepaper, Sun Microsystems, January 1999.

[War83] D.H.D. Warrem. An abstract prolog instruction set. Technical Report 309, SRI International, October 1983.

[Wer88] E. Werner. Toward a theory of communication and cooperation for multiagent planning. In M.Y. Vardi, editor, *P*roceedings of the Second Conference of Theoretical Aspects of Reasoning about Knowledge, pages 129–144, San Mateo, California, 1988. Morgan Kaufmann.

[Wer89] E. Werner. *D*istributed Artificial Intelligence, volume II, chapter Cooperative Agents: A Unified Theory of Communication and Social Structure, pages 3–36. Morgan Kaufmann, San Mateo, California, 1989.

[Wer90] E. Werner. What can agents do together: A semantics of co-operative ability. In *P*roceedings of the Ninth European Conference on Artificial Intelligence, pages 694–701, 1990.

[Wer91] E. Werner. A unified view of information, intention and ability. In Y. Demazeau and J.P. Müller, editors, *P*roceedings of the Second European Workshop on Modelling Autonomous Agents and Multi-Agent Worlds, pages 109–126, Amsterdam, The Nederlands, 1991. Elsevier.

[Whi96] J. White. Mobile agents white paper. General Magic White paper, 1996.

[Wil99] Graham Williams. Integrated delivery of large-scale data mining systems. In Zaki and Ho [ZH99]. in conjunction with ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD99).

[WJ95] Michael J. Wooldridge and Nicholas R. Jennings. Intelligent agents: Theory and practice. *T*he Knowledge Engineering Review, 10(2):115–152, 1995.

[WJM94] T. Wittig, N. R. Jennings, and E. H. Mamdani. ARCHON — A framework for intelligent cooperation. *I*EE-BCS Journal of Intelligent Systems Engineering — Special Issue on Real-time Intelligent Systems in ESPRIT, 3(3):168–179, 1994.

[Woo92a] Michel J. Wooldridge. *T*he Logical Modelling of Computational Multi-Agent Systems. PhD thesis, Department of Computing, Manchester Metropolitan University, Manchester, UK, October 1992.

[Woo92b] Michel J. Wooldridge. The logical modelling of computational multi-agent systems. Technical Report MMU–DOC–94–01, Department of Computing Manchester Metropolitan University, Manchester, UK, October 1992.

[Woo93] S. Wood. Planning and decision making in dynamic domains. *E*llis Horwood, 1993.

[WR92] C. Weider and J. Reynolds. RFC 1308: Executive introduction to directory services using the X.500 protocol, March 1992. Status: INFORMATIONAL.

[WRH92] C. Weider, J. Reynolds, and S. Heker. RFC 1309: Technical overview of directory services using the X.500 protocol, March 1992. Status: INFORMATIONAL.

[WSG+97] R. Wirth, C. Shearer, U. Grimmer, T. Reinartz, J. Schlösser, C. Breitner, R. Engels, and G. Lindner. Towards process-oriented tool support for KDD. In Jan Komorowski and Jan Zytkow, editors, *P*roceedings of the 1st European Symposium on Principles of Data Mining and Knowledge Discovery, pages 243–253, Berlin, Germany, June 1997. Springer-Verlag.

[WV95] R. Weihmayer and H. Velthuijsen. *W*orldwide Intelligent Systems: Approaches to Telecomunications and Network Management, chapter Distributed AI and Cooperative Systems for Telecomunications. IOS Press, Amsterdam, The Nederlands, 1995.

[WV98] R. Weihmayer and H. Velthuijsen. *A*gent Technology: Foundations, Applications and Markets, chapter Intelligent Agents in Telecomunications. UNICOM-Seminars. Springer-Verlag, 1998.

[X/O94] X/Open. DCE time service. X/Open CAE Specification C310, November 1994.

[Zad93] L. Zadeh. The role of fuzzy logic in the management of uncertainty in expert systems. *F*uzzy Sets and Fuzzy Systems, 11:199–227, 1993.

[Zak99] Mohammed J. Zaki. Parallel sequence mining on SMP machines a data clustering algorithm on distributed memory machines a data clustering algorithm on distributed memory machines. In Zaki and Ho [ZH99]. in conjunction with ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD99).

[Zak00] M. J. Zaki. *L*arge-Scale Parallel Data Mining, chapter Parallel Sequence Mining on SMP Machines. 2000. Published in [ZH00a].

[Zea97] M. J. Zaki and et al. Parallel algorithm for fast discovery of association rules. *D*ata Mining and Knowledge Discovery: An International Journal, 1:343–373, December 1997.

[ZH99] Mohammed J. Zaki and Ching-Tien Ho, editors. *W*orkshop on Large-Scale Parallel KDD Systems, San Diego, CA, USA, August 1999. ACM. in conjunction with ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD99).

[ZH00a] M. J. Zaki and C. T. Ho, editors. *L*arge-Scale Parallel Data Mining, volume 1759 of *L*NCS. Springer-Verlag, 2000.

