Universidad Tecnológica de la Mixteca.

Arquitectura de Cómputo Distribuido Basada en Objetos Componente: Análisis e Implantación.

TESIS PROFESIONAL

Que para obtener el Titulo de

Ingeniero en Computación

Presenta

Carlos Benavides Martínez.

Acatlima, Huajuapan de León, Oaxaca Invierno de 1998.

Con todo mi amor, respeto y admiración este trabajo se lo dedico a mi madre, la señora:

Josefina Benavides Martínez.

Agradecimientos

Muy en especial al Dr. Anatoli S. Koulinitch por su amistad y confianza, por haber aceptado ser mi asesor de tesis, y sobre todo gracias por su paciencia.

Al Ing. Gerardo García Hernández, por estar siempre al pendiente de este trabajo.

A el Ing. Francisco Espinosa Maceda, por estar siempre en la mejor disponibilidad de ayudarme, de escuchar y discutir información, que fue indispensable para encausarme a la realización de este presente trabajo.

A Oscar Cuevas por echarme la mano en la documentación de las referencias de ActiveX.

Al Departamento de Posgrado por las facilidades de instalaciones, apoyo a equipo de cómputo y material diverso, los cuales fueron indispensables.

A todos los profesores de la carrera que en realidad compartieron sus conocimientos a lo largo de mi formación profesional.

A mi madre y abuelitos por su cariño, por su ejemplo de trabajo y superación y sobre todo, por su apoyo incondicional.

Indice

Esta arquitectura se implementó en computadoras personales y un Servidor HP, bajo los sistemas operativos: Windows NT Server 4.0 y Windows 95/98; Red LAN TCP/IP; utilizando los siguientes lenguajes y herramientas para el desarrollo y pruebas:

- Microsoft Visual C++ 5.0
- Microsoft Visual Basic 5.0
- MathWorks Fuzzy Logic ToolBox.
- Microsoft Distributed COM 1.2 for Windows 95/98.
- Microsoft DCOMCNFG 1.1
- Microsoft OLE/COM Object Viewer 2.1
- Microsoft Process Viewer 4.0
- DameWare Development DBtnMap 1.1

Resultados.

- Se implementó la arquitectura computacionalmente.
- Se definieron las especificaciones propias de un sistema distribuido bajo la tecnología de objetos componentes (componentes).
- Se verificó que la arquitectura funcionara y se adaptara.

Contenido de la tesis.

El primer capítulo fundamenta los conceptos introductorios a los sistemas distribuidos y paralelos, también se revisan las características y problemas de diseño más relevantes que debe contener un sistema de esta categoría, su importancia y estado en los actuales sistemas de información. En el segundo capítulo se presenta un marco de referencia para el análisis de los sistemas de software, basados en componentes, el cual está constituido por un *modelo general de composición* y por un *conjunto de problemas relativos a la reusabilidad de componentes*, a demás se mencionan las principales características que ofrece la tecnología de componentes como soporte a los sistemas de gran escala y en particular a los distribuidos. En el tercer capítulo se presentan algunas de las primeras y más importantes especificaciones para la actividad del computo distribuido basado en objetos, también se describen algunos productos basados en estos modelos. En el capítulo cuarto se *describe* la *arquitectura de objetos componente*, para desarrollo de computo distribuido, la cual se propone en este trabajo y se hacen refencias a algunos otros trabajos relacionados. En el capítulo cinco se aborda la implementación de un prototipo de sistema mínimo de toma de decisión distribuido de recuperación y presentación de información con mecanismos de inferencia basados en lógica difusa, orientado al desarrollo del estado de Oaxaca en el nivel educativo y socioeconómico, con base en la arquitectura aquí propuesta, finalmente se mencionan las conclusiones y trabajo futuro.

Introducción.

Fueron dos los factores que se consideraron como las principales fuentes de inspiración para el desarrollo de este trabajo.
El primero de ellos es el hecho de que hay una gran cantidad de problemas que pueden ser expresados en forma de trabajo distribuido, a través de módulos software que llevan a cabo tareas individuales, los cuales cooperan en tiempo de ejecución para lograr un objetivo común. Estos módulos pueden ser ejecutados en forma simultánea (concurrente) en una o más computadoras o procesadores, generalmente obteniendo como beneficio un incremento en la velocidad de procesamiento, además de facilitar la cooperación misma entre ellos.
Consideramos que dos o más programas de computadora operan en forma cooperativa cuando todos trabajan para lograr un solo objetivo común, y todos ellos son capaces de intercambiar información de manera transparente, [Jennings et al, 1998].

El segundo es el hecho de que actualmente hay una creciente necesidad de cómputo distribuido y paralelo, conforme aumenta el uso de estos sistemas, cobra mayor importancia el estudio de los problemas que se plantean en ellos. Una de las características que diferencia a estos sistemas del resto es que las entidades computacionales que viven en ellos, pueden ser consideradas como unidades de software dispersas físicamente, que interoperan en entornos heterogéneos y cambiantes. Por otro lado sabemos que las ideas y conceptos de la orientación a objetos nos permite describir mejor el comportamiento de tales sistemas y tratar los problemas que en ellos se plantean.

En la arquitectura aquí propuesta, cada entidad computacional es una unidad individual concurrente, con una tarea que cumplir, su objetivo. Para realizarlo ha de saber adaptarse al medio, utilizar los recursos que éste le ofrece y negociar sus servicios. Pero al tratarse de un medio cambiante, la entidad se enfrenta con problemas que dificultan su misión: *nuevos servidores y servicios aparecen y otros desaparecen o dejan de funcionar*; por lo que es necesario comunicarse con otras entidades cuya interfaz[i] de servicio es desconocida, además el hacer frente a esos nuevos retos no deberá complicar innecesariamente el núcleo fundamental de la unidad de software.

Aparece por tanto la necesidad de que las unidades computacionales, desarrollen distintas habilidades como pueden ser la autonomía, adaptabilidad, robustez o la seguridad, y que lo puedan hacer de forma transparente, [Jennings et al, 1998].

---

[i] La interfaz de servicio se define como un contrato entre un servicio de proveedor y un servicio de cliente. Una vez publicadas, las interfaces no deben modificarse para evitar romper las dependencias externas.

Se necesita entonces disponer de un esquema y una metodología que permita diseñar unidades de software con tales propiedades, y que también permita incorporar tales facultades a unidades ya existentes. También, hemos de tener en cuenta que en los sistemas distribuidos deja de existir una visión global del sistema, siendo únicamente posible disponer de visiones locales del mismo, es decir desde las unidades de software.

¿Qué se propone en esta tesis?

Se propone una arquitectura, que dicta la forma en como una aplicación distribuida podría ser desarrollada, con base en el paradigma de objetos distribuidos, la manera en como los *objetos componente*[i] (componentes), se comunican y cooperan entre si, en forma concurrente, y como son distribuidos a través de un sistema de computo, donde interoperan a través de un *middleware*[ii] adoptado, además podrá servir como guía sistemática en el análisis e implementación de un sistema de computo distribuido basado en objetos, en general. Finalmente esta arquitectura sirve como base en la implementación de un prototipo de sistema mínimo de toma de decisión distribuido, el cual estará apoyado con la integración de bases de datos y bases de conocimiento, así como en reglas de producción con mecanismos de inferencia basados en lógica difusa (Fuzzy Logic).

Objetivo de la tesis.

El objetivo principal de la tesis es *implementar* la arquitectura de computo distribuido aquí propuesta, en un prototipo de sistema mínimo de toma de decisión distribuido para el desarrollo del estado de Oaxaca en el nivel educativo y socioeconómico.

Producto.

El producto esperado es la implementación del prototipo de sistema mínimo de toma de decisión distribuido de recuperación y presentación de Información orientado al desarrollo del estado de Oaxaca en el nivel educativo y socioeconómico, basado en la arquitectura aquí propuesta.

---

[i] Objeto Componente o simplemente *componente* es un encapsulamiento físico de uno o más servicios a los que se tiene acceso a través de las interfaces, externamente, todo lo que se conoce de un componente es su interfaz.

[ii] Middleware es la infraestructura que funciona como una capa protectora entre los programas de aplicación y la tecnología subyacente; en el capitulo 3, se amplia el concepto de *middleware* dentro del contexto de sistemas distribuidos.

Capitulo 1

LOS SISTEMAS DE CÓMPUTO DISTRIBUIDOS.

En este tiempo en el que la modernización exige cada vez mas que la gente se encuentre informada y que las empresas salgan ahora a buscar nuevos mercados ante la globalización comercial, la computadora es una herramienta de apoyo que nos permite obtener ventajas que no se podían tener antes. Sin embargo, uno de los problemas más fuertes, es poder compartir información de manera eficiente, segura y transparentemente.

Las nuevas tecnologías en cómputo distribuido y paralelo permiten abrir nuevas opciones de automatización y mecanismos para compartir información entre entidades geográficamente distantes, ya que su misión es llegar a interrelacionar de la manera más optima a un conjunto de entidades entre sí, mismas que cuentan con las mas variadas plataformas de cómputo. Por lo anterior es necesario puntualizar dos cosas:

- Primeramente, que se necesita *comunicar objetos* no importando la distancia.
- Segundo, que debemos hacerlo no importando la marca, modelo o sistema operativo que se tenga.

Éste primer capitulo presenta los fundamentos introductorios a los sistemas distribuidos y paralelos, además se revisan los problemas y características más relevantes de diseño para un sistema de esta categoría, su importancia y estado actual en los sistemas de información, así como las *perspectivas* que se perfilan y que en los siguientes capítulos se abordaran.

1.1 Conceptos de sistemas distribuidos y paralelos.

Parece claro que a pesar de los avances tecnológicos conseguidos en los últimos años, la tecnología del silicio está llegando a su límite. Si se quieren resolver problemas más complejos y de mayores dimensiones se deben buscar nuevas alternativas tecnológicas. Una de estas alternativas en desarrollo es el paralelismo y la distribución. Mediante el paralelismo se pretende conseguir la distribución del trabajo entre las diversas CPUs disponibles en el sistema de forma que realicen el trabajo simultáneamente, con el objetivo de aumentar considerablemente el rendimiento total.

De acuerdo con [Hwang et al, 1988], para que dos programas se puedan ejecutar en paralelo se deben verificar ciertas condiciones, que se presentan a continuación:

- Sea I_i el conjunto de todas las variables de entrada necesarias para ejecutar el proceso P_i.

- Sea O_i el conjunto de todas las variables de salida generadas por el proceso P_i.

Luego, las condiciones de Bernstein para dos procesos P_1 y P_2 son las siguientes:

1. $I_1 \cap O_2 = \varnothing$
2. $I_2 \cap O_1 = \varnothing$
3. $O_1 \cap O_2 = \varnothing$

Si se cumplen las tres condiciones entonces se dice que P_1 y P_2 pueden ser ejecutados en paralelo y se denota como $P_1 \parallel P_2$. Esta relación de paralelismo es *conmutativa*, esto es: $P_1 \parallel P_2 \Longrightarrow P_2 \parallel P_1$, pero no es *transitiva* $P_1 \parallel P_2$ y $P_2 \parallel P_3$ $\not\Longrightarrow P_1 \parallel P_3$.

Las condiciones de Bernstein aunque definidas para procesos, son validas al nivel de instrucciones. Para conseguir un buen nivel de paralelismo es necesario que el hardware y el software se diseñen conjuntamente, para mayor detalle consultar [Hwang et al, 1988] y [Bacon, 1994].

Sabemos que existen dos visiones del paralelismo, según [Bacon, 1994]:

- *Paralelismo hardware*: Es el paralelismo definido por la arquitectura de la máquina.
- *Paralelismo software*: Es el paralelismo definido por la estructura del programa. Se manifiesta en las instrucciones que no tienen interdependencias.

El paralelismo hardware o software, se presenta, a su vez, en dos formas:

- *Paralelismo de control*: Se pueden realizar dos o más operaciones simultáneamente. Se presenta en los pipelines y las múltiples unidades funcionales. El programa no necesita preocuparse de este paralelismo, pues se realiza a nivel hardware.

- *Paralelismo de datos*: Una misma operación se puede realizar sobre varios elementos simultáneamente.

Con relación al paralelismo hardware, Michael Flynn, realizó la siguiente clasificación de arquitecturas de computadoras, para un estudio más profundo ver [Bacon, 1994]:

1. *SISD (Single Instruction stream over a Single Data stream)*: Es la arquitectura de las máquinas secuenciales convencionales de un sólo procesador.
2. *SIMD (Single Instruction stream over a Multiple Data stream)*: Es la arquitectura de las computadoras con hardware para proceso vectorial.
3. *MISD (Multiple Instruction stream over a Single Data stream)*: Es la arquitectura de las computadoras que poseen un conjunto de procesadores que ejecutan diferentes instrucciones sobre los mismos datos.
4. *MIMD (Multiple Instruction stream over a Multiple Data stream)*: Es la arquitectura más genérica para los computadores paralelos, ya que es aplicable a cualquier tipo de problema, al contrario que las dos anteriores.

Por otro lado, existen ocasiones en las que se confunde el término concurrencia con el término paralelismo. La concurrencia se refiere a un *paralelismo potencial*, que puede o no darse. Existen dos formas de concurrencia, de acuerdo con [Bacon, 1994]:

- *Concurrencia implícita*: Es la concurrencia interna al programa, por ejemplo cuando un programa contiene instrucciones independientes que se pueden realizar en paralelo, o existen operaciones de E/S que se pueden realizar en paralelo con otros programas en ejecución. Está relacionada con el paralelismo hardware.

- *Concurrencia explícita*: Es la concurrencia que existe cuando el comportamiento concurrente es especificado por el diseñador del programa. Está relacionada con el paralelismo software.

Es habitual encontrar en la bibliografía el término de *programa concurrente* en el mismo contexto que el de *programa paralelo* o *distribuido*. Existen diferencias sutiles entre estos conceptos, de acuerdo con [Hwang et al, 1988]:

- *Programa concurrente*: Es aquél que define acciones que pueden realizarse simultáneamente, es decir que expresa el paralelismo potencial, donde hay que resolver los problemas de comunicación y sincronización resultantes.
- *Programa paralelo*: Es un programa concurrente diseñado para su ejecución en un hardware paralelo.
- *Programa distribuido*: Es un programa paralelo diseñado para su ejecución en una red de procesadores autónomos que no comparten la memoria.

Cabe mencionar que el término *concurrente* es aplicable a cualquier programa que presente un comportamiento paralelo actual o potencial. En cambio el término *paralelo* o *distribuido* es aplicable a aquel programa diseñado para su ejecución en un entorno específico. Cuando se emplea un solo procesador para la ejecución de programas concurrentes se habla de *seudoparalelismo*.

Algunos autores, como [Gerlhof et al, 1994] y [Tanenbaum, 1993] son más formales en la definición de sistemas distribuidos y paralelos, según los cuales existen tres dimensiones que pueden ser distribuidas y/o paralelizables: *el hardware, el control y los datos*. Dentro de este formalismo, un sistema es auténticamente distribuido y paralelo *si y sólo si* es distribuido y paralelo en las tres dimensiones. Este esquema identifica a los sistemas cliente-servidor como un modelo en el que se distribuye el hardware, el control, pero no la información sino a través de paradigmas de solicitud-respuesta, mientras que en un sistema distribuido y paralelo, no existe esta división de funciones: todos los procesos son cooperativos y realizan tanto funciones de cliente (petición), como de servidor (respuesta), en forma concurrente.

1.2 ¿ Por qué desarrollar aplicaciones distribuidas?

Las aplicaciones distribuidas y paralelas son mucho más escalables que sus contrapartes monolíticas[i]. Sabemos que algunas situaciones del medio que nos rodea son inherentemente distribuidas y/o paralelas, tales como: juegos multiusuarios, aplicaciones de Groupware[ii] y de CSCW[iii], como aplicaciones de teleconferencias, procesamiento de imágenes digitales para diagnostico, simulaciones en tiempo-real, etc; por mencionar algunas. Para que estas aplicaciones puedan ser realizadas, los beneficios de una robusta infraestructura de cómputo son obvios.

También como ejemplo, podríamos considerar una fábrica de robots, cada uno de los cuales contiene una poderosa computadora para el manejo de visión, planeación, comunicación y otras tareas, cuando un robot de la línea de ensamble nota que una parte por instalar es defectuosa, le pide al robot del departamento de partes que le traiga una refacción.

---

[i] Las aplicaciones monolíticas son aquellas en el que todas las unidades de una aplicación se combinan en un solo programa integrado que funciona sólo en una máquina.

[ii] Groupware, representa el software de apoyo para trabajar en grupo y se trata de un nuevo mercado con características del software para organizaciones y del software individual.

[iii] *Computer-Supported Cooperative Work (CSCW)*, surge un nuevo y multidisciplinario campo llamado). Utilizando la experiencia y colaboración de muchos especialistas, incluidos profesionales de la computación y de las ciencias sociales.

Si todos los robots actúan como dispositivos periféricos y el sistema se puede programar de esta manera, también se considera como un sistema distribuido. Como último ejemplo clásico, pensemos en un enorme banco con cientos de sucursales por todo el mundo. Cada oficina tiene una computadora maestra para guardar la cuentas locales y el manejo de las transacciones locales. Además, cada computadora tiene la capacidad de comunicarse con las otras sucursales y con una computadora central en las oficinas centrales. Si las transacciones se pueden realizar sin importar dónde se encuentre el cliente o la cuenta, también se podría considerar como un sistema distribuido y/o paralelo, para conocer un mayor numero de aplicaciones ver [Cockburn et al, 1996].

1.3 Características y problemas en el diseño de sistemas distribuidos.

Aunque los sistemas distribuidos y paralelos tienen sus aspectos fuertes, también tienen sus debilidades. El peor de los problemas es el software, pues no se tiene mucha experiencia en el *diseño, implantación y uso del mismo*, por ejemplo: ¿Qué tipos de sistemas operativos, lenguajes de programación y aplicaciones son adecuados para estos sistemas? ¿Cuánto deben saber los usuarios de la distribución? ¿Qué tanto debe hacer el sistema y qué tanto deben hacer los usuarios?. Un segundo problema potencial es debido a las redes de comunicación, estas pueden perder mensajes, al saturarse la red, ésta debe remplazarse o añadir una segunda. En ambos casos, hay que tender cables en una parte de uno o más edificios; o bien hay que remplazár las tarjetas de interfaz de la red (por ejemplo, por fibras ópticas). Una vez que el sistema llega a depender de la red, la pérdida o saturación de ésta puede negar algunas de las ventajas que el sistema distribuido debiera conseguir.

También hay que considerar, el hecho ya descrito de que los datos sean fáciles de compartir es una ventaja, pero se puede convertir en un arma de dos filos. Si las personas pueden tener acceso a los datos en todo el sistema, entonces también pueden tener acceso a datos con los que no tienen nada que ver. En otras palabras, es necesario contender con fuertes conceptos de seguridad y autenticación.

Además hay que considerar el problema de la heterogeneidad y compatibilidad entre sistemas operativos, lenguajes de programación, bases de datos, protocolos de redes, etc, es decir toda la infraestructura necesaria que integrara el sistema a implantar es con frecuencia un problema poder interoperar entre la gran diversidad de productos disponibles. En el capitulo 3 se revisaran las posibilidades ofrecidas por las tecnologías de objetos distribuidos y el middleware, con relación al problema de la integración de sistemas heterogéneos. En seguida se harán referencias en forma breve a algunos de los aspectos claves del *diseño* en la construcción de sistemas distribuidos.

Transparencia.

Tal vez el aspecto más importante sea la forma de lograr la imagen de un sistema, entonces podemos identificar los siguientes tipos de transparencia en un sistema distribuido, según se describe en [Tanenbaum, 1993]:

Transparencia de localización, se refiere al hecho de que en un sistema distribuido. Los usuarios no pueden indicar la localización de los recursos de hardware y software, como los CPU, impresoras, archivos y bases de datos.

Transparencia de migración, significa que los recursos deben moverse de una posición a otra sin tener que cambiar sus nombres. Así, el simple hecho de que un archivo o directorio ha emigrado de un servidor a otro lo ha obligado a adquirir un nuevo nombre, puesto que el sistema de montajes remotos no es transparente con respecto de la migración.

Trasparencia de réplica, significa que el sistema distribuido, es libre de fabricar por su cuenta copias adicionales de los archivos y otros recursos sin que lo noten los usuarios o exista alguna inconsistencia con las operaciones del sistema.

Transparencia respecto a la concurrencia, para lograr esta forma de transparencia es necesaria que el sistema *cierre,* en forma automática un recurso, una vez que alguien haya comenzado a utilizarlo, eliminando el bloqueo sólo hasta que termine el acceso, de acuerdo a ciertas configuraciones preestablecidas en el acceso a los datos o procesos concurrentes. De esta forma, todos los recursos tendrán un acceso secuencial, nunca concurrente, aparentemente para el usuario o grupos de usuarios.

Flexibilidad.

Es importante que el sistema sea flexible, ya que apenas estamos aprendiendo a construir sistemas distribuidos y paralelos. Es probable que este proceso tenga muchas salidas falsas. Las decisiones de diseño que ahora aparezcan razonables podrían demostrar ser incorrectas posteriormente. La mejor forma de evitar los problemas es mantener abiertas las opciones, ver [Tanenbaum, 1993].

Confiabilidad.

Uno de los objetivos originales de la construcción de sistemas distribuidos fue hacerlos más confiables que los sistemas con un procesador. Un sistema ampliamente confiable debe ser muy disponible, pero eso no es suficiente. Los datos confiados al sistema no deben *perderse o mezclarse* de manera alguna; si los archivos se almacenan de manera redundante en varios servidores, todas las copias deben ser consistentes. En general, mientras se tengan más copias, será mejor la disponibilidad, pero también aumentará la probabilidad de que sean inconsistentes, en particular si las actualizaciones son frecuentes. Los diseñadores de todos los sistemas distribuidos deben tener en mente este dilema en todo momento.

Otro aspecto de la *confiabilidad* general es la *seguridad*. Los archivos y otros recursos deben ser protegidos contra el uso no autorizado. Aunque este mismo aspecto aparece en los sistemas con un único procesador, es más severo en los sistemas distribuidos. En general, los sistemas distribuidos se pueden diseñar de forma que escondan las fallas; es decir, ocultarlos de los usuarios, si un servicio de archivo o algún otro servicio se construye a partir de un grupo de servidores con una cooperación cercana, entonces sería posible construirlo de forma que los usuarios no noten la pérdida de uno o dos servidores, de no ser por cierta degradación del desempeño, [Martin et al , 1997] , [Tanenbaum, 1993].

Desempeño

La construcción de sistemas distribuidos transparentes, flexibles y confiables no hará que uno gane premios si este es muy lento. Se pueden utilizar diversas métricas del desempeño. Como el *tiempo de respuesta*, pero también lo es el *rendimiento* (número de trabajos por hora), uso del sistema y cantidad consumida de la capacidad de la red, etc.

Además es frecuente que el resultado de cualquier parámetro dependa de la naturaleza de éste. El problema del desempeño se complica por el hecho de que la *comunicación*, factor esencial en un sistema distribuido es algo lenta por lo general, - factores físicos.

Por otro lado los trabajos que implican gran número de pequeños cálculos, en particular aquellos que interactúan en gran medida con otros, pueden ser la causa de algunos problemas en los sistemas distribuidos con una comunicación lenta en términos relativos, [Tanenbaum, 1993].

Escalabilidad.

La mayor parte de los sistemas distribuidos están diseñados para trabajar con unos cuantos cientos de CPU. Es posible que los sistemas futuros tengan mayores órdenes de magnitud y las soluciones que funcionen bien para 200 maquinas fallen de manera total para 200 millones. La cuestión es si los métodos que se desarrollan en la actualidad podrán escalarse hacia tales grandes sistemas. Para realizar un estudio más profundo de estos conceptos puede revisar la bibliografía de [Hwang et al, 1988] y [Bacon, 1994].

1.4 Conceptos de entidades básicas en sistemas distribuidos.

Como se menciono anteriormente, la imagen que presenta y la forma de pensar de los usuarios de un sistema distribuido, queda determinada en gran medida por el software del sistema, no por el hardware. Por su propia naturaleza sabemos que el software es vago y amorfo, en esta sección entenderemos la funcionalidad de las *entidades* más significativas en los sistemas distribuidos y paralelos en el ámbito del sistema operativo, con el objetivo de fortalecer las ideas que se retomaran en la arquitectura que se propone en este trabajo, - ver capitulo 4; también se mencionaran algunas áreas comunes de estudio relacionados con éstos sistemas, como el de la concurrencia.

1.4.1 Concepto de proceso.

El concepto central de todo sistema operativo es el *proceso,* todo lo demás se organiza con relación a este concepto.

Sabemos que las computadoras modernas realizan varias tareas al mismo tiempo. Por ejemplo, mientras se ejecuta un programa de usuario, la computadora puede estar leyendo del disco e imprimiendo un documento en la impresora.

En un sistema de multiprogramación, la CPU alterna de un programa a otro, ejecutando cada uno durante milisegundos, y conmutando a otro inmediatamente, de tal forma que al usuario se le proporciona cierta sensación de *ejecución paralela*, como si la computadora realizase varias tareas al mismo tiempo.

Aunque, estrictamente, la CPU ejecuta en un determinado instante un solo programa, durante un segundo puede haber trabajado con varios de ellos, dando una apariencia de paralelismo, [Tanenbaum, 1993].

Es en estos casos es cuando se tiende a hablar de *seudoparalelismo*, indicando la rápida conmutación entre los programas en la CPU, distinguiéndolo del paralelismo real de hardware, donde se realizan cálculos en la CPU a la vez que operan los dispositivos de E/S. Ya que es complicado controlar las distintas actividades paralelas, los diseñadores de sistemas emplean el **modelo de procesos** para facilitar la utilización del paralelismo, de acuerdo con [Martin et al, 1997] y [Tanenbaum, 1993]. Según lo describen los autores mencionados, en este modelo, todo el software de la computadora se organiza en *procesos secuenciales*, o simplemente *procesos*.

Un **proceso** es un programa en ejecución, incluyendo los valores del contador de programa (PC), registros y variables del programa. Conceptualmente, cada proceso tiene su propia CPU virtual. En realidad, lo que sucede es que la CPU física alterna entre los distintos procesos. Esta alternancia rápida es lo que se llama **multiprogramación**.

Si la CPU conmuta entre procesos, no es probable que un proceso que se vuelva ejecutar lo haga en las mismas condiciones en que lo hizo anteriormente. Por lo tanto, los procesos no se pueden programar con hipótesis implícitas a cerca del tiempo. El resultado de un proceso debe ser independiente de la velocidad de cálculo y del entorno en que se encuentre, de tal forma que sea reproducible.

La *diferencia entre proceso y programa es sutil*, pero crucial. El programa es el algoritmo expresado en una determinada notación, mientras que el proceso es la actividad, que tiene un programa, entrada, salida y estado.

También sabemos que una CPU puede compartirse entre varios procesos empleando un algoritmo de planificación, que determina cuando se debe detener un proceso en ejecución para dar servicio a otro distinto.

Como se menciono arriba, cada proceso es una entidad independiente, con su contador de programa y su estado interno. A hora los 3 estados básicos en que puede encontrarse un proceso - ver figura y tabla que siguen, de acuerdo con [Bacon, 1994] y [Tanenbaum, 1993] son:

1. *En ejecución (RUNNING):* El proceso está utilizando la CPU en el instante dado.
2. *Listo (READY):* El programa está en condición de ser ejecutado (pasar al estado En Ejecución) pero no lo está ya que se está ejecutando otro proceso temporalmente en la CPU.
3. *Bloqueado (SUSPEND):* El proceso no se puede ejecutar ya que se encuentra a la espera de un evento externo, incluso aunque la CPU se encuentre libre.

Evolución del estado de un proceso

1. El proceso se bloquea en espera de datos.
2. El planificador elige otro proceso
3. El planificador elige este proceso
4. Los datos están disponibles

Figura 1-1. Estados básicos de un proceso.

Como se describe en [Bacon, 1994], sabemos que en el *modelo de procesos* conviven *procesos de usuario* con *procesos del sistema*. La parte del sistema operativo que se encarga de realizar la conmutación entre procesos, y de esta forma repartir la utilización de la CPU, es el *planificador*, el cual tiene una enorme importancia en el funcionamiento óptimo del sistema.

El modelo de procesos se implementa en el sistema operativo mediante una **tabla de procesos**, con una entrada por cada proceso. Cada entrada contiene información sobre el estado del proceso, el contador de programa (PC), el puntero de pila (SP), la asignación de memoria, el estado de los archivos abiertos, información contable, información sobre la planificación del proceso, y otros datos a conservar al producirse el *cambio de contexto* de proceso. En general son datos sobre la administración del proceso, la administración de memoria y la administración de archivos.

A continuación se presentan la información más habitual que suele encontrarse en la tabla de procesos en el núcleo de un sistema operativo, según se describe en [Bacon, 1994] y [Tanenbaum, 1993]:

Administración de Procesos	*Administración de Memoria*	*Administración de Archivos*
- Identificador de proceso	- Puntero a segmento de texto	- Máscara de archivos
- Contador de programa (PC)	- Puntero a segmento de datos	- Directorio raíz
- Palabra de estado	- Puntero a segmento de pila	- Directorio de trabajo
- Puntero de Pila (SP)	- Estado de salida	- Descriptores de archivo
- Registros generales	- Identificador de proceso	- Identificador usuario efectivo
- Estado del Proceso	- Proceso padre	- Identificador grupo efectivo
- Máscara de señales pendientes	- Grupo de procesos	- Parámetros llamadas al sistema
- Tiempo de inicio del proceso	- Identificador usuario real	- Máscara de señales
- Tiempo de utilización de CPU	- Identificador usuario efectivo	
- Tiempo de CPU del hijo	- Identificador grupo real	
- Puntero a la cola de mensajes	- Identificador grupo efectivo	

Figura 1-2 Campos habituales de la Tabla de Procesos

Ahora podemos mencionar en forma breve, como se realiza la ejecución *concurrente*[i] en una sola CPU con varios dispositivos de E/S, lo que da la ilusión de que varias tareas se realizan en forma simultanea - cambio de contexto, según se describe en [Bacon, 1994]:

Sabemos que cada clase de dispositivo (discos duros, discos flexibles, reloj, terminal) tiene asociado una zona de memoria que se llama *vector de interrupción*, la cual contiene la dirección del procedimiento de servicio de la interrupción. Cuando se produce una interrupción, el contador de programa y la palabra de estado del proceso actual en ejecución, son enviados a la pila por el hardware de la interrupción. Entonces, la máquina salta a la dirección indicada en el vector de interrupción correspondiente. Esto es lo que realiza el hardware.

Inmediatamente después, el procedimiento de servicio de la interrupción guarda los registros en la entrada de la tabla de procesos correspondiente al proceso activo, se elimina la información depositada por el hardware en la pila y se llama a la rutina que procesa la interrupción. Después se debe encontrar el proceso que inició la solicitud y que estará a la espera de la interrupción. Normalmente dicho proceso estará en estado de *bloqueado*, por lo que debe ser despertado, para lo cual se pasa al estado *listo*, y a continuación se llama al planificador.

El planificador se encargará de decidir qué proceso pasa a ejecutarse a continuación según el algoritmo de planificación implementado, y finalmente se inicia el proceso activo seleccionado por el planificador.

[i] En la siguiente sección, se mencionan las características más significativas de la concurrencia.

Estas acciones se realizan constantemente en el sistema, así pues, se debe intentar optimizar al máximo y reducir el tiempo empleado en esta tarea. Otras veces el cambio de proceso activo se debe a una interrupción temporal, ya que normalmente un proceso activo tiene un *quantum*[1] de tiempo asignado, de tal forma que tras haber consumido ese tiempo de ejecución, el proceso es desalojado de forma similar a la explicada para el manejo de una interrupción. En realidad, lo que se produce es una interrupción del reloj (que es periódica).

Al procedimiento explicado se le suele llamar **cambio de contexto** o **context-switch**, pues lo que en realidad ocurre es que se pasa de ejecutar un proceso a ejecutar otro, todo de una forma transparente y rápida; para un estudio mas detallado se recomienda la bibliografía de [Hwang et al, 1988] [Bacon, 1994].

1.4.2 Características de la concurrencia.

Como se describe en [Hwang et al , 1988], la concurrencia es el nombre dado a las notaciones y técnicas empleadas para expresar el paralelismo potencial y para resolver los problemas de comunicación y sincronización resultantes. La concurrencia proporciona una abstracción sobre la que estudiar el paralelismo sin tener en cuenta los detalles de implementación. Esta abstracción ha demostrado ser muy útil en la escritura de programas claros y correctos empleando las facilidades de los lenguajes de programación modernos.

El problema básico en la escritura de un programa concurrente es identificar qué actividades pueden realizarse concurrentemente. Además la programación concurrente es mucho más difícil que la programación secuencial clásica por la dificultad de asegurar que el programa concurrente es correcto.

Los procesos concurrentes tienen las siguientes características, de acuerdo con [Bacon, 1994]:

- *Indeterminismo*: Las acciones que se especifican en un programa secuencial tienen un orden total, pero en un programa concurrente el orden es parcial, ya que existe una incertidumbre sobre el orden exacto de ocurrencia de ciertos sucesos, esto es, existe un indeterminismo en la ejecución. De esta forma si se ejecuta un programa concurrente varias veces puede producir resultados diferentes partiendo de los mismos datos.

[1] Un quantum es simplemente la cantidad de tiempo que el proceso tiene permitido para su ejecución antes de que el sistema operativo compruebe si existe un proceso de mayor prioridad listo para la ejecución.

- *Interacción entre procesos*: Los programas concurrentes implican interacción entre los distintos procesos que los componen, son:

    1. Los procesos que comparten recursos y compiten por el acceso a los mismos.
    2. Los procesos que se comunican entre sí para intercambiar datos.

En ambas situaciones se necesita que los procesos sincronicen su ejecución, para evitar conflictos o establecer contacto para el intercambio de datos. La interacción entre procesos se logra mediante variables compartidas o bien mediante el paso de mensajes. Además la interacción puede ser *explícita*, si aparece en la descripción del programa, o *implícita*, si aparece durante la ejecución del programa.

- *Gestión de recursos*: Los recursos compartidos necesitan una gestión especial. Un proceso que desee utilizar un recurso compartido debe *solicitar* dicho recurso, esperar a *adquirirlo*, *utilizarlo* y después *liberarlo*. Si el proceso solicita el recurso pero no puede adquirirlo en ese momento, es suspendido hasta que el recurso está disponible. La gestión de recursos compartidos es problemática y se debe realizar de tal forma que se eviten situaciones de retraso indefinido (espera indefinidamente por un recurso) y de deadlock (bloqueo indefinido o abrazo mortal).

- *Comunicación*: La comunicación entre procesos puede ser síncrona, cuando los procesos necesitan sincronizarse para intercambiar los datos, o asíncrona, cuando un proceso que suministra los datos no necesita esperar a que el proceso receptor los recoja, ya que los deja en un buffer de comunicación temporal.

A continuación se mencionan algunos de los problemas más significativos que destacan en la concurrencia, al nivel de procesos.

1.4.3 Problemas de la concurrencia.

Los procesos concurrentes a diferencia de los procesos secuenciales tienen una serie de problemas muy particulares derivados de las características de la concurrencia, a continuación se describen en forma breve cada uno de ellos, un estudio más profundo puede ser revisado en la bibliografía de [Gerlhof et al, 1994] y [Bacon, 1994]:

1. *Violación de la exclusión mutua*: En ocasiones ciertas acciones que se realizan en un programa concurrente no proporcionan los resultados deseados. Esto se debe a que existe una parte del programa donde se realizan dichas acciones que constituye una *sección crítica*, es decir, es una parte del programa en la que se debe garantizar que si un proceso accede a la misma, ningún otro podrá acceder. Se necesita pues garantizar la exclusión mutua, [Hwang et al , 1988].