[ZH00b] Mohammed J. Zaki and Ching-Tien Ho. Workshop report: Large-scale parallel KDD systems. In *S*IGKDD Explorations [ZH99], pages 112–114. in conjunction with ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD99).

[ZHA99] M. J. Zaki, C. T. Ho, and R. Agrawal. Parallel classification for data mining on shared-memory multiprocessors. In *P*roceedings International Conference on Data Engineering, March 1999.

[ZLP96] Mohammed J. Zaki, Wei Li, and Srinivasan Parthasarathy. Customized dynamic load balancing for a network of workstations. In *5*th IEEE International Symposium on High-Performance Distributed Computing (HPDC), August 1996.

[ZOPL96] M. J. Zaki, M. Ogihara, S. Parthasarathy, and W. Li. Parallel data mining for association rules on shared-memory multi-processors. In *C*D-ROM Proceedings of Supercomputing'96, Pittsburgh, PA, November 1996. IEEE.

Apéndice A

MODELO DE SISTEMAS ASíNCRONOS: AUTOMATA E/S

A.1 Introducción

En la tesis de Mark Tuttle [LT87a] se define un modelo de representación para sistemas asíncronos denominado Autómata de Entrada/Salida (*I/O Automaton*). En base a este formalismo, Mark Tuttle y Nancy Lynch (coautora de la tesis) [LT87b, LT87c, LT89] definieron múltiples problemas distribuidos, posteriormente recogidos en [Lyn97]. El modelo de autómata E/S es una herramienta formal muy general válida para definir casi cualquier sistema concurrente asíncrono. Ciertas propiedades útiles son definibles a su vez como restricciones dentro de este formalismo, permitiendo la verificación de tales propiedades para todo sistema que se defina en base a autómatas E/S.

Un autómata E/S modeliza un componente dentro de un sistema distribuido que interactúa con otros componentes. Su definición se basa en una máquina de estados simple en la cual las transiciones entre los estados se denominan *acciones*. Estas acciones están asociadas a interacciones externas con otros componentes (de entrada y de salida) así como a operaciones internas del componente.

A.2 Autómata E/S

La definición de un autómata E/S se basa en el concepto de firma de un autómata.

A.2.1 Definición de Firma

Una **firma** o *signature* S es un conjunto de acciones, definidas como la tupla $(in(S), out(S), int(S))$ donde:

- $in(S)$ son las acciones de entrada. Operaciones del componente solicitadas por otros componentes.
- $out(S)$ son las acciones de salida. Operaciones de otros componentes solicitadas por este componente.
- $int(S)$ son las acciones internas. Operaciones del componente solicitadas y realizadas de forma interna.

En base a estos tres conjuntos de acciones se definen los siguientes conjuntos derivados:

- $ext(S) = in(S) \cup out(S)$ son las acciones externas.
- $local(S) = out(S) \cup int(S)$ son las acciones controladas localmente.
- $acts(S) = in(S) \cup out(S) \cup int(S)$ son todas las acciones.

A la tupla $(in(S), out(S), \emptyset)$ se la denomina firma externa o interfaz externo.

A.2.2 Definición de Autómata E/S

Un **autómata E/S** A se define como $(sig(A), states(A), start(A), trans(A), tasks(A))$, donde:

- $sig(A)$ es una firma.
- $states(A)$ es el conjunto de estados (no necesariamente finito).
- $start(A)$ es el subconjunto no vacío de $states(A)$ de estados iniciales.
- $trans(A)$ es una relación entre estados y transiciones, representada como $trans(A) \subseteq states(A) \times acts(sig(A)) \times states(A)$.
  Esta relación debe cumplir que para cada estado s y cada acción de entrada π, debe existir la transición $(s, \pi, s') \in trans(A)$.
- $tasks(A)$ es la partición de tareas, una relación de equivalencia definida sobre el conjunto de acciones locales $local(sig(A))$.

A lo largo de este apéndice se usará $acts(A)$ en lugar de $acts(sig(A))$ como representación de las acciones del autómata A. De forma análoga $in(A)$ por $in(sig(A))$, etc.

Se dice que el autómata A es cerrado si $in(A) = \emptyset$.

A cada uno de los elementos (s, π, s') de $trans(A)$ se denomina **transición** de A. Estas transiciones se denominan transiciones de entrada, de salida o internas, dependiendo del tipo de acción que sea π. Para esta misma transición (s, π, s') se dice que la acción π está habilitada por el estado s. Ya que todas las acciones de entrada están habilitadas en todos los estados, se dice que el autómata tiene entrada habilitada.

A.2.2.1 Definición de Ejecución

Se denomina **fragmento de ejecución** a una secuencia finita $s_0, \pi_1, s_1, \pi_2, s_2, \ldots, \pi_r, s_r$ o a una secuencia infinita $s_0, \pi_1, s_1, \pi_2, s_2, \ldots, \pi_r, s_r, \ldots$. Dichas secuencias son series alternadas de estados y acciones de forma tal que $\forall_{i \leq 0}(s_i, \pi_{i+1}, s_{i+1}) \in trans(A)$. Es importante resaltar que en caso de tratarse de una secuencia finita, ésta debe terminar en un estado (no en una acción). Si el estado inicial de la secuencia s_0 pertenece a $start(A)$ entonces se la denomina **ejecución**. Sean α y α' dos fragmentos de ejecución (α finito), tales que el último estado de α coincida con el primero de α', se denota como $\alpha.\alpha'$ a la concatenación de ambos fragmentos, una vez eliminado el estado común.