2. *Bloqueo mutuo o Deadlock*: Un proceso se encuentra en estado de deadlock si está esperando por un suceso que no ocurrirá nunca. Se puede producir en la comunicación de procesos y más frecuentemente en la gestión de recursos. Existen cuatro condiciones necesarias para que se pueda producir *deadlock*:

   - Los procesos necesitan acceso exclusivo a los recursos.
   - Los procesos necesitan mantener ciertos recursos exclusivos mientras esperan por otros.
   - Los recursos no se pueden obtener de los procesos que están a la espera.
   - Existe una cadena circular de procesos en la cual cada proceso posee uno o más de los recursos que necesita el siguiente proceso en la cadena.

3. *Retraso indefinido o starvation*: Un proceso se encuentra en starvation si es retrasado indefinidamente esperando un suceso que puede no ocurrir nunca. Esta situación se puede producir si la gestión de recursos emplea un algoritmo en el que no se tenga en cuenta el tiempo de espera del proceso, [Hwang et al, 1988].

4. *Injusticia o unfairness*: Se pueden dar situaciones en las que exista cierta injusticia con relación a la evolución de un proceso. Se deben evitar estas situaciones de tal forma que se garantice que un proceso evoluciona y satisface sus necesidades sucesivas en algún momento.

5. *Espera ocupada*: En ocasiones cuando un proceso se encuentra a la espera por un suceso, una forma de comprobar si el suceso se ha producido es verificando continuamente si el mismo se ha realizado ya. Esta solución de espera ocupada es muy poco efectiva, porque desperdicia tiempo de procesamiento, y se debe evitar. La solución ideal es suspender el proceso y continuar cuando se haya cumplido la condición de espera.

Estos son los problemas más significativos que se presentan en los procesos concurrentes, a hora podemos estudiar la forma en como se comunican los procesos y de que manera se resuelven algunos de los problemas de la concurrencia, anteriormente descritos.

1.4.4 Comunicación entre procesos.

En muchas ocasiones es necesario que dos procesos se comuniquen entre sí, o bien sincronicen su ejecución de tal forma que uno espere por el otro. La comunicación entre procesos se denota como IPC (InterProcess Communication).

A veces los procesos comparten un espacio de almacenamiento común (un archivo, una zona de memoria, etc.) en la que cada uno puede leer o escribir. Los problemas surgen cuando el acceso a dicha zona no está organizado, sino que es más o menos aleatorio o más bien dependiente de las condiciones de la máquina. A estas situaciones se las denomina **condiciones de competencia**. Se deben solucionar las condiciones de competencia, esto es, garantizar que sólo accede un proceso al mismo tiempo a la zona compartida. El objetivo es garantizar la **exclusión mutua,** ver detalles en [Bacon, 1994]; una forma de garantizar que si un proceso utiliza una variable, archivo compartido o cualquier otro objeto compartido, los demás procesos no pueden utilizarlo.

Otra forma más abstracta de plantear el problema es pensar que normalmente un proceso realiza su tarea independientemente, pero a veces necesita acceder a una serie de recursos compartidos con otros procesos o bien debe llevar a cabo acciones críticas que pueden llevar a conflictos. A esta parte del programa se la llama **sección crítica** ver detalles en [Bacon, 1994]. Se debe evitar, pues, que dos procesos se encuentren en su sección crítica al mismo tiempo.

La condición de la exclusión mutua, no es suficiente para evitar todos los conflictos que se pueden producir entre procesos paralelos que cooperan y emplean datos compartidos. Existen *cuatro condiciones* que se deben cumplir para obtener una buena solución, que en [Hwang et al, 1988], [Bacon, 1994] y [Tanenbaum, 1993], son descritas como las siguientes:

1. Garantizar *la exclusión mutua*: Dos procesos no deben encontrarse al mismo tiempo en su sección crítica.

2. *Indeterminación*: No se deben hacer hipótesis a cerca de la velocidad o el número de procesadores, durante la ejecución de los procesos.

3. Ninguno de los procesos que se encuentran fuera de su sección crítica puede bloquear a otros procesos.

4. Evitar *el retraso indefinido*: Ningún proceso debe esperar eternamente a entrar en su sección crítica.

Existen varias soluciones para garantizar estos principios. Así para garantizar la *exclusión mutua* tenemos las siguientes opciones, según se describe en [Hwang et al, 1988] y [Bacon, 1994]:

1. *Desactivar las interrupciones*: Consiste en desactivar todas las interrupciones del proceso antes de entrar a la sección crítica, con lo que se evita su desalojo de la CPU, y volverlas activar a la salida de la sección crítica. Esta solución no es buena, pues la desactivación de las interrupciones deja a todo el sistema en manos de la voluntad del proceso de usuario, sin que exista garantía de reactivación de las interrupciones.

2. *Emplear variables de cerradura*: Consiste en poner una variable compartida, una cerradura, a 1 cuando se va a entrar en la sección crítica, y devolverla al valor 0 a la salida. Esta solución en sí misma no es válida porque la propia cerradura es una variable crítica. La cerradura puede estar a 0 y ser comprobada por un proceso A, éste ser suspendido, mientras un proceso B chequea la cerradura, la pone a 1 y puede entrar a su sección crítica; a continuación A la pone a 1 también, y tanto A como B se pueden encontrar en su sección crítica al mismo tiempo. Existen muchos intentos de solución a este problema, a continuación se mencionan algunas de ellas, de acuerdo con [Tanenbaum, 1993]:

    - El algoritmo de Dekker
    - El algoritmo de Peterson
    - La instrucción hardware TSL: Test & Set Lock.

Las soluciones de Dekker, Peterson y TSL son correctas pero emplean espera ocupada. Básicamente lo que realizan es que cuando un proceso desea entrar en su sección crítica comprueba si está permitida la entrada o no. Si no está permitida, el proceso se queda en un bucle de espera hasta que se consigue el permiso de acceso. Esto produce un gran desperdicio de tiempo de CPU, pero pueden aparecer otros problemas como la espera indefinida.

Una solución más adecuada es la de bloquear o dormir el proceso (sleep) cuando está a la espera de un determinado evento, y despertarlo (wakeup) cuando se produce dicho evento.

Esta idea es la que emplean las siguientes soluciones, estas descripciones fueron tomadas de [Hwang et al, 1988], [Bacon, 1994], quienes proporcionan fundamentos más claros en la materia, respecto de otros autores:

1. **Semáforos:** Esta solución fue propuesta por Dijkstra [Bacon, 1994]. Un semáforo es una variable contador que controla la entrada a la sección crítica. Las operaciones P (o *wait*) y V (o *signal*) controlan, respectivamente, la entrada y salida de la sección crítica. Cuando un proceso desea acceder a su sección crítica realiza un *wait(var_semaf)*. Lo que hace esta llamada es, si *var_semaf == 0* entonces el proceso se bloquea, sino *var_semaf = var_semaf -1*. Al finalizar su sección crítica, libera el acceso con *signal(var_semaf)*, que realiza *var_semaf = var_semaf + 1*. Las acciones se realizan de *forma atómica*, de tal forma que mientras se realiza la operación P o V ningún otro proceso puede acceder al semáforo. Son el mecanismo más empleado para resolver la exclusión mutua, pero son restrictivos y no totalmente seguros (depende de su implementación), aunque son empleados en ocasiones para implementar otros métodos de sincronización.

2. **Secciones críticas condicionales:** Esta solución fue propuesta por Hoare [Hwang et al , 1988] y [Bacon, 1994] como mejora de los semáforos. Consiste en definir las variables de una sección crítica como recursos con un nombre, de esta forma la sección crítica se precede con el nombre del recurso que se necesita y opcionalmente una condición que se debe cumplir para acceder a la misma. Es un buen mecanismo, pero no suele ser soportado por la mayoría de los lenguajes de programación.

3. **Monitores:** Esta solución también fue propuesta por Brinch Hansen [Bacon, 1994] y [Tanenbaum, 1993]. Un monitor es un conjunto de procedimientos, variables y estructuras de datos que se agrupan en un determinado módulo. Los procesos pueden llamar a los procedimientos del monitor cuando lo deseen para realizar las operaciones sobre los datos compartidos, pero no pueden acceder directamente a las estructuras de datos internas del monitor.
   Su principal propiedad para conseguir la exclusión mutua es que sólo un proceso puede estar activo en un monitor en cada momento.

4. **Paso de mensajes o transferencia de mensajes:** Es sin duda, el modelo más empleado para la comunicación entre procesos. Para la comunicación se emplean las primitivas *send* (para enviar un mensaje) y *receive* (para poner al proceso a la espera de un mensaje). Su ventaja es que se puede emplear para la sincronización de procesos en sistemas distribuidos. La comunicación puede ser *síncrona* o *asíncrona*. Empleando mensajes se pueden implementar semáforos o monitores, y viceversa.

En los sistemas operativos tradicionales al crear un nuevo proceso, lo que en realidad suele hacer el sistema es realizar una copia exacta del padre en el hijo (operación de clonación). Tenemos, entonces, dos procesos iguales.

La ineficiencia de este procedimiento, aparece cuando el proceso padre es de un tamaño considerable, y el proceso hijo sólo se crea para realizar una pequeña tarea y después finalizar su ejecución, ya que es necesario volcar todo el contenido del proceso padre en el proceso hijo. Además, como padre e hijo son dos procesos totalmente independientes (salvo esa relación padre-hijo), su comunicación o compartición de datos es complicada y poco eficiente. Además, cuando se produce el cambio de contexto entre padre e hijo, la mayoría de los datos no cambian, con lo que otro tipo de gestión podría optimizar este tipo concreto de cambio de contexto, problema que se trata de resolver por medio de los *threads*, concepto que se abordara en la siguiente sección.

1.4.5 Concepto de hilo o thread.

Estamos consientes que aveces existen ocasiones en las que sería conveniente que *dos procesos trabajasen de forma concurrente pero con una serie de datos comunes*. Este problema es difícil de resolver normalmente, ya que los procesos son independientes y la comunicación de los datos debe realizarse mediante algún mecanismo de comunicación entre procesos (IPC), como pueden ser: *semáforos, memoria compartida, paso de mensajes*, etc; - como se comento en secciones anteriores parece claro que se introduce una cierta complejidad en la resolución de este tipo de problemas.

Una solución que parece razonable consistiría en que los procesos compartiesen el espacio de direcciones (la memoria), de esta forma no sería necesario su intercomunicación, ya que todos tendrían acceso a la misma zona de memoria.

Se plantean sin embargo otros problemas, como es el *acceso concurrente* a una misma posición de memoria, evitando los problemas de inconsistencia de datos. De ahí surge la idea de los *threads o hilos, los hilos son flujos de control independientes dentro de un mismo proceso que comparten datos globales (variables globales, archivos, etc.), pero poseen una pila, variables locales y contador de programa, propios*. Se les suele llamar "procesos de peso ligero" porque su contexto es menor que el contexto de un *proceso pesado* (procesos soportados por el kernel del sistema operativo). Por lo tanto, los cambios de contexto entre hilos son menos costosos que los cambios de contexto entre procesos.

La noción de hilo, como flujo de control, se remonta al menos, al año 1965. Un "proceso hilado" (*threaded process*, proceso construido mediante hilos) tiene el potencial de incrementar el rendimiento total de la aplicación en cuanto a productividad y tiempo de respuesta se refiere, mediante *ejecución de código asíncrono y paralelo.*

También podemos considerar que la ejecución de código paralelo se realiza mediante la ejecución de dos o más partes de un proceso en dos o más procesadores en un instante de tiempo dado. La ejecución asíncrona se puede realizar conmutando la ejecución del segmento de código actual que se bloquea por alguna razón, a otro segmento, según como se describe en [Bacon, 1994] y [Tanenbaum, 1993].

Al igual que los procesos, los hilos pueden estar en alguno de los siguientes estados:

- *En ejecución (Running)*: cuando el hilo posee la CPU y se encuentra activo.
- *Bloqueado (Suspend)*: cuando el hilo se encuentra esperando algún evento o a la espera de que otro libere el bloqueo por el que se encuentra detenido.
- *Listo o preparado (Ready)*: cuando el hilo está preparado para su ejecución, y se encuentra a la espera de ser elegido por el planificador.
- *Terminado (Finished)*: cuando el hilo ha finalizado, pero todavía no ha sido recogido por el hilo padre, aunque no puede ser planificado nunca más.

Como sabemos, todos los hilos de un proceso comparten el mismo espacio de direcciones, con lo cual comparten las mismas variables globales (las variables locales no son compartidas, ya que se almacenan en la pila, y cada hilo tiene su propia pila independiente). Como cada hilo tiene acceso a cualquier dirección virtual, un hilo puede leer, escribir o limpiar la pila de otro hilo de su mismo proceso. Normalmente los procesos, que pueden ser de distintos usuarios, pueden ser hostiles entre sí. En cambio en un sistema multihilo, un proceso pertenece a un único usuario, y puede haber creado varios hilos para que cooperen entre sí; se puede realizar un estudio mas a detalle consultando la bibliografía de [Bacon, 1994] y [Tanenbaum, 1993].

Podemos distinguir dos tipos de implementaciones de los hilos: hilos al nivel de *usuario* y los hilos al nivel de *kernel*, de acuerdo con [Bacon, 1994]:

Un hilo al nivel de usuario mantiene todo su estado en el espacio de usuario. Como consecuencia de ello, el hilo no utiliza recursos del kernel para su gestión, y se puede conmutar entre hilos sin cambiar el espacio de direcciones.

Como desventaja, los hilos al nivel de usuario no se pueden ejecutar mientras el kernel está ocupado, por ejemplo, con paginación o E/S, ya que esto necesitaría algún conocimiento y participación por parte del kernel. Los hilos de nivel kernel se pueden ejecutar realmente en paralelo en máquinas SMP (Symmetric Multiprocessors). Además el mismo código binario es válido tanto para una máquina de un procesador como una máquina multiprocesador, ver [Gerlhof et al, 1994].

1.4.6 Ventajas de los hilos.

La principal ventaja de los hilos es el rendimiento. Normalmente el rendimiento es una cantidad percibida y no dada. *Unas aplicaciones se beneficiarán de los hilos mientras que otras no.*

Existe un gran número de ventajas para emplear los hilos de usuario así como los hilos del núcleo, según la bibliografía de [Gerlhof et al, 1994] y [Tanenbaum, 1993]:

Ejecución paralela: Los hilos de nivel kernel se pueden ejecutar realmente en paralelo en máquinas SMP (Symmetric Multiprocessors).

Ejecución de código asíncrono: Ya que existen múltiples hilos de ejecución en un programa construido con hilos, es posible que otro hilo del mismo proceso sea planificado para ejecutar en el caso de que el hilo de ejecución actual se bloquee. Esto incrementa la probabilidad de que la aplicación asociada se ejecute mucho más rápido. Normalmente existen algunas cosas a tener en cuenta:

- En el modelo de hilos de usuario, algunos bloqueos bloquearán todo el proceso y se planificará un proceso diferente.
- En el modelo de hilos de núcleo, es posible que un hilo de otro proceso sea planificado cuando sucede un bloqueo del hilo actual.

En ambos modelos, un hilo puede bloquearse y pueden no existir otros hilos listos para ejecutarse en el proceso, por cualquier tipo de razones.

Así pues, el incremento del rendimiento no siempre está garantizado, aunque al menos, el *uso de los hilos puede incrementar la probabilidad de un mejor rendimiento.*

Recursos compartidos: Los hilos comparten la mayoría de los recursos en un proceso. Comparten el acceso a los ficheros, memoria compartida, y el espacio de direcciones virtual. El empleo de los hilos permite a los programadores conservar los recursos del sistema. El rendimiento puede beneficiarse también porque los recursos compartidos entre hilos necesitan menos gestión que los recursos compartidos entre procesos.

Velocidad de las operaciones con hilos: Los hilos tienen un *menor contexto* que los procesos, así pues, las operaciones con hilos son generalmente más rápidas que las operaciones similares con procesos. En especial, la creación, finalización y cambio de contexto de un hilo, son operaciones más rápidas con los hilos que con los procesos. Como ejemplo, en los procesadores de propósito general (SPARC, MIPS, ALPHA, HP-PA, x86) el cambio de contexto entre hilos de un mismo proceso lleva del orden de 50us. mientras que el cambio de contexto entre hilos de distintos procesos lleva del orden de 100us. Estos tiempos son muy inferiores al tiempo de cambio de contexto completo entre procesos. Así en Solaris la creación de un proceso es unas 30 veces más lento que la creación de un hilo, las variables de sincronización son unas 10 veces más lentas y el cambio de contexto unas 5 veces más lento, Para mas detalles, se recomienda consultar la bibliografía de [Hwang et al, 1988], [Bacon, 1994].

Tiempo de respuesta: Si es posible separar operaciones en un programa, los hilos se pueden emplear para mejorar los tiempos de respuesta de la aplicación. Por ejemplo, supongamos que estamos usando una utilidad de correo electrónico. En una versión de un solo hilo, mientras almacenamos un mensaje podemos apreciar algún retraso antes de que la interfaz de usuario sea refrescada. Esto es porque el programa está primero haciendo una operación de E/S para almacenar el mensaje y después refresca la pantalla. Estas operaciones las realiza de forma secuencial. Normalmente, si esta aplicación fuese una versión programada con varios hilos, un hilo podría gestionar la E/S mientras otro hilo gestiona la interfaz de usuario. Estas operaciones pueden funcionar en paralelo, con la consiguiente ganancia en el tiempo de respuesta, según se resume en [Hwang et al, 1988].

Existe un estándar (POSIX 1003.1c) lo que permite hacer a las aplicaciones portables entre distintas plataformas. El mismo código fuente es válido para distintas plataformas, ver [Bacon, 1994].

En la sección que sigue se mencionan algunas aplicaciones de los hilos, proporcionando así una vista más clara sobre lo que se podría realizar, usando la técnica del multihilado.

1.4.7 Aplicaciones de los hilos.

Algunos programas presentan una estructura que puede hacerles especialmente adecuados para entornos multihilo. Normalmente estos casos involucran operaciones que pueden ser solapadas. La utilización de múltiples hilos, puede conseguir un grado de paralelismo que incremente el rendimiento de un programa, e incluso hacer más fácil la escritura de su código. Algunos ejemplos, como se describen en [Gerlhof et al, 1994] y [Bacon, 1994], son:

- Utilización de los hilos para expresar *algoritmos inherentemente paralelos*. Se pueden obtener aumentos de rendimiento debido al hecho de que los hilos funcionan muy bien en sistemas multiprocesadores. Los hilos permiten expresar el paralelismo de alto nivel a través de un lenguaje de programación.