Se define $execs(A)$ como todo el conjunto de posibles ejecuciones de A. Se dice que un estado de A es alcanzable si existe alguna ejecución de A que termine en dicho estado.

A.2.2.2 Definición de Traza

Se denomina **traza** de una ejecución α, denotándose como $trace(\alpha)$, al comportamiento externo de la ejecución. Dicho comportamiento está definido por la subsecuencia de α compuesta sólo por acciones externas.

Se dice que β es una traza del autómata A si es traza de alguna ejecución de A. El conjunto de todas las trazas de A se denota como $trace(A)$.

A.3 Operaciones del Autómata E/S

A continuación se definen las operaciones de **composición** y **ocultamiento** de componentes.

A.3.1 Composición

La operación de **composición** define un nuevo autómata que modeliza un sistema compuesto por diferentes autómatas asociados a los diferentes componentes que lo conforman.

La composición de un conjunto de componentes está sujeta a una serie de restricciones. Se dice que la colección de firmas $\{S_i\}_{i\in I}$ asociadas a dichos componentes es **compatible** si $\forall_{i,j\in I} i \neq j$ se cumple:

① $int(S_i) \cap acts(S_j) = \emptyset$
Debido a que las acciones internas de un componente deben estar ocultas para el resto de componentes del sistema, se exige que los conjunto de las acciones internas de los componentes sean disjuntos.

② $out(S_i) \cap out(S_j) = \emptyset$
Los conjuntos de acciones de salida $out(A_i)$ deben de ser también disjuntos, de forma que cada acción del sistema es controlada por uno y sólo un componente.

③ La composición de conjuntos de componentes no está restringida únicamente a conjuntos finitos de componentes, pero si se exige que toda acción esté definida en un conjunto finito de componentes.

Se dice que una colección de autómatas es **compatible** si sus firmas lo son.

La **composición** $S = \prod_{i\in I} S_i$ de un conjunto compatible de firmas $\{S_i\}_{i\in I}$ se define como la firma con:

❒ $out(S) = \bigcup_{i\in I} out(S_i)$

❒ $in(S) = \bigcup_{i\in I} in(S_i)$

❒ $int(S) = \bigcup_{i\in I} int(S_i)$

La **composición** $A = \prod_{i\in I} A_i$ de un conjunto compatible de autómatas $\{A_i\}_{i\in I}$ se define como un nuevo autómata con:

❒ $sig(A) = \prod_{i\in I} sig(A_i)$ (Composición de las firmas).

❒ $states(A) = \prod_{i\in I} status(A_i)$ (Producto cartesiano de los estados).

❒ $start(A) = \prod_{i\in I} start(A_i)$ (Producto cartesiano de los estados iniciales).

❒ $trans(A)$ está formado por tuplas (s, π, s') tal que $\forall_{i\in I}$ si $\pi \in acts(A_i)$ entonces $(s_i, \pi, s'_i) \in trans(A_i)$ sino $s_i = s'_i$.

❒ $tasks(A) = \bigcup_{i\in I} tasks(A_i)$ (Unión de las tareas).

Siendo A el autómata resultante de la composición de los autómatas $\{A_i\}_{i\in I}$ y siendo α una ejecución de A, se denota como $\alpha|A_i$ a la proyección de la misma sobre el autómata A_i. $\alpha|A_i$ es la secuencia obtenida tras la eliminación de los pares π_r, s_r de α para los cuales π_r no es una acción de A_i y sustituyendo s_j restantes por $(s_j)_i$, es decir la componente ***i-esima*** (asociada a A_i) del estado s_j.

Teorema: Sea $\{A_i\}_{i\in I}$ un conjunto compatible de autómatas y $A = \prod_{i\in I} A_i$.

① Si $\alpha \in execs(A)$ entonces $\alpha|A_i \in execs(A_i)$ para todo $i \in I$.

② Si $\beta \in traces(A)$ entonces $\beta|A_i \in traces(A_i)$ para todo $i \in I$.

Teorema: Sea $\{A_i\}_{i\in I}$ un conjunto compatible de autómatas y $A = \prod_{i\in I} A_i$. Suponiendo que α_i es una ejecución de A_i para todo $i \in I$ y supóngase que β es una secuencia de acciones de $ext(A)$ tal que $\beta|A_i = trace(\alpha_i)$ para todo $i \in I$. Entonces existe una ejecución α de A tal que $\beta = trace(\alpha)$ y $\alpha_i = \alpha|A_i$ para todo $i \in I$.

Teorema: Sea $\{A_i\}_{i\in I}$ un conjunto compatible de autómatas y $A = \prod_{i\in I} A_i$. Suponiendo que β es una secuencia de acciones de $ext(A)$. Si $\beta|A_i \in traces(A_i)$ para todo $i \in I$, entonces $\beta \in traces(A)$.