- Utilización de los hilos para *solapar operaciones de E/S lentas* con otras operaciones en un programa. Esto permite obtener un aumento del rendimiento, permitiendo bloquear a un simple hilo que espera a que se complete una operación de E/S, mientras otros hilos del proceso continúan su ejecución, evitando el bloqueo entero del proceso.

- Utilización de los *hilos en interfaces de usuario*. Se pueden obtener aumentos de rendimiento empleando un hilo para interactuar con un usuario, mientras se pasan las peticiones a otros hilos para su ejecución.

- Utilización de los *hilos en servidores*. Los servidores pueden utilizar las ventajas del multihilo, creando un hilo gestor diferente para cada petición entrante de un cliente.

- Utilización de los *hilos en procesos pipeline*: Se puede implementar cada etapa de una tubería o pipeline mediante un hilo separado dentro del mismo proceso.

- Utilización de los *hilos en el diseño de un kernel* multihilo de sistema operativo distribuido que distribuya diferentes tareas entre los hilos.

- Utilización de los *hilos como soporte de aplicaciones de tiempo real*, acelerando los tiempos de respuesta para los eventos asíncronos a través de la gestión de señales.

Algunas de estas técnicas pueden ser implementadas usando múltiples procesos. Los hilos son sin embargo son más interesantes como solución porque:

- Los procesos tienen un alto coste de creación.
- Los procesos requieren más memoria.
- Los procesos tienen un alto coste de sincronización.
- Compartir memoria entre procesos es más complicado, aunque compartir memoria entre hilos tiene sus problemas.

Los hilos se crearon para permitir combinar el *paralelismo con la ejecución secuencial y el bloqueo de llamadas al sistema*. Las llamadas al sistema con bloqueo facilitan la programación, y el paralelismo obtenido mejora el rendimiento.

1.4.8 Problemas potenciales de los hilos.

Los problemas más comunes en cuanto a la gestión de hilos se refieren son los siguientes, de acuerdo con [Gerlhof et al, 1994]:

- Complejidad: Aunque es cierto que la programación con hilos puede proporcionar una forma más natural de resolver los problemas, existen otras cosas que se deben tener en cuenta. La programación con hilos es más difícil que la clásica programación secuencial. Por ejemplo, cuando se mantiene, depura u optimiza una aplicación con hilos, se deben tener en cuenta la existencia de múltiples hilos de código en ejecución. Por ello generalmente es más fácil depurar y optimizar un programa de *hilo simple*.

- Sincronización: Se debe coordinar el acceso a los datos compartidos mediante bloqueos. Olvidar un bloqueo puede producir la corrupción de los datos.

- Posibilidad de deadlock: Las dependencias circulares en los bloqueos pueden originar deadlock.

- Depuración difícil, debido a las dependencias entre los datos y las dependencias de tiempo. Depurar en un sistema multiprocesador es más complicado, más costoso, y los errores no siempre son reproducibles. Una biblioteca de hilos puede ser útil para detectar y analizar errores en un entorno uniprocesador antes de ser probada en un sistema multiprocesador. Existen dos tipos de errores, los errores serie que pueden ocurrir en un entorno uniprocesador, y los errores paralelos que son inherentes a una ejecución paralela, y son difíciles de detectar.

- Disponibilidad de herramientas: Para favorecer el desarrollo de aplicaciones multihilo, la industria necesitará crear herramientas de depuración y optimización más refinadas. Sin embargo, la tecnología de los depuradores y optimizadores es relativamente joven lo que a corto plazo supone un problema para los programadores.

Estos son algunos de los problemas potenciales de mayor relevancia, a los que podríamos enfrentarnos al tratar de implementar la técnica del multihilado. Por ahora pospondremos nuestro análisis de conceptos sobre los hilos de ejecución, pues en el capitulo cuarto, se mencionaran las características de mayor relevancia del multihilado (multithreading), en la arquitectura que se propone de sistemas distribuidos. Sin embargo si se desea hacer un estudio más profundo acerca de los problemas del multihilado, es recomendable revisar la bibliografía de [Gerlhof et al, 1994] y [Asche, 1996].

1.5 Perfil de los sistemas distribuidos y paralelos.

Retomando nuestro estudió, hasta este momento sabemos que los sistemas distribuidos deben cumplir con *cuatro características esenciales*:

- "Funcionar bajo una Arquitectura o plataforma Abierta" (El hecho de que la plataforma este compuesta por un conjunto de Plataformas o redes Heterogéneas, debe ser transparente para los procesos o sistemas que se ejecuten bajo esta).

- Las Aplicaciones que se ejecutan bajo la plataforma pueden tener sus Bases de Datos o de Conocimiento Distribuidas, y esto ser transparente para los usuarios de dichas aplicaciones.

- Que el procesamiento que implica la ejecución de tareas o aplicaciones sobre la plataforma, se haga de forma, distribuida y paralela, a lo largo de los componentes, que estén bajo la plataforma, y esto también debe de ser transparente.

- Y lo más importante, que se ejecute bajo un sistema operativo o middleware[1] el cual se encargue de *"coordinar"* la ejecución distribuida de los procesos o aplicaciones para lo cual se debe valer de procesos cooperantes que se encarguen de partir la ejecución de las tareas, en tareas independientes (paralelizar) y posteriormente valerse de otros procesos cooperantes (agentes) para que distribuyan el procesamiento de dichas tareas a lo largo de la plataforma, y que recojan los resultados y si alguno de los que ejecutan parte de la tarea no la cumplen, entonces asignar de nuevo estas tareas hasta que se complete la ejecución del sistema.

La ventaja que trae esto, es que no se repitan taréas (dos personas, se dedique a hacer la misma cosa, sin saber) y que el almacenamiento de los datos que componen el sistema no se haga tan redundante y difícil, por consiguiente de actualizar, es como Imaginarse que todas las computadoras fueran iguales y solo conectadas no mas que por un medio físico y el hecho de que parte de las taréas del sistema, se ejecuten en distintos sitios sea independiente tanto para quien programa el sistema como para quien lo usa. Bueno en vista de que existen muchos intereses de por medio (muchos no desean compartir sus cosas) el redundar datos, en una base de datos (deben existir niveles o protocolos de seguridad) distribuida, es casi inevitable pero si existe alguien quien este coordinando esto se puede reducir el nivel de redundancia al máximo hacer como una especie de normalización, en una base de datos relacional.

Ahora retornemos a la realidad actualmente la plataforma existente (Internet) esta formada por un conjunto de redes heterogéneas, que no todas se hablan el mismo lenguaje de redes y no existen Sistemas Operativos Distribuidos (existen algunas aproximaciónes), sobre Internet por lo cual ya no se puede lograr transparencia para quien construya aplicaciones sobre esta plataforma por lo cual quien intente hacerlo *tiene que preocuparse* de definir protocolos estándar para lograr transparencia en la arquitectura.

Uno de las Grandes ideas, para evitar esto, fue definir el famoso modelo de Sistemas Abiertos ISO/OSI, el cual habla de definir 7 capas a las cuales debe ajustarse toda plataforma y en la cual existan protocolos estándar para la comunicación o servicios entre las capas, lo cual es difícil de lograr ya que cuando las cosas son comerciales, todos quieren ganar de acuerdo como se menciona en [Tanenbaum, 1993] y [Martin et al, 1997].

[1] El concepto de Middleware, se abordara en el capitulo 4.

Bueno pues ya no le queda mas remedio a quien implemente las aplicaciones sobre la plataforma que tratar de definir, protocolos estándar agregar una capa estándar, o definir protocolos de mas alto nivel sobre el conjunto de protocolos heterogéneos ya existentes para tratar de simular (o disfrazar) lo que quería establecer el modelo ISO/OSI, además de preocuparse por combinar el modelo cliente-servidor existente con procesos cooperantes especializados en ciertos tipos de tareas que ayuden a repartir el procesamiento y a realizar las labores de distribución y paralelización de las taréas, pero es la aplicación quien debe coordinar estos procesos cooperantes y no existe alguien (que debería ser el sistema operativo) quien vele por el cabal cumplimiento de todas las taréas y la optimización de las mismas, es aquí donde surge la necesidad de los nuevos modelos de computación basados en objetos distribuidos y el middleware, como veremos en el capitulo 3, y con base en esto podemos justificar la necesidad de contar con arquitecturas que den *soporte* y satisfagan al tipo de aplicaciones que se exigen hoy en día donde hay que contender con información abundante, imprecisa, incompleta; que además permita interactuar con ambientes en los que a menudo se compite por recursos y, con la suficiente portabilidad para operar sobre diversas plataformas de cómputo.

Capitulo 2

ASPECTOS DE DESARROLLO DE SOFTWARE BASADO EN COMPONENTES.

Nuestra capacidad de creación de software de computación no está a la par con la evolución del hardware. Se necesita una revolución industrial en el software.
Es muy probable que esta revolución provenga de las técnicas orientadas a objetos, combinadas con herramientas CASE, generadores de código, programación visual, la meta es maximizar la reutilización, así como construir, distribuir, almacenar e interoperar objetos complejos [Bruun et al, 1996].

En éste capitulo se presenta la infraestructura necesaria para la actividad del reuso y el marco de referencia para el análisis de los sistemas basados en componentes. Este último, está constituido por un modelo general de composición y por un conjunto de problemas relativos a la reusabilidad, a demás se mencionan las principales características que ofrece la tecnología de componentes como soporte a los sistemas de gran escala y en particular a los distribuidos. Finalmente, se presentan las conclusiones acerca de la relación que existe entre el paradigma orientado a objetos, el reuso de software y que los *componentes* proveen la mejor manera de construir aplicaciones.

2.1 La orientación a objetos.

Dentro de la orientación a objetos, hay muchos conceptos fundamentales que contribuyen a modelar un sistema. Que parte desde gentes que resguardan alrededor del mundo en términos de objetos, modelos de negocios y software basados en objetos del mundo, los cuales reflejan la realidad mas naturalmente.

Es común encontrar en la literatura frases como: "entre las muchas razones por que la gente adopta los métodos orientados a objetos está la reputación para mejorar el reuso de software". Sin embargo, no existe mucha documentación que explique y justifique dicha reputación. Por otro lado, el paradigma orientado a objetos, no ha satisfecho las expectativas en cuanto a la *reusabilidad* y se puede decir que parte de la falta de éxito de la orientación a objetos no se debe a defectos en la tecnología, si no en el amplio malentendido de la manera como la tecnología soporta la reusabilidad, por éstas razones necesitamos verificar la vinculación que tiene la *orientación a objetos con el reuso del software*.

Una de las preocupaciones actuales más urgentes de la industria de la computación es la de crear software y sistemas más pronto y de más bajo costo. Para hacer un buen uso del poder cada vez mayor de las computadoras, se necesita un software de mayor complejidad. Aparte de más complejo, también es necesario que dicho software sea más confiable.

Existen dos tecnologías de objetos básicas, de acuerdo con [Koulinitch et al, 1997]:

1. Tecnología Orientada a Objetos (Object Oriented Technology, Tecnología Orientada a objetos).
   Que incluye:
   - Programación orientada a objetos.
   - Análisis, diseño y modelado orientado a objetos.

2. Tecnología Basada en Objetos (Object Based Technology, OBT), usa mecanismos cliente/servidor, incluyendo:
   - Programación basada en componentes
   - Análisis, diseño y modelado de componentes (objetos) distribuidos
   - Cómputo distribuido y concurrente.

La diferencia entre tecnología orientada a objetos (OO) y tecnología basada en objetos (OBT) es como la diferencia entre un producto y la especificación técnica para ese producto. Esta especificación contiene información que es útil para crearla, pero que no lo es para su uso en el nivel de aplicación.

A hora, podemos definir a un **componente** como:

Un encapsulamiento físico de uno o más servicios a los que se tiene acceso a través de las interfaces. Suelen ser archivos binarios, colecciones de desencadenantes de bases de datos o procedimientos almacenados, o cualquiera de varias entidades físicas de software. Un componente se define por los servicios que proporciona y por la forma en que interactúa con otros componentes. La construcción interna y la implementación no se muestran al mundo exterior y, en este sentido, los componentes comparten las características modulares y de encapsulamiento de los objetos (Orientación a objetos).
Externamente, todo lo que se conoce de un componente es su interfaz. Un componente consumidor puede solicitar los servicios de un componente proveedor, independientemente de cómo y dónde se implemente el componente proveedor; otras definiciones semejantes, las podemos encontrar en [Koulinitch et al, 1997] y [Watson, 1998].

Una de las más interesantes características de los componentes es que pueden ser combinados con otros componentes, a esto se le conoce como "conecta y usa" o plug and play; debido a que como sabemos, un componente es una pieza de software reusable y autocontenida que es independiente de cualquier aplicación.

Las ventajas de los componentes son las siguientes; puede consultar a [Watson, 1998], donde se describen con mas detalle estas características:

- Interfaces estándar de programación basada en un estándar y robusto sistema de objetos binarios.
- Corporación transparente con los sistemas de servicios distribuidos basados en objetos.
- Independencia: El componente ha de disponer de autogobierno, ser capaz de buscar los servicios que necesita y de decidir libremente qué solución escoger entre las posibles.
- Adaptabilidad: El componente ha de ser capaz de acomodarse a distintas interfaces y protocolos, y a cambios en sus requerimientos. Debe poder componerse con otros y ser flexible, versátil, personalizable y extensible.
- Mantenimiento. La lógica corporativa desplegada en los servidores centralizados, en lugar de en escritorios de usuarios dispersos, hace que mantenerse a la altura de los cambios sea más fácil y reduce el tiempo de inversión en soluciones.
- Incrementa la escalabilidad. La capacidad de los componentes de tratar las diferentes cargas de trabajo de la forma más apropiada posible en diferentes contextos. Un componente que rinde bien en ciertas condiciones

2.2 Relación entre el paradigma Orientado a objetos y el reuso de Software.

A continuación se ilustrara la relación que tiene el modelo orientado a objetos con el reuso de software a través del estudio de sus conceptos y mecanismos. El marco de referencia para éste desarrollo tiene su centro en la interconexión de componentes y está plasmado en un modelo de composición que unifica la exposición. Las implicaciones de la herencia, dan lugar al mecanismo del polimorfismo y que ofrece aportaciones particulares al reuso de componentes; Estos mecanismos se describen con mas detalle en [Tsichritzis et al, 1992] y [Brockschmidt, 1993].

2.2.1 Herencia

En el contexto de modelado de programas basado en *tipos de datos*, se estructuran las abstracciones del problema en un *sistema de tipos*, donde un tipo determina las características estructurales y de comportamiento de los objetos que le pertenecen, regulando así su uso, según [Tsichritzis et al, 1992]. Es notorio que diferentes tipos presenten características estructurales y de comportamiento comunes, lo que da pie a una *relación de subtipo*. Así, se pueden definir nuevos tipos con base en otros, señalando que mantienen una relación de subtipo y especificando cómo difieren de ellos.

Desde el punto de vista de la reusabilidad, la herencia aborda:

a) El problema de similitud entre implementaciones, pues el código común a varias clases sé factoriza en una clase superior de la cual heredan todas las demás, así en lugar de existir repetición de código, cada subclase reutiliza las definiciones e implementaciones de la superclase.

b) El problema de refinamiento, pues si se quiere extender y(o) modificar los servicios de un componente, se crea una nueva subclase y se redefinen o agregan los servicios requeridos, con la particularidad de que esta extensión es intercambiable con el componente original en el código cliente.

De esta manera, tenemos que la herencia permite adaptar los componentes orientados a objetos a diferentes contextos sin modificar la implementación del componente original. Por lo tanto, se agrega flexibilidad permitiendo adaptar la conducta del componente reusable sin afectar a sus clientes, como se describe en [Tsichritzis et al, 1992] y [Brockschmidt, 1993].

2.2.2 Polimorfismo.

Como describe [Brockschmidt, 1993], el polimorfismo permite que el tipo de dato del objeto *servidor* pueda variar dinámicamente, por lo que el requerimiento del servicio podrá ser respondido por diferentes implementaciones. Esto representa una gran flexibilidad en la interconexión de componentes.

Como se menciona en [Tsichritzis et al, 1992], el polimorfismo contempla dos aspectos:

1. Polimorfismo para la variable: una variable se declara estáticamente de un cierto tipo, indicando que puede adoptar solamente los valores (objetos) de ese tipo (clase), pero gracias a la herencia, también puede adoptar los valores de los subtipos de su clase (sus subclases).

2. Polimorfismo para la invocación del procedimiento: como la variable puede dinámicamente adoptar valores de cualquiera de sus subclases, la invocación de un procedimiento sobre esta variable (objeto) no puede ser asociada estáticamente a una determinada implementación, para ello se utiliza el mecanismo de *enlace dinámico*, que determina el tipo dinámico de la variable y asocia la implementación correcta en tiempo de ejecución.

De esta manera, vemos que el polimorfismo se logra en los lenguajes orientados a objetos por la combinación de *herencia*, *enlace dinámico*, y *la invocación de servicios sobre variables*, que dinámicamente adoptan valores de sus subtipos.

Pero, ¿qué tiene que ver el polimorfismo con la reusabilidad?

En primer lugar, aborda el problema de independencia de representación de servicios que se le conoce como *invocación dinámica no uniforme*, pues permite al reutilizador diseñar algoritmos generales que sirvan para componentes reusables con diferentes características de implementación y que coincidan en la especificación de un mismo comportamiento a través de una superclase común. De esta manera, si se reutilizan nuevas implementaciones de una abstracción (del depósito o de construcción propia), no será necesario modificar el código de los algoritmos que utilizan dicha abstracción.
Las posibilidades de reuso que proporciona el polimorfismo, dependen en gran medida de cómo se estructura la jerarquía de clases. En particular, el uso de este mecanismo se ve obstaculizado si en la jerarquía se tienen clases por conveniencia para la implementación y no modelan las relaciones naturales de subtipo entre las entidades del problema.

Un criterio que beneficia al reuso polimórfico es tratar de construir la jerarquía de clases con el máximo número de niveles posibles, manteniendo la representación de las relaciones de subtipo del problema. De esta manera, se tendrá tipos abstractos más generales que proporcionan la funcionalidad suficiente en cada nivel, propiciando así el reuso polimórfico en un mayor número de niveles de la jerarquía; esta es una de las estrategias que en común se menciona en [Nierstrasz et al, 1992] y [Chandy et al, 1996].

2.3 El modelo de composición del paradigma Orientado a objetos.

Sabemos que un objeto está formado por una interfaz y por una implementación. Entenderemos por interfaz de un objeto como *la especificación del conjunto de servicios que ofrece más la especificación del contexto de aplicación de estos servicios* [Bruun et al, 1996] y [Nierstrasz, 1993]. Cada servicio que ofrece el objeto tiene una especificación idéntica a la de un procedimiento.

En el modelo de composición orientado a objetos, existe una clara diferencia sintáctica entre el componente y los servicios que proporciona. Es decir, para invocar un servicio, se debe especificar *el identificador del objeto más el identificador del servicio requerido*. De tal manera, vemos que este modelo de composición agrega un elemento sintáctico a la composición de componentes. Por supuesto, este agregado sintáctico no representa el beneficio para el modelo de composición, sólo es una base sobre la cual se estructuran mecanismos estáticos y dinámicos que amplían las posibilidades de composición.

De esta manera, si queremos analizar la relación que existe entre el paradigma orientado a objetos y el reuso de software, se considera que un marco de referencia adecuado debe estar ligado a la noción de modelo de composición.

Específicamente el marco de referencia, que se presenta en [Nierstrasz, 1993], esta formado por una generalización del modelo de composición y por un conjunto de problemas relativos a la reusabilidad que debe abordar el modelo para soportar apropiadamente la interconexión de componentes.

2.3.1 Generalización del modelo de composición.

Independientemente del paradigma de programación empleado podemos visualizar un modelo de composición general con tres actores principales, de acuerdo con [Bruun et al, 1996] y [Nierstrasz, 1993]:

- **Servidor:** proveedor de servicios (es el componente reusable).
- **Interfaz:** información acerca de los servicios que proporciona el servidor y del contexto en que pueden ser solicitados.
- **Cliente:** consumir los servicios.

El elemento central en este modelo es la *interfaz*, pues implica una intermediación con una implementación (*servidor*) y su uso por otra entidad *(cliente*).

De esta manera, tenemos una triple relación *servidor-interfaz-cliente* que conforma un modelo de composición donde el cliente le solicita servicios al servidor a través de una interfaz describe la abstracción representada por el componente (aislando la implementación) y especifica la forma de cómo debe utilizarse. De esta manera, la interfaz especifica un *protocolo de conexión* para el componente y el cliente debe ajustarse a este protocolo para poder utilizar los servicios disponibles. Con base en este modelo general, podemos establecer unos principios generales que faciliten y propicien el reuso de componentes, según se describe en [Nierstrasz, 1993]:

- **Independencia:** debe mantenerse desacoplados lo más posible a los clientes de los servidores. Básicamente debemos asegurar: independizar a los clientes de la representación de datos en los servidores, minimizar el efecto que tengan sobre ellos los cambios de implementación en los servidores e independizar a los servidores de las características especiales de futuros clientes.

- **Genericidad:** debe capturar la máxima similitud de una abstracción cuando se utiliza en diferentes contextos. Básicamente debemos asegurar: evitar la repetición de código que se diferencie sólo en los tipos de datos que utiliza y abstraer las similitudes entre servidores para promover un acceso uniforme.

- **Refinamiento:** es la posibilidad de adaptar (modificar o extender) el componente a cambios en los requerimientos, sin perturbar a los clientes de dicho componente. En la medida en que se satisfaga el principio de independencia, habrá más posibilidades de lograr el refinamiento.

Los principios descritos son importantes porque definen requerimientos generales para la construcción y uso de componentes reusables.

2.3.2 Problemas que aborda un modelo de composición.

Un modelo de composición debe resolver algunos problemas básicos para lograr el reuso de código. No es de extrañar que la mayoría de estos problemas se puedan plantear en términos de demandas para la construcción de programas en general, pués reutilizar componentes es básicamente construir programas. Sin embargo, lo que realmente nos interesa es mostrar las implicaciones que tienen para el reuso del software, los problemas que se mencionaran enseguida, fueron originalmente propuestos en la bibliografía de [Nierstrasz, 1993] y [Chandy et al, 1996].

El problema de la variación de tipos.

Se presenta cuando hay estructuras de datos contenedores iguales y (o) procedimientos que tienen el mismo comportamiento algorítmico y difieren sólo en los tipos de datos que utilizan. Es un problema para el cliente que debe solicitar la construcción de un nuevo componente para un nuevo tipo de dato y es un obstáculo para los implementadores del componente que deben repetir el mismo código. También es un problema para el mantenimiento, pues los componentes mantienen una relación de implementación a través de su código común y si se decide modificar una implementación. Se deberá modificar la implementación de todos los componentes relacionados.

Problema de independencia en representación de datos.

Los sistemas evolucionán a medida que se imponen nuevas demandas. Por lo general, estas demandas se pueden clasificar en cambios de representación de la información y en cambios de la funcionalidad del sistema.

El modelo de componente debe reducir la incidencia de éstos cambios lo más que pueda, pues de lo contrario sus beneficios se limitarán al desarrollo inicial del sistema, perdiéndose en la evolución del mismo. En cuanto a los cambios de representación de la información el modelo de composición debe aislar al cliente de los cambios en los formatos de almacenamiento.

Para lograr esto, debe evitar que los clientes accedan directamente a las representaciones de los datos, en lugar de ello, debe proporcionar servicios que lo hagan. En cuanto a los cambios en la funcionalidad del sistema se debe desplazar el centro del reuso basado en funcionalidades, porque es precisamente lo que cambia y afecta a los clientes, de acuerdo con [Nierstrasz, 1993].

Problema de independencia en representación de servicios.

Este problema surge cuando se tiene diferentes implementaciones para un mismo servicio. Afecta fundamentalmente al cliente que en lugar de hacer la invocación de un solo servicio, tiene que hacer una invocación diferente para cada implementación; por ésta razón también se le conoce como *invocación no uniforme*. Este problema de independencia de representación en doce el cliente tiene que saber a qué implementación concreta se está refiriendo en la invocación. Se manifiesta en situaciones de invocación estática y dinámica, como es descrito en [Tsichritzis et al, 1992]:

- **Invocación estática no uniforme**. El problema surge cuando los mecanismos de programación disponibles obligan a que cada implementación del servicio tenga un nombre diferente para poder distinguirlas. El cliente tiene que hacer invocaciones con diferentes nombres para solicitar un mismo servicio y el servidor tiene que proporcionar diferentes interfaces para un solo servicio, haciendo más complejo el modelo de composición.

- **Invocación dinámica no uniforme.** El problema surge cuando se tiene diferentes representaciones de una abstracción de datos y a cada una se le asocia una implementación distinta de un servicio que proporciona la misma funcionalidad abstracta. El panorama se completa cuando dinámicamente hay que detectar el tipo de representación concreta de la abstracción e invocar el servicio correcto asociado.

Por otro lado, si se utiliza una estrategia de *discriminación dinámica* basada en estructuras condicionales, el código se verá poblado de estas estructuras para cada servicio. Peor aún, una nueva representación de una abstracción provocará la modificación y recompilación del código cliente que utiliza las estructuras condicionales. Para solucionar este problema se necesita un mecanismo de programación que determine automáticamente el servicio correcto que debe invocarse en función del tipo de representación concreta de la abstracción que se está utilizando.

Problema de la similitud entre implementaciones.

El principio de la reusabilidad de código es utilizar códigos que ya están escritos. Cuando se interconecta un componente reusable a un nuevo programa, se utiliza la interfaz del componente, en lugar de repetir todo el código. Así, la noción del reuso implica evitar la repetición de código y no solo en el uso, también debe evitarse en la construcción de componentes reusables.

Esto se logra factorizando en un sólo lugar las similitudes entre las diferentes implementaciones. Por otra parte, a través de un código común, diferentes implementaciones mantienen una integridad conceptual, así, si no se puede expresar éste código común y se decide modificar una estrategia de implementación, se tendrá que modificar las implementaciones de todos los componentes que siguen tal estrategia, dificultando la evolución y mantenimiento del depósito de componentes, de acuerdo con [Bruun et al, 1996] y [Chandy et al, 1996].

El problema del refinamiento.

Lo ideal es reutilizar un componente sin modificarlo, sin embargo, no siempre el componente reusable satisface los requerimientos solicitados en su totalidad, puede ser frecuente que sólo una parte de él sea útil a las demandas impuestas. En este contexto, es muy importante que el modelo de composición proporcione un mecanismo que permita modificar y (o) extender los servicios del componente, de manera que se pueda adaptar a los nuevos requerimientos.

Esta adaptación del componente a un nuevo contexto, debe asegurar que las modificaciones no afecten a sus clientes. En términos más concretos, no puede modificar la interfaz del componente, sintáctica ni semánticamente. Pero no sólo eso, modificar la implementación de los servicios también puede afectar a los clientes en sus suposiciones acerca del desempeño, por lo que si se quiere hacer una modificación deberá construirse un componente nuevo.

De aquí vemos que el asunto se complica aún más porque al refinar un componente se está creando una nueva implementación del mismo, y sería muy conveniente que éste nuevo componente lo puedan utilizar los clientes del anterior sin modificar su código de anfitrión. Para lograr esto se requiere construir un nuevo componente, con base en el otro y proporcionar mecanismos para intercambiarlos según las necesidades, a esto se le conoce como ***interoperabilidad***, según se describe en [Tsichritzis et al, 1992] y [Chandy et al, 1996].

El problema de la compatibilidad de interfaces.

Un componente modela una abstracción y está asociado a una implementación determinada. El depósito de componentes (Object Factory), responsables puede tener diferentes implementaciones para una misma abstracción, por ejemplo, una *Pila* implementada con una Lista o un arreglo. De igual forma, se puede implementar el servicio de búsqueda con un método secuencial o binario.

El cliente debe poder acceder de la manera uniforme a cualquiera de las operaciones de búsqueda o a cualquiera de las implementaciones de la estructura. Es decir, debe poder tener acceso uniforme a diferentes implementaciones y con ello, poder intercambiar los servidores sin modificar su código anfitrión. Para lograr esto es necesario que las implementaciones presenten un interfaz compatible, en el sentido de que en ambas se puedan solicitar los mismos servicios de la misma manera, ver detalles en [Nierstrasz et al, 1992] y [Nierstrasz, 1993].

Otros problemas.

Los problemas anteriormente mencionados, no son los únicos que debe abordar un modelo de composición, a continuación se mencionan otros más [Tsichritzis et al, 1993]:

- **El problema de la granularidad.** Aunque el concepto de clase/objeto logra un mayor nivel de granularidad de la abstracción que se reutiliza con relación al procedimiento, es todavía insuficiente para lograr grandes beneficios. En la construcción de sistemas grandes hallamos repetición de estructuras complejas que representan arquitecturas completas de subsistemas y no encontramos mecanismos de programación que abarquen la granularidad de estos componentes y las relaciones que tienen con otros componentes parecidos. Se cree [Tsichritzis et al, 1993], que una propuesta que pretenda atacar este problema debe proveer mecanismos concretos en lenguajes y ambientes de programación que mapeen estas abstracciones a programas ejecutables.

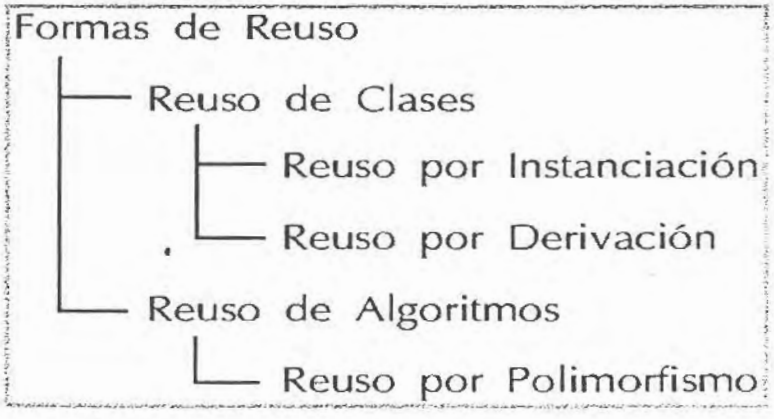
- **El problema de la interconexión multiparadigma.** Cuando hablamos de un depósito de componentes (object factory) reutilizables, podemos tener componentes concebidos bajo distintos paradigmas: bibliotecas de procedimientos, bibliotecas de clases, conjuntos de reglas lógicas, componentes de representación de conocimiento, componentes de manejo de procesos concurrentes, paralelos y distribuidos, etcétera.

Es muy probable que para un desarrollo necesitemos interconectar estos componentes, sin embargo, no existen *mecanismos estándares*[1] de programación que permitan interconectar estos componentes en un nivel en el que se abstraigan los detalles de implementación y la sintaxis particular de cada uno de ellos.

- **El problema de la integridad semántica.** La interconexión de componentes debe asegurar que no se modifiquen las propiedades individuales de los componentes y que se satisfagan las propiedades requeridas en la interconexión. En este campo, ha habido avances importantes con el uso de especificaciones formales, lamentablemente, éstas no forman parte del Paradigma Orientado a objetos, para mas detalles ver la bibliografía de [Chandy et al, 1996] y [Tsichritzis et al, 1992].

2.4 Formas de Reuso Orientado a objetos.

Podemos clasificar en tres categorías de reuso, las diferentes posibilidades de interconexión de componentes que proporciona el paradigma: por *instanciación*, por *derivación* y por *polimorfismo*, según nos se describe en [Bruun et al, 1996]:



[1] Actualmente, el único mecanismo **mas o menos** estándar, para la interconexión de componentes, es el lenguaje de definición de interfaz (IDL), sin embargo las implementaciones que ofrecen dos de las mas importantes organismos en objetos distribuidos (CORBA/DCOM), aun distan mucho de llegar a unificar un estándar.

Reuso por Instanciación.

Cada vez que se genera una instancia de una clase se está reutilizando la representación de una abstracción y la implementación de los servicios que proporciona; así, el programador puede invocar estos servicios a través de la interfaz del objeto, sin preocuparse de cómo están implementados.

Esta es la forma de reuso más común en el paradigma y lo vemos plasmado en la construcción de un programa orientado a objetos cada vez que se crean objetos:

- Para construir nuevos objetos. Se crean nuevas abstracciones con base en las ya existentes, declarando los objetos como atributos de la nueva abstracción (a esto lo llamamos reuso por composición)
- Para su uso local en la implementación de los servicios de un objeto
- Para su uso como parámetros actuales en la invocación de un servicio

Este tipo de reuso es adecuado cuando el componente se ajusta completamente a los requerimientos. Está basado en la propiedad de encapsulación asociada a la abstracción de datos, pues la interfaz esconde la implementación que se está reutilizando y a la vez proporciona el protocolo para su acceso, ver [Nierstrasz , 1993].

Reuso por Derivación.

Cuando la especificación del componente no se ajusta por completo a los requerimientos, se puede heredar de una clase y extender su funcionalidad. Con esto se está reutilizando el componente para crear una nueva abstracción, especificando nuevos comportamientos y(o) implementaciones. En teoría, la derivación se puede hacer sin conocer los detalles de implementación de la superclase, es decir, sólo basándose en su interfaz. Por tanto, esta forma de reuso está basada en los conceptos de herencia y encapsulación.

Algunos autores, como [Chandy et al, 1997] resaltan la particularidad de que la herencia aumenta la complejidad del componente al aumentar la cantidad de métodos disponibles, y que en algunas ocasiones es preferible hacer un reuso por instanciación en lugar de derivación para reducir la complejidad de la clase. Podemos decir que no se esta muy de acuerdo con esta posición, porque independientemente de la complejidad de la clase, la relación de herencia modela una conexión de subclasificación entre entidades del problema, que no es posible modelar con una relación de instanciación.

Por otro lado, esta propuesta conduce a una construcción de la jerarquía de clases que obstaculiza el reuso por polimorfismo, como sabemos es una forma muy poderosa de reuso. En general, podemos decir que si existe confusión entre reutilizar un componente por *instanciación o derivación*, podemos estar ante un mal diseño de la jerarquía de clases, ante un mal modelado o ante un mal uso de los conceptos del paradigma.

Reuso por Polimorfismo.

Podemos decir el reuso por instanciación y por derivación son una forma de reuso de clases. El polimórfico, en cambio, es una forma de *reuso de algoritmos*. Si un método presenta diferentes implementaciones en las subclases de una superclase, se dice que el método es polimórfico respecto a la superclase. En este contexto, un cliente que utilice las subclases puede reutilizar el algoritmo polimórfico, pasando como argumento el objeto de la clase derivada.

El uso de clases abstractas para propiciar el reuso por polimorfismo, separa la implementación de la especificación de la abstracción y, con ello, facilita el reuso por derivación porque no impone restricciones de implementación a la clase heredera. De esta manera, se pueden construir múltiples implementaciones para una misma abstracción y, además, teniendo varias opciones de implementación, se aumenta la posibilidad de reuso por instanciación, para una descripción mas clara, puede consultar a [Tsichritzis et al, 1992].

2.5 Evaluación de las Formas de Reuso.

Indudablemente, el reuso por instanciación es el *más fácil* de todos, el componente se reutiliza tal cual está definido. El reuso por derivación es más difícil, porque requiere definir o redefinir implementaciones, pero la dificultad realmente surge porque éstas están estrechamente ligadas a la representación de los datos y muchas veces es necesario mirar el código de las superclases (violar la encapsulación) para extender su funcionalidad. Por último, el reuso por polimorfismo es el *más difícil* de todos porque implica la construcción de una jerarquía de clases con niveles desacoplados al máximo en términos de abstracción-implementación y que además modele las relaciones de subclasificación del problema.