A.3.2 Ocultamiento

La operación de **ocultamiento** permite la reclasificación de operaciones externas de un autómata como acciones internas. Esto permite que no sean invocadas desde otros autómatas y que no aparezcan en las trazas.

La operación de ocultamiento entre firmas se define como: Si S es una firma y $\Sigma \subseteq out(S)$ entonces $hide_\Sigma(S)$ se define como la nueva firma S' donde $in(S') = in(S)$, $out(S') = out(S) - \Sigma$ y $int(S') = int(S) \cup \Sigma$.

Una vez definida la operación entre firmas, el ocultamiento para autómatas se define como: Si A es un autómata y $\Phi \subseteq out(A)$ entonces $hide_\Phi(A)$ se define como el nuevo autómata A' obtenido reemplazando $sig(A)$ por $sig(A') = hide_\Phi(A)$.

A.4 Propiedades y Métodos de Prueba

Una vez descritos los conceptos básicos del modelo de autómata E/S, a continuación se muestran una serie de propiedades y métodos de prueba aplicables sobre dichos autómatas.

A.4.1 Equitatividad

Un fragmento de ejecución α de un autómata A se dice que es **equitativo** si se cumplen las siguientes condiciones para todo elemento C del conjunto $tasks(A)$ de clases de equivalencia de acciones:

① Si α es finito, entonces C no está habilitado para el estado final de α.

② Si α es infinito, entonces α contiene o un número infinito de ocurrencias de acciones de C o un número infinito de ocurrencias de estados en los que C no está habilitado.

Se denota el conjunto de ejecuciones equitativas de A como $fairexecs(A)$. Del mismo modo, se dice que β es una traza equitativa si β es una traza de una ejecución equitativa de A. El conjunto de trazas equitativas se denota como $fairtaces(A)$.

Teorema: Sea $\{A_i\}_{i\in I}$ un conjunto compatible de autómatas y $A = \prod_{i\in I} A_i$.

① Si $\alpha \in fairexecs(A)$ entonces $\alpha|A_i \in fairexecs(A_i)$ para todo $i \in I$.

② Si $\beta \in fairtraces(A)$ entonces $\beta|A_i \in fairtraces(A_i)$ para todo $i \in I$.

Teorema: Sea $\{A_i\}_{i\in I}$ un conjunto compatible de autómatas y $A = \prod_{i\in I} A_i$. Suponiendo que α_i es una ejecución equitativa de A_i para todo $i \in I$ y supóngase que β es una secuencia de acciones de $ext(A)$ tal que $\beta|A_i = trace(\alpha_i)$ para todo $i \in I$. Entonces existe una ejecución equitativa α de A tal que $\beta = trace(\alpha)$ y $\alpha_i = \alpha|A_i$ para todo $i \in I$.

Teorema: Sea $\{A_i\}_{i\in I}$ un conjunto compatible de autómatas y $A = \prod_{i\in I} A_i$. Suponiendo que β es una secuencia de acciones de $ext(A)$. Si $\beta|A_i \in fairtraces(A_i)$ para todo $i \in I$, entonces $\beta \in fairtraces(A)$.

Teorema: Sea A un autómata E/S:

① Si α es una ejecución finita de A, entonces existe una ejecución equitativa de A que comienza por α.

② Si β es una traza finita de A, entonces existe una traza equitativa de A que comienza por β.

③ Si α es una ejecución finita de A y β es cualquier secuencia (finita o infinita) de acciones de entrada de A, entonces existe una ejecución equitativa $\alpha \cdot \alpha'$ de A tal que la secuencia de acciones de entrada en α' sea β.

④ Si β es una traza finita de A y β' es cualquier secuencia (finita o infinita) de acciones de entrada de A, entonces existe una ejecución equitativa $\alpha\dot{\alpha}'$ de A tal que $trace(\alpha) = \beta$ y la secuencia de acciones de entrada en α' sea β'.

A.4.2 Afirmaciones Invariantes

Se denominan **afirmaciones invariantes** o sencillamente **invariantes** a toda propiedad de un autómata A que es verdadera para todos los estados alcanzables de A. Estas propiedades son muy interesantes para la verificación de ciertas características de los sistemas. Estas propiedades son generalmente comprobables en base a pruebas por inducción.

A.4.3 Propiedades de Traza

En muchas ocasiones el comportamiento de un sistema se estudia como una *caja negra* de la cual se comprueban únicamente las trazas de las ejecuciones (o ejecuciones equitativas). Una **propiedad de traza** P consiste en:

- $sig(P)$ una firma que no contiene acciones internas.
- $traces(P)$ un conjunto (finito o infinito) de secuencias de acciones de $acts(sig(P))$.

La afirmación de que un autómata A satisface una propiedad de traza P implica que cumple alguna de estas dos condiciones:

- $extsig(A) = sig(P)$ y $traces(A) \subseteq traces(P)$.
- $extsig(A) = sig(P)$ y $fairtraces(A) \subseteq traces(P)$.