Como criterio general, podemos decir que el reuso por instanciación está asociado al reuso de clases *concretas*, el reuso por derivación está asociado a clases *abstractas o base* y el reuso por polimorfismo está asociado a una *jerarquía de clases abstractas,* para las abstracciones y clases concretas para diferentes implementaciones. Por otra parte, el tipo de reuso orientado a objetos que se utiliza guarda una estrecha relación con el estado de madurez del depósito de componentes. Cuando se tienen jerarquías de clases reutilizables muy estables, generalmente se emplea el reuso por instanciación; mas detalles se presentan en [Bruun et al, 1996] [Chandy et al, 1996]. El reuso por polimorfismo implica el reuso por derivación, aunque el comportamiento polimórfico no denota necesariamente falta de madurez de los componentes.

Finalmente, una de las cosas que más interesa ilustrar es la *flexibilidad* de interconexión, porque mientras haya mayor diversidad de formas de integración de componentes, mayor será la posibilidad de reuso. Por otro lado, la *genericidad* también es un factor muy fuerte porque permite adaptar el componente a nuevos contextos mediante parámetros. De manera especial tenemos a la herencia que permite la adaptación del componente por refinación y al polimorfismo que permite adaptar los algoritmos polimórficos a nuevas implementaciones sin afectar al cliente, de acuerdo con lo descrito en [Tsichritzis et al, 1992], [Nierstrasz et al, 1992] y [Bruun et al, 1996].

2.6 Características de los componentes en sistemas distribuidos

El objetivo de esta sección es describir, de manera breve e informal, varias propiedades para los componentes en los sistemas distribuidos. Vamos a definir tres fundamentales: *la Autonomía, la Robustez y la Competitividad.* Entendemos por Autonomía la habilidad de un componente de tomar decisiones de forma independiente. La Robustez garantiza la disponibilidad, fiabilidad y seguridad del componente. Y la Competitividad dota al componente de una estrategia general de supervivencia en términos de durabilidad y resistencia.

El problema es que estas tres propiedades son demasiados *generales y complejas* para ser implementadas directamente. La idea clave es "descomponerlas", definiéndolas en función de propiedades más básicas.

Por lo que podemos definir la propiedad de **Autonomía** como la composición de otras tres propiedades, según se describe en [Tsichritzis et al, 1992]:

- *Independencia:* El componente ha de disponer de autogobierno, ser capaz de buscar los servicios que necesita y de decidir libremente qué solución escoger entre las posibles.

- *Adaptabilidad*: El componente ha de ser capaz de acomodarse a distintas interfaces y protocolos, y a cambios en sus requerimientos. Debe poder componerse con otros y ser flexible, versátil, personalizable y extensible.

- *Autoprotección:* El componente debe protegerse ante fallos externos y circunstancias inesperadas, con el objeto de no depender nunca del comportamiento y buen funcionamiento del resto de los componentes.

La **Robustez** se consigue mediante la composición de tres propiedades básicas:

- *Integridad:* El componente ha de ofrecer un comportamiento robusto frente a distintos usos de su interfaz, tanto válidos como inválidos.

- *Control de Acceso*: Todos los accesos hacia (y desde) el componente deben ser autentificados y autorizados convenientemente.

- *Alta Disponibilidad* (High Availability): El componente ha de protegerse ante la ocurrencia de fallos en los procesos o máquinas en donde se ejecuta.

Finalmente, la **Competitividad** puede expresarse en función de las siguientes propiedades:

- *Mayor Esfuerzo/Menor Desgaste:* El componente debe hacer sus mayores esfuerzos para satisfacer las peticiones de sus usuarios (en términos de tiempo de respuesta, funcionalidad y calidad del servicio), mientras que debe minimizar sus costes (o sufrimientos) para realizarlo.

- *Durabilidad:* El componente debe incorporar mecanismos que le permitan actualizarse, renovarse y mejorar con el tiempo.

Nuestro objetivo ahora es definir una arquitectura general capaz de especificarlas de una manera formal e implementarlas, permitiendo añadirlas a los componentes de un sistema, y componerlas entre sí.

2.7 Perfil del desarrollo de software basados en componentes.

El objetivo de la tecnología de componentes es proveer a los usuarios y desarrolladores de software los mismos niveles de interoperabilidad de aplicaciones "conecta y usa", a demás estos componentes *podrían* "vivir" en diferentes redes, sistemas operativos, o aplicaciones. Considerando que un componente no está acotado a un programa en particular, lenguaje de computadora o implementación, por lo que objetos construidos como componentes proveen la mejor manera de construir aplicaciones, por lo que los servicios para alcanzar los objetivos orientados a usuarios, negocios y datos, así como la lógica corporativa para la negociación entre los mismos, pueden estar incluidos en componentes que funcionan de forma autónoma unos de otros e independientemente de su ubicación física en una red.

Esta estrategia de desarrollo de aplicaciones basada en componentes proporciona las siguientes ventajas:

Posibilidad de reutilización. Muchas aplicaciones pueden compartir y reutilizar la funcionalidad encapsulada en componentes.

Flexibilidad. El trabajo se puede distribuir desde el escritorio a servidores de red más potentes, lo cual facilita el control de las necesidades de rendimiento y el ancho de banda de la red.

Capacidad de administración. Los proyectos grandes y complejos se pueden dividir en proyectos de componentes más sencillos y seguros.

Mantenimiento. La lógica corporativa desplegada en los servidores centralizados, en lugar de en escritorios de usuarios dispersos, hace que mantenerse a la altura de los cambios sea más fácil y reduce el tiempo de inversión en soluciones.

Incrementa la escalabilidad. La capacidad de los componentes de tratar las diferentes cargas de trabajo de la forma más apropiada posible en diferentes contextos. Un componente que rinde bien en ciertas condiciones de uso moderado puede desmoronarse cuando se desplaza a una red más ocupada o con más usuarios.

Por otro lado, en lo que respecta al análisis de los mecanismo de programación, que se efectúo en este capitulo, se considera que se hace una aportación particular a las posibilidades de integración de los componentes reusables a nuevos contextos.

Sin embargo, las mayores aportaciones están relacionados al modelo de composición orientado a objetos, al que se hizo referencia. No es de extrañarse que esto suceda por dos razones:

1. Los conceptos del paradigma orientado a objetos son el resultado de una crisis sobre el paradigma procedural.
2. El paradigma orientado a objetos abarca los conceptos del paradigma procedural.

De esta forma, la naturaleza evolutiva y unificadora del paradigma orientado a objetos lo convierte en un candidato adecuado para el *reuso*, utilizando un modelo de composición apoyado en mecanismos y posibilidades de interconexión de componentes. Podemos decir entonces, que existe una cierta afinidad entre el paradigma orientado a objetos y el reuso de software.

Los pocos resultados experimentales - ActiveX, JavaBeans, Open Doc, *[Watson, 1998]*. - muestran coherencia con esta afirmación y nos ayudan a reafirmar nuestra convicción sobre la afinidad que existe entre el paradigma orientado a objetos y el reuso de software. Se advierte que esto no significa que al utilizar el paradigma Orientado a objetos se obtengan automáticamente los beneficios del reuso de componentes.

En primer lugar, hace falta un conocimiento sólido de la manera como los mecanismos del paradigma contribuyen a este objetivo y de cuándo y cómo utilizarlos. Por otro lado, es de vital importancia tener componentes reusables como capital de inversión y la infraestructura para su manejo adecuado: *clasificación, búsqueda, recuperación y documentación*. Adicionalmente, estos elementos técnicos son sólo el medio, no el fin, que permite estructurar metodologías para implementar el aspecto de la reusabilidad en el desarrollo de software.

Capitulo 3

TECNOLOGIA DE SISTEMAS DE CÓMPUTO BASADOS EN OBJETOS DISTRIBUIDOS.

En al actualidad, la idea principal de la tecnología de información es dividir los sistemas de aplicaciones complejas en objetos (o módulos independientes, -componentes) fácilmente desarrollables, comprensibles, diseñados como módulos activos, además estos componentes podrán trabajar en conjunto sólo si han sido diseñados y construidos bajo las interfaces estándar. En éste capitulo se presentan algunas de las primeras y más importantes especificaciones para la actividad del cómputo distribuido basado en objetos, también se presenta la descripción de algunos *productos middleware basados en estos modelos*. Así que en las siguientes secciones de este capitulo se dedicara a describir las características más significativas de los modelos CORBA y COM/DCOM/ActiveX, respectivamente en el orden como se acaba de mencionar. Para el caso de ActiveX, se hizo un resumen sobre esta tecnología de componentes y se presenta como parte del apéndice A.

Nota: El concepto de Middleware, se mencionara en la siguiente sección.

3.1 Objetos Distribuidos.

Como su nombre lo indica, los objetos distribuidos son el resultado de la combinación de dos paradigmas: la noción de distribución de la tecnología cliente/servidor y la noción de orientación a objetos. La idea de los objetos distribuidos consiste en que la división de los sistemas a lo largo de las redes se hará más flexible, más homogénea y más estandarizada, dando como resultado menores dificultades para desarrollar aplicaciones complejas, robustas y adaptativas que en la actualidad se demandan en la industria e investigación aplicada, a demás promete poder utilizar e integrar los sistemas que se usaban anteriormente (legacy applications) [Vinoski, 1997].

Desde un punto de vista simplificado, se necesitan dos cosas para alcanzar las posibilidades que ofrece el cómputo distribuido basado en objetos: Primero, hacer que todo (incluyendo los sistemas que se usaban anteriormente) parezca orientados a objetos, y segundo, poder accesar estos objetos a través de los límites del sistema.

En respuesta a estas necesidades, la industria del software ha generado varias alternativas, entre las cuales las dos más importantes son CORBA del grupo OMG y DCOM de Microsoft.

Considerando que los objetos clásicos proveen magníficas facilidades para reuso de código a través de los tres pilares de la orientación a objetos: *herencia, encapsulamiento y polimorfismo*; como se analizo en el capitulo 2. Sin embargo el problema es que esos objetos clásicos por si mismo no proveen una infraestructura mediante la cual *pueda interactuar el software* creado por diferentes desarrolladores en el mismo espacio de dirección (una misma computadora), mucho menos con en diferentes espacios de dirección (redes o sistemas operativos). Solo el compilador del lenguaje que creo el objeto y su sistema operativo donde "vive", conoce de su existencia. El mundo exterior no conoce el objeto y no tiene manera de accesarlo. La solución es desarrollar aplicaciones distribuidas *basadas en objetos componentes (componentes)*. Mediante este proceso, distintos equipos de proyecto pueden desarrollar componentes independientes, permitiendo una mayor productividad, así como poder desplegarlos físicamente a través de múltiples equipos que integran a la red.

Durante la interacción, los componentes asumen dinámicamente los papeles de clientes/servidores y la interacción de estos componentes distribuidos se unen en un negociador de Solicitudes de Objetos (ORB), este provee los medios para localizar y activar otros objetos en la red, efectuándose de forma transparente. El ORB intercepta las llamadas y es responsable de encontrar un objeto que pueda implementar una solicitud, pasarle los parámetros, invocar sus métodos y regresar los resultados. El cliente no tiene que saber dónde está localizado el objeto servidor. Por lo que el ORB [Vinoski, 1997] y [Yang et al, 1996] es el *middleware de la computación distribuida basada en objetos* que permite la funcionalidad e interoperabilidad de componentes en las grandes redes heterogéneas, tales como Internet y las Intranets corporativas. Por ejemplo, una Intranet corporativa puede estar hecha a base de: Mainframes, estaciones de trabajo UNIX, PC's corriendo bajo cualquiera de las versiones de Windows, OS/2 y Macintosh, pueden poseer dispositivos de diferentes tipos, y trabajar bajo distintas redes y protocolos como: Ethernet, FDDI, ATM, TCP/IP, Netware, etc.

Por lo que obviamente para resolver algunos de estos problemas se necesita de una infraestructura capaz de proveer la invocación de programas remotos y la comunicación con ellos; por lo que la comunicación es transparente de las aplicaciones a través de la red, esto se logra con algún *middleware basado en un ORB*.

3.2 Concepto de Middleware

En forma muy general, **el middleware es el medio que actúa como intermediario entre las aplicaciones y la infraestructura de red**. Como se sitúa en medio de estas dos grandes capas, de ahí el nombre de "Middleware" [Yang et al, 1996] y [Koulinitch et al , 1997].

Aunque no existe una clasificación estándar de middleware, en este rubro se encuentran distintos tipos de software como: monitores de transacciones, ligas estándares entre manejadores de bases de datos y *front-ends* (ODBC, SAG-CLI, IDAPI, etc.), distintos tipos de *gateways* (de bases de datos, de correo electrónico, de seguridad, etc.), herramientas de manipulación de datos (extractores, convertidores de formatos) así como las nuevos ORB's de objetos distribuidos (DCOM y CORBA, entre las principales) y los sistemas basados en mensajes (MOM: Message Oriented Middleware); para mas detalles ver [Vinoski, 1997] y [Yang et al, 1996].

3.2.1 Funcionalidades a Buscar en el Middleware

Podemos distinguir cinco funcionalidades básicas a cubrir por un sistema de middleware orientada a soportar sistemas distribuidos. A continuación se describen brevemente cada una de ellas, según [Vinoski, 1997], [Koulinitch et al, 1997] y [Wang et al, 1997]:

Cobertura: Aquí se revisan las plataformas (hardware y sistema operativo), fuentes de datos, protocolos y front-ends que soporta el middleware a utilizar. Se sugiere que se establezcan prioridades de plataformas y fuentes de datos (por ejemplo: es riguroso el soporte a DB2 en mainframe y Oracle en Unix, es deseable el soporte a DB/Datacom, ADO - ActiveX Data Object) y necesidades futuras.

Facilidad: Sea que vayamos a montar un robusto y complejo sistema, requerimos que el middleware incluya diversos extractores de datos, que a su vez hagan todas las conversiones de formatos necesarias (en tipos de datos, entre ASCII y EBCDIC, representaciones de variables numéricas, etc.) y de lenguajes de acceso (dialectos SQL otras formas de acceso no-SQL). Las facilidades que estos extractores tienen varían grandemente de proveedor a proveedor.

Algunos de ellos requieren - como ya se ha comentado - que alguien termine escribiendo los diversos programas de extracción, otros en cambio permiten que todo se haga a través de una interfaz en la que se definen los distintos tipos de extracciones que se realizarán, y sobre qué plataformas actuarán.

Por otro lado, en el caso de que tengamos que escribir un programa que "converse" con el middleware, debemos revisar las facilidades brindadas por las interfaces correspondientes (APIs).

Apertura: Se puede englobar aquí el soporte de las herramientas a estándares como ODBC, SAG-CLI, DRDA de IBM, CORBA o DCOM, así como la documentación propia del producto y de sus APIs. Es válido considerar el soporte a los estándares propios de nuestra organización, incluyendo la integración con middlewares propietarios como Oracle SQL*Net o Sybase DB-Library.

Seguridad: Con la explosión de Internet y la conectividad TCP/IP, el tema de la seguridad informática ha tomado nuevos aires. Sin embargo, poco hacen la mayoría de los usuarios por garantizar que los productos de middleware que utilizan, posean las características de seguridad necesarias de acuerdo a cada organización. Hablando del middleware para sistemas distribuidos, típicamente requerimos que permita el encriptamiento (cifrado) y autenticación de mensajes, y que se conecte con los manejadores de datos y con los sistemas operativos para realizar la autenticación de usuarios y el control de acceso. Generalmente es conveniente que dichas funcionalidades sean implantadas a través del soporte a estándares como DES, RSA, X.509 o Kerberos / DCE, ver [Martin et al, 1997].

Administración: Este grupo de funcionalidades incluye el monitoreo del rendimiento de todas las piezas del middleware, el obtener información estadística sobre dicho rendimiento y sobre la utilización de recursos, la detección de fallas y el manejo y control de las distintas configuraciones. A este respecto, la mayoría de las herramientas tienen una administración remota muy incipiente, pero la tendencia es muy positiva a este respecto. Ya son varios los fabricantes que han anunciado sus planes futuros y, entre otras cosas, anuncian una integración con plataformas de administración como Tivoli/TME o Microsoft/SMS, otros -por ejemplo- han liberado productos que llevan estadísticas de las consultas que se han realizado, las bases de datos a las que se han conectado y las tablas que han afectado, para que con base en esta información, se puedan hacer recomendaciones sobre la forma de definir o afinar los extractores.

Distintos consumidores de información: Herramientas de desarrollo y 4GLs (Visual-BASIC, SQL-Windows, Power Builder), herramientas de consulta y de toma de decisiones (Access, Focus, Business Objects, DSS-Agent), navegadores de información (browsers y herramientas de minería de datos - data mining).

Distintos sistemas y ambientes operativos: DOS, Windows-NT, OS/2, Unix, VMS, MVS, Linux, etc.

Distintas familias de protocolos de red: SNA, TCP/IP, IPS/SPX, X.25, entre otros [Yang et al, 1996] y [Martin et al, 1997].

Las funcionalidades básicas, mencionadas anteriormente, debieran satisfacerse para concebir un sistema de middleware que llegara a establecerse en el desarrollo de un sistema de cómputo, en nuestro caso un sistema distribuido. En las siguientes secciones, se va ha describir en forma muy general, las principales características y funcionalidades que ofrecen las dos más importantes alternativas que ofrece la industria del software, en lo que se refiere a tecnología de objetos distribuidos y estos son CORBA y COM/DCOM. En cuanto a productos se refiere CORBA, tiene varias firmas, que apoyan sus especificaciones, las más relevantes ala fecha, en que se desarrollaba este trabajo, son: Visigenic, IONA, BEA and NCR. Por su lado Microsoft COM/DCOM, ha hecho alianzas con Software Agents Corporation, quien ha transportado la tecnología de componentes a otras plataformas de sistemas operativos como Digital UNIX, MVS y Mac; con excelentes resultados.