La **composición de propiedades de traza** se define como, para un conjunto de propiedades traza $\{P_i\}_{i \in I}$, cuyas firmas sean compatibles. $P = \prod_{i \in I} P_i$ es la propiedad de traza que cumple:

- $sig(P) = \prod_{i \in I} sig(P_i)$.
- $traces(P)$ es el conjunto de secuencias β de acciones externas de P tales que $\beta | acts(P_i) \in traces(P_i)$ para todo $i \in I$.

A.4.4 Seguridad

Se define la propiedad de **seguridad** o de **traza segura** como la propiedad de traza P con las siguientes condiciones:

- $traces(P) \neq \emptyset$.
- $traces(P)$ es de *prefijo cerrado*, es decir si $\beta \in traces(P)$ y β' es un prefijo finito de β, entonces $\beta' \in traces(P)$.
- $traces(P)$ es de *límite cerrado*, es decir si $\beta_1, \beta_2, \ldots$ es una secuencia infinita de secuencias finitas de $traces(P)$ y por cada i de β_i este es prefijo de β_{i+1}, entonces la única secuencia β que es el límite de β_i bajo la extensión sucesiva en ese orden también pertenece a $traces(P)$.

A.4.5 Vivacidad

La propiedad de **vivacidad** es otra propiedad de traza que determina que toda secuencia de $acts(P)$ tiene alguna extensión en $traces(P)$, es decir que existen trazas que comiencen por dicha secuencia.

Teorema: Si P cumple las propiedades de **seguridad** y **vivacidad**, entonces P define todo el conjunto de las posibles secuencias, finitas e infinitas, de $acts(P)$.

Teorema: Sea P es una propiedad de traza cualquiera, tal que $traces(P) \neq \emptyset$, entones existe un propiedad S de **seguridad** y una propiedad L de **vivacidad**, tales que:

① $sig(S) = sig(L) = sig(P)$.

② $traces(P) = traces(S) \cap traces(L)$.

A.4.6 Razonamiento por Composición

El mecanismo de **razonamiento por composición** permite analizar un sistema completo enfocando la prueba de determinadas propiedades de un componente determinado dentro del sistema. Esta herramienta de análisis se basa en los dos siguientes teoremas.

Teorema: Sea $\{A_i\}_{i\in I}$ un conjunto compatible de autómatas y $A = \prod_{i\in I} A_i$ su composición. Sea $\{P_i\}_{i\in I}$ un conjunto compatible de propiedades de traza cuya composición es $P = \prod_{i\in I} P_i$.

① Si $extsig(A_i) = sig(P_i)$ y $traces(A_i) \subseteq traces(P_i)$ para todo i, entonces $extsig(A) = sig(P)$ y $traces(A) \subseteq traces(P_i)$.

② Si $extsig(A_i) = sig(P_i)$ y $fairtraces(A_i) \subseteq traces(P_i)$ para todo i, entonces $extsig(A) = sig(P)$ y $fairtraces(A) \subseteq traces(P_i)$.

Teorema: Sea $\{A_i\}_{i\in I}$ un conjunto compatible de autómatas y $A = \prod_{i\in I} A_i$ su composición. Sea P una propiedad de seguridad tal que $acts(P) \cap int(A) = \emptyset$ y $in(P) \cap out(A) = \emptyset$.

① Si A_i satisface la propiedad P para todo $i \in I$, entonces A también cumple P.

② Si A es un autómata cerrado, A satisface P y $acts(P) \subseteq ext(A)$, entonces $traces(A)|acts(P) \subseteq traces(P)$.

③ Si A es un autómata cerrado, A satisface P y $acts(A) = ext(A)$, entonces $traces(A) \subseteq traces(P)$.

A.4.7 Pruebas Jerárquicas

Una estrategia de prueba muy útil para sistemas complejos es la denominada **pruebas jerárquicas**. Esta estrategia se basa en la definición de diferentes niveles de detalle en la descomposición de un sistema. De esta forma es posible descender desde el nivel más abstracto al mas detallado por medio de un refinamiento sucesivo. Cada uno de estos niveles estará compuesto por un autómata cuyas operaciones tendrán una menor o mayor granularidad.

El mecanismo de refinamiento consiste en ir detallando la estructura del autómata, definiendo **relaciones de simulación**. Una **relación de simulación** f desde el autómata A hasta el autómata B es una relación entre $states(A)$ y $states(B)$ de la forma $f \subseteq states(A) \times states(B)$. Para esta función se ha adoptado la notación $u \in f(s)$ para indicar que $(u,s) \in f$. Una relación de simulación de A a B debe cumplir:

① Si $s \in start(A)$, entonces $f(s) \cap start(B) \neq \emptyset$.

② Si s es un estado alcanzable de A, $u \in f(s)$ es un estado alcanzables de B y $(s,\pi,s') \in trans(A)$, entonces existe un fragmento de ejecución α de B que comienza en el estado u y termina en otro estado $u' \in f(s')$ tal que $trace(\alpha) = trace(\pi)$.

Teorema: Si existe una relación de simulación de A hasta B, entonces $traces(A) \subseteq traces(B)$.