3.3 Arquitectura común de Negociación de Solicitudes de Objetos (CORBA).

El OMG (Object Managment Group, - Grupo de Gestión de Objetos) fue formado en 1989 para desarrollar, adoptar y promover estándares (especificaciones) para el desarrollo de aplicaciones en ambientes distribuidos heterogéneos. Su principal aportación es el OMA (Object Managment Architecture). EL OMA está formado de varios componentes, de los cuales CORBA (Common Object Request Broker Architecture) es su principal componente, también llamado ORB (Negociador de Solicitudes de Objetos) [Yang et al, 1996].

La Arquitectura común de Negociación de Solicitudes de Objetos(CORBA) lo que pretende es ser la solución software a un mundo diverso formado por componentes heterogéneos. Osea, que CORBA puede resolver el problema de interoperabilidad entre porciones software inicialmente desconectadas.

Esto es, que podré conseguir que la lógica de negocio incrustada en código COBOL pueda ser utilizada mediante invocación de métodos desde un entorno, por ejemplo, C++ o Smalltalk. Pero cabe mencionar que *CORBA no es un producto software tipo milagroso que resuelve todo*; solo son especificaciones que tratan de unificar los procedimientos y mecanismos para la uniformidad de un mundo heterogéneo de aplicaciones, es decir *no se encarga de desarrollar aplicaciones, sino estándares.*

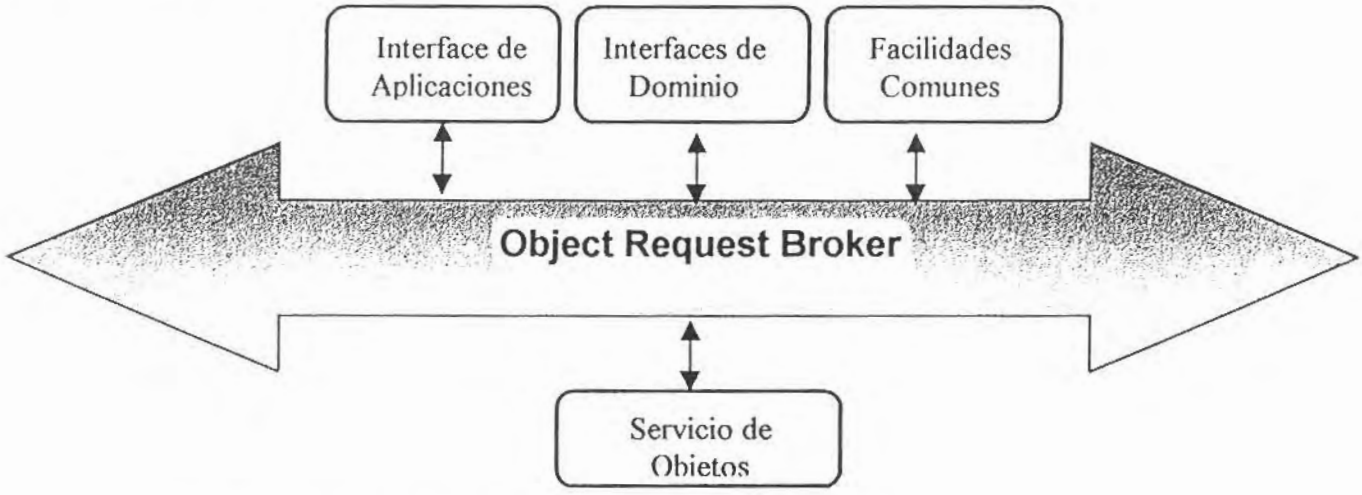
3.3.1 El Object Managment Architecture.

El Object Management Architecture (OMA), es una arquitectura integrada por cuatro componentes básicos: los orientados a sistemas (Object Request Broker y Object Services), y los orientados a la aplicación (Application Objects y Common Facilities). Con base a esto, el OMG respaldó un estándar denominado Common Object Request Broker Architecture (CORBA), la pieza fundamental de esta propuesta.

El Object Request Broker (ORB) es el elemento fundamental de esta arquitectura, encargado de *administrar y manejar todas las comunicaciones entre sus componentes, clientes, objetos y demás interfaces, permitiendo de esta manera una interacción entre objetos en ambientes heterogéneos y distribuidos*, sin importar la plataforma y lugar donde estos residan o cómo se hayan implementado, a si que el OMA está compuesto de un *Modelo de Objetos* y *un Modelo de Referencia*. El Modelo de Objetos define como los objetos distribuidos en un ambiente heterogéneo pueden ser descritos, mientras que el Modelo de Referencia caracteriza las interacciones entre los objetos, [Vinoski, 1997] y [Martin et al, 1997].

En el Modelo de Objetos, un objeto es una entidad encapsulada, con una identidad inmutable cuyos servicios pueden ser accesados solo a través de interfaces bien definidas. Los clientes hacen peticiones a los objetos para realizar servicios en su comportamiento. La implementación y la localización es transparente al que hace la petición, según se describe en [Yang et al, 1996].

A continuación se muestran los componentes del Modelo de Referencia, siendo el ORB la parte más importante en la comunicación entre clientes y objetos. La figura muestra, el concepto de Arquitectura de los Objetos. Se muestra componentes específicos de una aplicación que interactúan entre sí, para resolver un problema dado, ver figura que sigue.



Modelo general de la estructura del ORB de CORBA.

Servicios de Objetos: Son interfaces independientes del dominio que son usadas por muchos programas de objetos distribuidos. Dos ejemplos de estos servicios son el *Naming service y el Trading Service*, los cuales sirven para buscar objetos por nombre y por propiedades. Existen además otras especificaciones de Servicios de Objetos para el manejo del ciclo de vida de los objetos, seguridad, transacciones y notificación de eventos entre otras.

Facilidades Comunes: Son parecidos a los Servicios de Objetos, pero la diferencia es que están orientadas a las aplicaciones del usuario final. Un ejemplo de tal facilidad es la facilidad de Componentes de Documentos Distribuidos (DDCF), una facilidad común de documentos compuestos basados en *OpenDoc*. DDCF permite la presentación e intercambio de objetos basados en un modelo de documento, por ejemplo la facilidad de ligar un objeto de hoja de cálculo dentro de un documento de reporte

Interfaces de Dominio: Estas interfaces tienen funciones similares a los Servicios de Objetos y las Facilidades Comunes, pero que están orientadas a dominios de aplicación específicos. Por ejemplo una de las primeras Interfaces de Dominio fueron los habilitadores de Manejo de Producción de datos, para el dominio de manufactura [Yang et al, 1996].

Interfaces de Aplicación: Estas interfaces son desarrolladas para una aplicación en especial. Como son específicas, y el OMG no se encarga de desarrollar aplicaciones, sino estándares, estas interfaces no están estandarizadas.

3.3.2 Composición de CORBA.

- El ORB Core
- El OMG Interface Definition Language (IDL)
- El Repositorio de Interfaces
- Mapeos de Lenguaje
- Stubs y Skeletons
- Invocación Dinámica y despacho
- Protocolos Inter-ORB

El ORB Core.

El ORB es el encargado de manejar las solicitudes a los objetos y regresa cualquier respuesta a los clientes que hicieron las solicitudes. El objeto al que un cliente desea que el ORB direccione una solicitud es llamado el objeto destino. La principal característica del ORB es la **transparencia** de como facilita la comunicación entre el objeto y el cliente.

El ORB se encarga de ocultar:

- La localización de objetos
- La implementación de los objetos
- El estado de ejecución de los objetos
- El mecanismo de comunicación del objeto.

Estas características del ORB permiten a los desarrolladores a centrarse más en el desarrollo de su aplicación que en la forma de comunicar los diversos componentes de que está constituida.

Para realizar alguna solicitud a un objeto, el cliente debe especificar el **objeto destino** utilizando una *referencia al objeto*, esta referencia puede ser obtenida mediante varias formas:

- Creación del Objeto
- Servicio de Directorio
- Conversión de una cadena y de regreso.

El OMG Interface Definition Language (IDL).

El lenguaje de definición de Interfaces, provee características al ORB que permiten a los desarrolladores a centrarse más en el desarrollo de su aplicación que en la forma de comunicar los diversos componentes de que está constituida.

Para realizar alguna solicitud a un objeto, el cliente debe especificar el objeto destino utilizando una *referencia al objeto*, esta referencia puede ser obtenida mediante varias formas:

- Creación del Objeto
- Servicio de Directorio
- Conversión de una cadena y de regreso.

Antes de que un cliente pueda hacer solicitudes a un objeto, el cliente debe saber el tipo de operaciones que son aceptadas por el objeto. *La interface de un objeto*, especifica las operaciones y tipos que el objeto soporta y así, definir las solicitudes que pueden ser hechas al objeto. Las interfaces de los objetos son escritos en el OMG Interface Definition language (OMG IDL). Las interfaces son similares a las clases en C++ y las interfaces en Java.

Un ejemplo del OMG IDL es el siguiente:

```
interface filosofo {
    Object Create();
    void Piensa();
    void Come();
                }
```

Una característica importante del OMG IDL es la independencia del lenguaje de programación que sea utilizado para desarrollar la aplicación.

Debido a su naturaleza declarativa más que procedural, este no es utilizado como lenguaje de programación, lo que obliga a que el comportamiento de un objeto sea definido a parte de la interface del propio objeto.

Esto permite que los objeto puedan ser programados en diferentes lenguajes y se van a comunicar unos con otros sin problemas. EL OMG IDL provee de un conjunto de tipos que son similares a esos encontrados en un gran número de lenguajes de programación como son: long, double, boolean, tipos estructurados mediante struct { .. }, uniones, plantillas como sequence y string.

Dentro de las interfaces se definen los miembros del objeto, sus métodos o servicios, así como excepciones si es que hay, según explica [Yang et al, 1996].

Una importante característica de las interfaces del OMG IDL es que pueden heredar componentes de otras interfaces.

La herencia de interfaces es muy importante en CORBA, ya que permite al sistema estar abierto para extensión, pero cerrado para modificación. En CORBA, todas las interfaces son derivadas de la *interface object*. Y como esto es automático no tiene que hacer explícito en la definición de la interface.

```
interface Aplicación {
        object nueva();
      }
interface Procesador_Palabras : Aplicacion {
        object nuevo_procesador();      }
```

El Repositorio de Interfaces.

Todas las aplicaciones basadas en CORBA requieren accesar al sistema de tipos del IDL cuando se están ejecutando. Esto es necesario debido a que la aplicación debe saber los tipos de valores pasados a las solicitudes. Adicionalmente las aplicaciones deben saber el tipo de interfaces soportados por los objetos que están usando.

Muchas aplicaciones requieren solo conocimiento estático del sistema de tipos del IDL. Típicamente la especificación es traducida del IDL al lenguaje destino mediante el Mapeo de Lenguaje.

Mediante este mapeo, se obtiene código que es integrado a la aplicación, con lo que se obtiene conocimiento estático del sistema de tipos. Si este sistema de tipos cambia, entonces es necesario recompilar la aplicación, o al menos aquellos módulos que ocupan esos objetos que cambiaron. El *Repositorio de Interfaces* permite al IDL a que el sistema de tipos sea accesado y modificado en tiempo de ejecución. Este Repositorio no es más que otro objeto de CORBA que puede ser utilizado en cualquier momento, las ventajas de su uso se ven cuando se utiliza invocación dinámica de objetos, mas detalles en [Yang et al, 1996].

Mapeos de Lenguaje.

Como se menciono el IDL es solo un lenguaje declarativo, no un lenguaje completo de programación, por lo que la implementación de los servicios del objeto se debe de hacer en un verdadero lenguaje de programación. Para ello el mapeo de lenguajes determina como las interfaces definidas en el IDL serán mapeadas a las facilidades ofrecidas por un determinado lenguaje de programación. Existen hoy en día programas que hacen el mapeo para C, C++, Java, Ada 95, COBOL, SMALLTALK, UNIX Bourne Shell.

Stubs y Skeletons.

Además de generar los tipos para el lenguaje de programación usado, los *compiladores del IDL* generan stubs para los clientes y skeletons para los objetos. *Un stub es un mecanismo que crea y realiza solicitudes de servicios en un cliente, mientras que un skeleton es un mecanismo que envía solicitudes a las implementaciones de los objetos, por lo que son específicos de la interface.*

El despacho a través de stubs y skeletons es a menudo llamado invocación estática.

El stub trabaja directamente con el cliente para *marshall* la solicitud. Esto es que el stub ayuda a convertir la solicitud de la representación interna del lenguaje de programación, a una representación apta para la transmisión de la información y que pueda llegar al objeto destino. Una vez que la solicitud llega al objeto destino, con ayuda del skeleton, se *unmarshall* la solicitud y se despacha al objeto. *cuando el objeto desea devolver alguna información se procede de la misma forma en que llego* [Yang et al, 1996].

Invocación Dinámica y despacho.

Adicionalmente a la invocación estática, a través de stubs y skeletons, CORBA soporta interfaces para invocación dinámica:

- Interface para invocación dinámica
- Interface para despacho dinámico de objetos

Estos mecanismos dinámicos se pueden ver como stubs y skeletons genéricos, estos se encuentran en el ORB y no son dependientes del IDL [Yang et al, 1996].

Interface para invocación dinámica.

Mediante el uso de esta interfaz, una aplicación cliente puede hacer solicitudes en cualquier objeto sin tener conocimiento sobre sus interfaces en tiempo de compilación. Las invocaciones pueden ser realizadas en las formas siguientes:

- *Síncrona.*- El cliente se bloquea.
- *Síncrona aplazada.*- El cliente continua y después recoge la respuesta del servidor.
- *Una Vía.* El cliente solo envía, pero no recibe.

Interface para el despacho dinámico de objetos.

En forma análoga a la invocación, está el despacho por parte del servidor, el cual permite a los servidores ser escritos sin tener skeletons para los objetos de los cuales se les ha pedido algún servicio.

Protocolos Inter-ORB.

Antes de CORBA 2.0, uno de los más grandes problemas con los ORBs comerciales es que ellos no podían interoperar. Este problema se debía a que la especificación de *CORBA no imponía* formatos de datos particulares o protocolos para la comunicación de ORBs. CORBA 2.0 introduce una arquitectura general para la interoperabilidad de los ORBs que brinda interoperabilidad basada en puentes e interoperabilidad ORBs.

La interoperabilidad directa es posible cuando los ORBs residen en el *mismo dominio*, en otras palabras, que ellos entienden las mismas referencias a los objetos, usan el mismo sistema de tipos IDL y comparten la misma información de seguridad. La interoperabilidad *basada en puentes* es necesaria cuando los ORBs pertenecen a dominios distintos. El papel del puente es mapear la información específica de un ORB a otro ORB de otro dominio.

La arquitectura de interoperabilidad general de ORB's está basada en el Protocolo General Inter-ORB (GIOP), el cual especifica sintaxis de transferencia y un conjunto estándar de formatos de mensajes para interoperación de ORBs sobre cualquier transporte orientado a conexiones. El Protocolo Internet Inter-ORB (IIOP) especifica como GIOP es construido sobre transportes TCP/IP [Yang et al, 1996].

3.3.3 Caso: Visigenic VisiBroker for Java.

Visigenic fue quien desarrolló el primer CORBA ORB escrito totalmente en Java para crear, gestionar y desplegar aplicaciones Java distribuidas por múltiples plataformas. VisiBroker para Java permite aplicaciones distribuidas interoperables para Internet, entornos informáticos de empresa e Intranet corporativa mediante la implementación original del protocolo Inter-ORB de Internet (IIOP) de CORBA. IIOP es un protocolo basado en estándares para enlazar aplicaciones Java y objetos distribuidos y ha sido adoptado por casi todos los principales fabricantes de software de Internet y empresa. Visigenic ofrece además una implementación de VisiBroker para C++ con interoperabilidad completa. VisiBroker está disponible para muchas plataformas incluyendo Sun Solaris HP-UX, IBM AIX, SGI IRIX, Digital UNIX y Windows 95/NT.

Combinando sus herramientas de desarrollo de alto nivel orientadas a objetos con middleware de objeto distribuido de Visigenic, Borland podrá ofrecer a IT corporativas una solución completa e integrada para desarrollar, desplegar y gestionar aplicaciones de empresa distribuidas. Visigenic está jugando un papel esencial en la creación de la base sobre la que se desarrollan aplicaciones nuevas y fundamentales, basándose en su experiencia en tecnologías de acceso a los datos y de objetos distribuidas basadas en estándares, Visigenic se ha convertido en el proveedor principal de tecnología ORB en el mercado de software. Los productos de Visigenic incluyen VisiBroker para Java y VisiBroker para C++ y productos adicionales basados en CORBA.

Resumiendo CORBA.

En efecto CORBA es el mecanismo de comunicación entre objetos de la OMA (Object Management Arquitecture), normalizaciones ambas creadas por el OMG. CORBA se diseñó pensando en la interacción entre objetos implementados en distintos lenguajes, previendo la evidente necesidad de interoperar con "legacy objects: objetos heredados", posiblemente codificados en COBOL, C, etc. A tal fin CORBA prevé interfaces codificadas en IDL (Interface Definition Language), que permiten el intercambio de mensajes entre objetos de desconocida implementación. A la vez, CORBA incluye poderosas especificaciones de interoperabilidad entre ORBs (en CORBA 2.0), un potente servicio de nombres (incluido con otros como notificación de eventos y gestión transaccional en los denominados "servicios de objetos"), facilidades comunes e interfaces de Dominio (Domain Interfaces).

3.4 Modelo de objeto componente (Component Object Model).

Las aplicaciones que funcionan en Windows 95 y Windows NT tienen espacios de memoria protegidos. Esto asegura que una aplicación no pueda acceder a la memoria ocupada por otra aplicación. Aunque en general éste es un modelo adecuado, existen razones para proporcionar un nivel de comunicación entre aplicaciones. Este nivel de comunicación es proporcionado en Windows 95/98 y Windows NT mediante el Component Object Model (COM). Este ofrece el protocolo que permite a una aplicación interactuar con otra en la misma máquina [Pinnock, 1998].

El **Component Object Model**, es una arquitectura de componentes de software que permite que las aplicaciones y sistemas sean construidos a partir de componentes provistos por distintos proveedores de software. COM es la arquitectura subyacente que forma los fundamentos de servicios de software de mayor nivel, como los provistos por ActiveX[1]. Los servicios de ActiveX abarcan varios aspectos de software basados en componentes, incluyendo documentos compuestos, controles personalizados, "scripting" y otras interacciones de software. Estos servicios proveen de distinta funcionalidad al usuario; sin embargo, comparten un requisito fundamental por un mecanismo que permita a los componentes binarios de software, enlazarse y comunicarse el uno al otro en una forma bien definida.