Apéndice B

SINTAXIS DE DEFINICIÓN DE COMPONENTES

Índice General

B.1 Introducción

Este documento define el lenguaje en el que se define los componentes de la arquitectura MOIRAE. Partiendo de la definición del componente es posible por medio de una determina aplicación generar el código de dicho componente para cualquier lenguaje de programación. El contenido de este apéndice ha sido extraído de [Peñ00].

B.2 Consideraciones Iniciales

El objeto de la sintaxis presentada en este informe es disponer de una herramienta para la descripción de componentes en dos sentidos:

① Definiendo cada uno de los elementos que conforman el componente desde el punto de vista externo. Esto implica proporcionar al resto de entidades externas al componente la información necesaria para hacer uso del componente.

② Diseñando el comportamiento interno del componente. Determinando la algorítmica de realización de las acciones externas declaradas anteriormente.

Esta división de facetas de la sintaxis se puede interpretar de forma análoga lo que en lenguajes de alto nivel son los ficheros de cabecera o definición y los archivos de implementación.

B.2.1 Convenios de Notación

- `texto`: Indican símbolos o palabras reservadas del lenguaje.
- *texto*: Son otras reglas de la gramática. Como caso especial de este tipo:
  - *C_code*: Es un fragmento de código C o C++ similar al cuerpo de una función, sin definiciones de funciones o tipos y con cualquier tipo de estructuras de control o llamadas a otras funciones. Habitualmente, estos fragmentos de código aparecen entre llaves. Ejemplo:

```
{
  int aux,var=3;
  for(int i=0;i<10;i++)
  {
    aux=foo(i);
    var+=aux<<i;
  }
  baa(var);
}
```

  - *C_predicate*: Es otro fragmento de código C o C++ pero, en este caso debe de ser válido como argumento de una función, es decir, no debe contener secuencias de control o definición de variables. Todos estos fragmentos de código aparecen entre corchetes. Ejemplo:

```
[ x>23 && (is_correct(y) || my_bool) ]
```

Para ambos fragmentos de código, la referencia a elementos (comandos, servicios, variables de estado, etc.) del componente se realiza anteponiendo un dolar (`$`) al identificador del elemento. Si por ejemplo, se desea invocar al comando `flush_buffer` desde uno de estos códigos, entonces se hará como: `$flush_buffer(...)`.

- `<EMPTY>`: Representa un predicado vacío, es decir ningún tipo de lexema.
- `IDENTIFIER`: Es un identificador válido del sistema. Los identificadores aceptados han de cumplir las mismas reglas que en la sintaxis de C o C++.

B.3 Fichero de Definición

B.3.1 Estructura

```
definition_file     :=  definition_regions
definition_regions  :=  <EMPTY>
                     |  definition_region definition_regions
definition_region   :=  import_region
                     |  external_def_region
                     |  domain_region
                     |  component_region
```

B.3.2 Área `import`

```
import_region       :=  import_line
                     |  import_line import_region
import_line         :=  #import file_name <NEW_LINE>
file_name           :=  relative_file_name
                     |  absolute_file_name
relative_file_name  :=  "file_path"
absolute_file_name  :=  <file_path>
file_path           :=  FILE_SYSTEM_ENTRY
                     |  FILE_SYSTEM_ENTRY/file_path
```

B.3.3 Área `definitions`

```
external_def_region  :=  definitions { C_code }
```

Este fragmento de código debe incluir la definición de funciones, tipos y similares y no la implementación de las mismas (salvo funciones `inline`). Su contenido debe ser similar al que puede aparecer en un fichero de cabecera (`.h`) de C o C++.

B.3.4 Área `domaindef`

```
domain_region       :=  domain_definition
                     |  domain_definition domain_region
domain_definition   :=  domaindef type domain_name ;
domain_name         :=  IDENTIFIER
```

B.3.5 Área component

```
component_region       :=  domponent_definition
                         |  component_definition component_region
component_definition   :=  component component_name inheritance
                           { component_elements }
component_name         :=  IDENTIFIER
inheritance            :=  <EMPTY>
                         |  : superclasses
superclasses           :=  component_name
                         |  component_name , superclasses

component_elements   :=  <EMPTY>
                       |  component_element component_elements
component_element    :=  services_def
                       |  commands_def
                       |  primitives_def
                       |  events_def
                       |  status_def
                       |  relations_def

services_def     :=  services { operation_list }
commands_def     :=  commands { operation_list }
primitives_def   :=  primitives { operation_list }
operation_list   :=  operation
                   |  operation operation_list

operation          :=  type IDENTIFIER (arg_list)
                       operation_events ;
arg_list           :=  <EMPTY>
                     |  arg , arg_list
arg                :=  scope type IDENTIFIER
scope              :=  in | out | inout
operation_events   :=  <EMPTY>
                     |  operation_event operation_events
operation_event    :=  throw event

events_def       :=  events { event_list }
event_list       :=  event
                   |  event event_list
event            :=  IDENTIFIER (event_arg_list);
event_arg_list   :=  <EMPTY>
                   |  event_arg , even_arg_list
event_arg        :=  type

status_def    :=  status { status_list }
status_list   :=  status_var
                |  status_var status_list
status_var    :=  type IDENTIFIER

relations_def    :=  relations { relations_list }
relations_list   :=  relation
                   |  relation relations_list
relation         :=  relation_scope component_type IDENTIFIER
                     optionality identification cardinality
relation_scope   :=  public | private
component_type   :=  IDENTIFIER
optionality      :=  <EMPTY> | optional
identification   :=  <EMPTY> | identified | anonymous
cardinality      :=  <EMPTY> | simple | multiple
```

B.4 Fichero de Implementación

B.4.1 Estructura

```
implementation_file      :=  #component IDENTIFIER
                             implementation_regions
implementation_regions   :=  <EMPTY>
                           |  implementation_region implementation_regions
implementation_region    :=  options_region
                           |  service_region
                           |  command_region
                           |  primitive_region
                           |  equivaletship_region
                           |  code_region
```

B.4.2 Descripción de opciones del componente

```
options_region       :=  option_declaration
                       |  option_declaration options_region
option_declaration   :=  #option IDENTIFIER = option_value <NEW_LINE>
```

El símbolo *option_value* está compuesto por cualquier cadena de caracteres distinta de un fin de línea salvo que justo antes del fin de línea aparezca el carácter *slash* (\). Por ejemplo:

```
#option agent_type=reactive:0
#option middleware=CORBA:MICO:2.3.0
#option message_channel=DI-DAMISIS Message Channel
#option gui=gtk:1.2.3
#option long_option=This option is very long option   \
                    as well as a very useless option, \
                    the end.
```

Las opciones actualmente soportadas serán:

① `agent_type`: Identifica el tipo de modelo de razonamiento y representación del agente encargado de la parte de control. Los tipos de valores posibles son:

  ① `reactive:0`: Son agentes reactivos de nivel básico. Operan mediante un mecanismo de estímulo/reacción elemental.

  ① `reactive:1`: Son agentes reactivos también pero permiten evaluar condiciones antes de reaccionar ante un estímulo.

  ① `deliberative:prolog`: Usa un motor prolog como mecanismo de inferencia de razonamiento del agente.

  ① `deliberative:BDI`: Implementa un agente de tipo BDI (*Belief–Desire–Intention*).

② `middleware`: Determina el entorno de comunicaciones que se usara para desarrollar los componentes. La utilización de un *middleware* determinado implica el uso de sus servicios como soporte al desarrollo de los mecanismos internos del sistema. Actualmente sólo se plantea soportar `CORBA:MICO:`*versión*.

③ `message_channel`: Indica que el componente va a disponer de acceso a un canal compartido de mensajes.

④ `gui`: Proporciona el soporte para el desarrollo de interfaces de usuario por medio de librerías de desarrollo de los mismos (Motif, GTK, X11, QT, ...). Si no se habilita esta opción es posible que si el código del componente tenga conflictos con las librerías de dichos interfaces.

B.4.3 Descripción de comandos y servicios

```
service_region     :=  service operation_header operation_description
command_region     :=  command operation_header operation_description
operation_header   :=  IDENTIFIER (arg_list)

operation_description   :=  case_list
                        |   case_description
case_list               :=  case
                        |   case case_list
case                    :=  case_condition : case_description
case_condition          :=  [C_predicate]
                        |   {C_code}

case_description   :=  statement_type {C_code}
statement_type     :=  action | event | status
```

B.4.4 Descripción de primitivas

```
primitive_region   :=  primitive operation_header primitive_code
primitive_code     :=  [C_predicate]
                   |   {C_code}
```

B.4.5 Relación de Equivalencia

```
equivalentship_region   :=  equivalent (IDENTIFIER,IDENTIFIER) equivalent_code
equivalent_code         :=  [C_predicate]
                        |   {C_code}
```

B.4.6 Código Adicional

```
code_region   :=  code {C_code}
```

Este fragmento de código puede incluir definición e implementación de funciones.

B.5 Dominios Básicos

```
type                :=  Short | Long | Float | Double
                        |  String | Boolean | Char
                        |  component
                        |  nominal_set
component           :=  IDENTIFIER
nominal_set         :=  atomic_nominal_set
                        |  nominal_set + nominal_set
atomic_nominal_set  :=  { valuelist }
value_list          :=  IDENTIFIER
                        |  IDENTIFIER , value_list
```

B.6 Ejemplos

B.6.1 Diseño de un *Buffer*

B.6.1.1 Buffer.def

```
component Buffer
{
  // Servicios del Componente
  services
  {
     void read_item (out String item)   // Prototipo del servicio
        throw BUFFER_IS_EMPTY();        // Posible evento a lanzar
     void write_item (in  String item)
        throw BUFFER_IS_FULL();
  }
  // Comandos del Componente (Operaciones de uso interno)
  commands
  {
     boolean init();                    // Hace las veces de constructor
     boolean flush_buffer()
        throw BUFFER_IS_EMPTY();
  }
  // Lista de Eventos
  events
  {
     BUFFER_IS_EMPTY();
     BUFFER_IS_FULL();
```

```
  }
  // Variables de estado
  status
  {
     short   size;
     short   used;
  }
  // Predicados compuestos de estado (sensores)
  primitives
  {
     boolean space_available();
  }
}
```

B.6.1.2 Buffer.moi

```
// Define el componente a implementar (solo uno por fichero)
#component Buffer

// Implementacion del servicio
service read_item(I):
{
  // Lista de Casos
  [$used==0] : event  { $BUFFER_IS_EMPTY(); }
  [$used!=0] : action { $I.set(buffer.head()); buffer.pop_front(); }
               status { $used=$used-1; } ;
}

service write_item(I)
{
  [!$space_available()] : event  { $BUFFER_IS_FULL(); }
  [ $space_available()] : action { buffer.push_back($I.value()); }
                          status { $used=$used+1; }
}

command init()
{
```

```
 action [true]
 status { $size=10; $used=0; }
}

command flush_buffer(I)
{
  [$used==0] : event  { $BUFFER_IS_EMPTY(); }
  [$used!=0] : action { buffer=new Buffer(10); return(true); }
               status { $size=10; $used=0; }
}

// Definicion de la primitiva
primitive space_available() [$used<$size]

// Predicado de equivalencia de estado
equivalent(A,B) [$A.used==$B.used]
```

B.6.2 Jerarquía de Elementos de Acceso a Datos

B.6.2.1 DataAccess.def

```
definitions
{
  #define BLOCK_SIZE 1024
  typedef char[BLOCK_SIZE] Block;
}
component DataAccess
{
  services
  {
     void read_block (in Long key, out Block block)
        throw WRONG_KEY(Long)
        throw KEY_UNUSED(Long)
        throw MEDIA_ERROR(Short);
     void write_item (in Long key, in  Block block)
        throw WRONG_KEY(Long)
        throw KEY_USED(Long)
```

```
        throw MEDIA_ERROR(Short);
  }
  events
  {
     WRONG_KEY(Long key);
     KEY_UNUSED(Long key);
     KEY_USED(Long key);
     MEDIA_ERROR(Short code);
  }
}
```

B.6.2.2 Buffered.def

```
#import "Buffer.def"
component Buffered
{
  relations
  {
     private Buffer buffer: identified simple;
  }
  status
  {
     boolean hasBuffer;
  }
  commands
  {
     boolean init();
     boolean set_buffer(in Buffer buff)
       throw BUFFER_REDEFINED();
     boolean insert(in String item)
       throw BUFFER_UNDEFINED();
     boolean extract(out String item)
       throw BUFFER_UNDEFINED();
  }
  events
  {
    BUFFER_UNDEFINED();
```

```
    BUFFER_REDEFINED();
  }
}
```

B.6.2.3 Buffered.moi

```
#component Buffered

command init()
{
  action [true]
  status { $hasBuffer=false; }
}

command set_buffer(in Buffer buff)
{
  [ $hasBuffer] : event { $BUFFER_REDEFINED(); }
  [!$hasBuffer] : action { $buffer.assign(buff); }
                  status { $hasBuffer=true; }
}

command insert(in String item)
{
  [!$hasBuffer] : event { $BUFFER_UNDEFINED(); }
  [ $hasBuffer] : action { $buffer.write_item(item); }
}

command extract(in String item)
{
  [!$hasBuffer] : event { $BUFFER_UNDEFINED(); }
  [ $hasBuffer] : action { $buffer.read_item(item); }
}
```

B.6.2.4 FileDataAccess.def

```
#import "DataAccess.def"
#import "Buffered.def"
```

```
component FileDataAccess : DataAccess, Buffered
{
  services
  {
     void read_block (in Long key, out Block block)
        throw WRONG_KEY(Long)
        throw KEY_UNUSED(Long)
        throw MEDIA_ERROR(Short);
     void write_item (in Long key, in  Block block)
        throw WRONG_KEY(Long)
        throw KEY_USED(Long)
        throw MEDIA_ERROR(Short);
  }
  commands
  {
     void read_fetch(in Long key)
        throw READ_FETCH(Long);
     void write_fetch(in Long key,in Block block)
        throw WRITE_FETCH(Long,BLOCK);
  }
  events
  {
     READ_FETCH(Long);
     WRITE_FETCH(Long,Block);
  }
}
```

B.6.2.5 FileDataAccess.moi

```
#component FileDataAccess

service read_block(in Long key, out Block block)
{
  [key<=1000 || key> 2000] : event { $WRONG_KEY(key); }
  [key> 1000 && key<=2000] : action { $read_fetch(key);
                                      block=cached_block(); }
}
```

```
service write_block(in Long key, in Block block)
{
  [key<=1000 || key> 2000] : event { $WRONG_KEY(key); }
  [key> 1000 && key<=2000] : action { $write_fetch(key,block); }
}

command read_fetch(in Long key)
{
  event { $READ_FETCH(key); }
}

command write_fetch(in Long key,in Block block)
{
  event { $WRITE_FETCH(key,block); }
}

code
{
  private Block cached_block()
  {
    /* Function Code */
  }
}
```