[1] El apéndice A, presenta un resumen sobre la tecnología de componentes ActiveX.

Cabe destacar que Microsoft está aplicando COM en áreas específicas tales como documentos compuestos, controles, automatización, transferencia de datos, almacenamiento y nombres y otros, cualquier desarrollador puede tomar ventaja de la estructura y los fundamentos que COM provee, una mayor referencia se puede hallar en [Pinnock, 1998] y [Brockschmidt, 1993].

Ahora, podrían surgir una de la siguientes preguntas, por ejemplo, ¿Cómo habilita COM la interoperabilidad? ¿Qué lo hace un modelo útil y de unificación? Para resolver estas preguntas, será útil, primero, definir los principios de diseño básicos de COM y conceptos arquitectónicos. Al hacer esto, examinaremos los problemas específicos que resuelve y cómo provee de soluciones para estos problemas.

3.4.1 Fundamentos del modelo de objeto componente (COM).

La palabra objeto tiende a significar algo distinto para todos. Para clarificar, en COM, un objeto es una pieza de código compilado que provee de un servicio al resto del sistema. Para evitar confusión, probablemente es mejor referirse a un objeto COM como un "objeto componente" o simplemente como un "componente".

Esto evita el confundir a los objetos COM con objetos de código, como los definidos en C++. Los componentes soportan una interfaz base denominada *IUnkown,* junto con cualquier combinación de otras interfaces, dependiendo de qué funcionalidad exponga el componente.

Frecuentemente los componentes tienen datos asociados, pero a diferencia de los objetos de C++, por ejemplo, un componente nunca tendrá acceso directo a otro componente en su totalidad. En vez de esto, un componente siempre tendrá acceso a otro componente por medio de *apuntadores a interfaces*.

Esta es una característica primaria de la arquitectura del Component Object Model, porque permite conservar completamente la encapsulación de los datos y procesos, un requisito fundamental para un verdadero estándar de componentes de software. También permite que haya interoperabilidad (entre procesos o entre redes) ya que todo el acceso a datos es por medio de métodos que pueden existir en un objeto "proxy" que adelante las peticiones y regrese los resultados, según se describe en [Pinnock, 1998] y [Brockschmidt, 1993].

El component Objet Model provee de diversos conceptos fundamentales que son las bases del modelo. Estos incluyen [Pinnock, 1998] y [Brockschmidt, 1993]:

- Un estándar binario para el llamado de funciones entre componentes.
- Un agrupamiento de funciones en interfaces.
- Una interfaz base que provee de:
    - Una forma en que los componentes pueden descubrir dinámicamente las interfaces implementadas por otros componentes.
    - Conteo de referencia para permitir a los componentes monitorear su tiempo de vida y borrarse a sí mismos cuando sea apropiado.
    - Un mecanismo de identificación única de componentes y sus interfaces.
    - Un "cargador de componentes" para preparar interacciones entre componentes y ayudar a manejar las interacciones de componentes en los escenarios entre procesos o entre redes.

Estándar Binario.

Para cualquier plataforma (combinación de hardware y sistema operativo) COM define una forma estándar de organizar las tablas virtuales de funciones (vtables) en memoria y una forma estándar de llamar funciones por medio de las vtables.

Por lo tanto, cualquier lenguaje que pueda llamar a funciones por medio de apuntadores (C, C++, Ada, entre otros) puede ser usado para escribir componentes que interoperen con otros componentes escritos en el mismo estándar binario. La doble indirección (el cliente mantiene un apuntador a un apuntador a una vtable) permite que se comparta la vtable entre múltiples instancias de la misma clase del objeto. Es un sistema con cientos de instancias de objetos, el compartir la vtable puede reducir considerablemente los requisitos de memoria.

Interfaces.

En COM, las aplicaciones interactúan entre ellas y entre el sistema por medio de colecciones de funciones denominadas *interfaces*. Hay que notar que todos los servicios de ActiveX son simplemente interfaces COM. Una interfaz COM es un contrato entre componentes de software para proveer de un conjunto pequeño pero útil de operaciones (métodos) relacionados semánticamente.

Una interfaz es la definición del comportamiento y responsabilidades esperados. El soporte a arrastrar y soltar de OLE es un buen ejemplo. Toda la funcionalidad que en un componente debe implementar para ser destino de soltar se agrupa en la interfaz *Idrop Target;* toda la funcionalidad para ser fuerte (source) de "drag" se encuentra en la interfaz IdragSource.

Por convención, el nombre de las interfaces comienza con **"I"**. Incidentalmente, un apuntador a un componente es realmente un apuntador a una de las interfaces que un componente implementa; esto quiere decir que se puede usar un apuntador a un componente para invocar un método exclusivamente, pero no para modificar datos, como fue descrito arriba.

Aquí hay un ejemplo de la interfaz Ilookup con dos métodos miembro:

```
Interface Ilookup: public Iunknown
{
    public:
    virtual HRESULT __ stdcall LookUpByName
    (LPTSTR 1pName, TCHAR ** 1plpNumber) = 0;
    virtual HRESULT __stdcall LookUpbyNumber
    (LPTSTR 1pNumber, TCHAR **1plpNumber) = 0;
};
```

Atributos de las interfaces.

Debido a que una interfaz es una forma contractual para que un componente exponga sus servicios, hay algunos puntos importantes a entender:

- Una interfaz no es una clase.
- Una interfaz no es un componente
- Los clientes sólo interactúan con apuntadores a una interfaz.
- Los componentes pueden implementar herencia múltiple.
- Las interfaces tienen tipos.
- Las interfaces son inmutables.

El uso único de las interfaces en COM provee de cinco beneficios principales:

- La habilidad para que la funcionalidad en las aplicaciones (clientes o servidores de objetos) evoluciones en el tiempo.
- Interacción de objetos rápida y simple
- Reutilización de interfaces.
- Transparencia de localización.
- Independencia de lenguaje de programación.

Identificadores Globalmente únicos (GUIDs).

COM usa identificadores globalmente únicos – enteros de 128 bits que están garantizados para ser únicos en el mundo, en el espacio y en el tiempo - para identificar todas las interfaces y los componentes. Estos identificadores globalmente únicos son UUIDs (Ids universalmente únicos) como los definidos por el distributed Computing Environment del open software foundation. Esto ayuda a asegurar que los componentes de COM no se conecten accidentalmente con el componente, interfaz o método equivocado, inclusive en redes con millones de componentes.

Iunknown.

COM define una interfaz especial, Iunknown, para implementar funcionalidad esencial. Todos los componentes deben implementar la interfaz Iunknown, que tiene tres métodos: QueryInterface, AddRef y Release.

En sintaxis de C++ Iunknown aparece como sigue:

```
Interface Iunknown'
{
virtual HRESULT QueryInterface (IID & iid; void ** ppvobj) = 0;
virtual ULONG Addref () = 0;
virtual ULONG RELEASE () = 0;
}
```

AddRef y Release son métodos para el *conteo* de referencias. El método de AddRef se invoca cuando otro componente está usando la interfaz; el método *Release* se invoca cuando el otro componente ya no requiere de esa interfaz.

Mientras que el conteo de referencia del componente sea distinto a cero, debe permanecer en memoria; cuando el conteo de referencia sea cero, el componente puede descargarse a sí mismo de forma segura porque ningún otro componente mantiene referencias a él.

QueryInterface es el mecanismo que permite a los clientes descubrir dinámicamente (en tiempo de ejecución) si una interfaz es soportada por un componente o no; al mismo tiempo es el mecanismo que un cliente usa para obtener un apuntador a una interfaz de un componente.

Biblioteca del objeto componente (Component Object Library).

El component Object Library es un componente de sistema que provee de las mecánicas para COM. El Component Object Library provee de la facilidad para hacer llamadas de los métodos contenidos en "Iunkown" entre procesos; también encapsula el trabajo relacionado con instanciar componentes y establecer conexiones entre componentes, a través de la bibliotecas de interfaces.

Ahora que tenemos un buen entendimiento de las piezas fundamentales de COM, veamos cómo estas piezas se unen para habilitar el software en base a componentes:Interoperabilidad, versiones, independencia de lenguaje, e interoperabilidad entre procesos transparente.

Adicionalmente COM provee de una arquitectura de alto rendimiento para cumplir con los requisitos de un mercado de componentes comercial, con las siguientes características:

Interoperabilidad y rendimiento. La interoperabilidad se provee por COM por medio del uso de vtables para definir un estándar binario a nivel de interfaces para invocar métodos entre componentes. Las llamadas entre componentes COM en el mismo proceso no son más lentas que una invocación a objeto asociado en tiempo de compilación en C++.

Versiones. Un buen mecanismo de versiones permite que un componente del sistema sea actualizado sin requerir actualizaciones a los otros componentes del sistema. El manejo de versiones en COM se resuelve por medio de interfaces e "Iunknown::QueryInterface". El diseño de COM elimina por completo la necesidad de cuestiones como repositorios de versiones o administración centralizada de versiones de componentes.

Independencia de lenguaje. Los componentes pueden ser implementados en diversos lenguajes de programación y usados por clientes que hayan sido escritos en lenguajes de programación completamente distintos. Esto es porque COM representa un estándar para objetos a nivel binario, no a nivel de código fuente.

Interoperabilidad entre procesos transparente. En el diseño de COM se comprendió desde el principio que la *interoperabilidad* tenía que ocurrir entre procesos dado que no se puede esperar que todas las aplicaciones se vuelvan a escribir como DLLs cargadas en memoria compartida. También al resolver el problema de *interoperabilidad entre procesos* se resolvió el problema de comunicar a componentes que estén corriendo en distintas computadoras por medio de una red, usando exactamente la misma interfaz programática que en el caso de componentes comunicándose en la misma computadora [Pinnock, 1998] y [Brockschmidt, 1993].

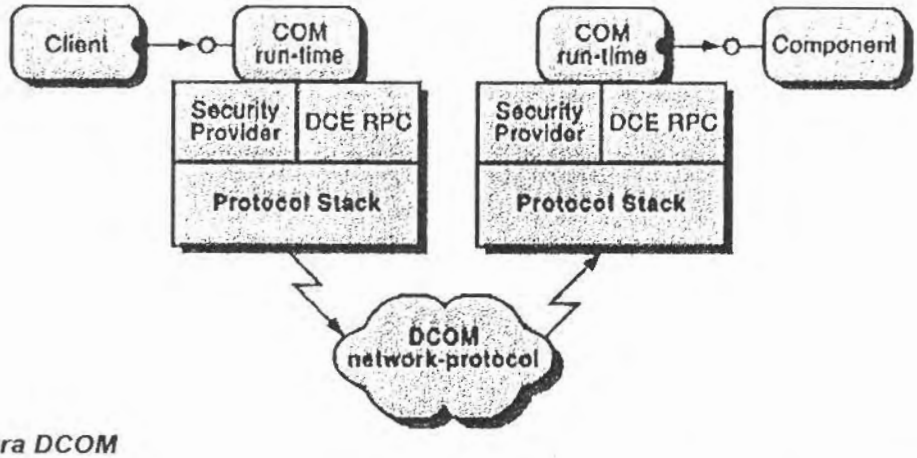
3.5 Modelo de objeto componente distribuido (DCOM).

Como ya sabemos, COM es un estándar para la **comunicación entre objetos** independientemente del lenguaje en el que hayan sido escritos. El objeto cliente accederá a los métodos del objeto servidor a través de interfaces COM normalizados.

DCOM extiende lo anterior a un **ámbito de red**; los objetos a comunicarse no tienen porqué compartir la misma máquina. Pero *esto no es nuevo*: lo podíamos hacer desde hacía tiempo con las RPC. De hecho, DCOM está construido sobre las RPC; se trata de un estándar de más alto nivel, con el que podremos escribir aplicaciones distribuidas en un entorno de red sin necesidad de conocer todos los entresijos de las RPC, y además con un enfoque orientado a objetos. De hecho, Microsoft también se refiere a DCOM como Object RPC (**ORPC**).

DCOM se incluyó a Windows NT en su versión 4.0 (también salió en 1996 una versión para Windows 95), y actualmente la empresa **Sofware AG Corp.** lanzo versiones para Solaris, Linux y otros sistemas operativos.

La imagen que se muestra a continuación representa gráficamente la arquitectura DCOM [Pinnock, 1998] y [Wang et al, 1997]:



Arquitectura DCOM

DCOM ofrece una tecnología de componentes distribuidos con muchos beneficios que van más allá de proporcionar simplemente la habilidad de crear aplicaciones en red. Las secciónes siguientes resaltan algunos de los *beneficios de DCOM.*

3.5.1 Independencia en la localización.

Cuando comienza la ejecución de una aplicación distribuida en una red, varios apremios del diseño que están en conflicto llega a ser evidentes:

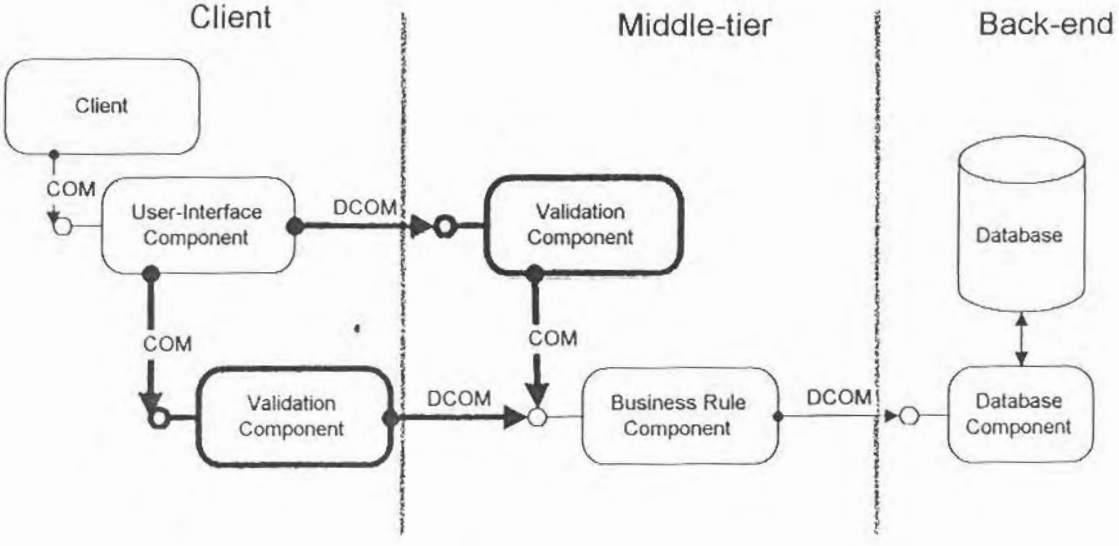
- Los componentes que interactúan más deben estar "más cerca" el uno del otro.
- Algunos componentes se pueden ejecutar solamente en máquinas o en localizaciones específicas.
- Componentes más pequeños aumentan la flexibilidad del desarrollo, pero también aumentan el tráfico en la red.
- Componentes más grandes reducen el tráfico en la red, pero también reducen la flexibilidad del desarrollo.

Con DCOM el diseño crítico de los problemas son bastante fáciles de trabajar, debido a que los detalles del desarrollo no se especifican en el código de fuente. DCOM oculta totalmente la localización de un componente, si está en el mismo proceso del cliente o en una máquina a medio camino alrededor del mundo. En todos los casos, la manera que el cliente se conecta con un componente y llama a los métodos del componente son idénticos.

DCOM *no requiere no solamente ningún cambio al código de fuente, él incluso no requiere que el programa sea recompilado.* Una reconfiguración simple cambia la manera en que los componentes se conectan el uno con el otro.

Supongamos, por ejemplo, que ciertos componentes se deben poner en una máquina específica o en una localización específica, si la aplicación tiene numerosos componentes pequeños, nosotros podemos reducir la carga de la red desplegándolos en el mismo segmento de la LAN, en la misma máquina, o aún en el mismo proceso. Si la aplicación esta compuesta de un número pequeño de componentes grandes, la carga de la red es un problema menor, así que podemos poner estos componentes en las máquinas más rápidas disponibles, dondequiera que se encuentren estas máquinas [Pinnock, 1998] y [Wang et al, 1997].

La siguiente figura muestra un ejemplo de cómo un mismo "componente de validación" se puede desplegar en la máquina del cliente, cuando la anchura de banda de la red entre la máquina " cliente" y " Middle-tier" es suficiente, y en la máquina servidor, cuando el cliente está teniendo acceso a la aplicación con una conexión lenta de la red.



Localización e Independencia.

Con independencia de la localización de DCOM, la aplicación puede combinar componentes relacionados en las máquinas cercanas, el uno al otro dentro de una sola máquina o aún en el mismo proceso.

Incluso si un número más grande de componentes pequeños pone las funciones en ejecución de un módulo lógico más grande, pueden interactuar entre ellos obrar recíprocamente eficientemente entre uno u otro.

3.5.2 Gestión de la conexión.

DCOM utiliza un contador de referencia (reference counting) para determinar cuántos clientes están conectados a un determinado servidor. Así, cuando un nuevo cliente se añade a la lista, el contador se incrementa y, a la inversa, cuando un cliente se desconecta del servidor de aplicaciones, la referencia disminuye. Cuando el contador de referencia llega a cero, la aplicación se cierra. Cuando un servidor de aplicaciones ya no esté en funcionamiento, DCOM lanzará la aplicación y establecerá el contador de referencia de la manera más adecuada.

Si un cliente se desconecta sin que disminuya el contador de referencia, debe existir un modo de que el servidor disminuya el contador de referencia. Esto se lleva a cabo enviando el servidor un "ping" al cliente. Si el servidor no logra hacer llegar al cliente un "ping" tres veces seguidas, la conexión se considerará cerrada y el contador de referencia será ajustado (garbage collecting).

3.5.3 Interfaces múltiples.

DCOM se implementa fundamentalmente mediante la utilización de interfaces. Los interfaces son los métodos expuestos y las propiedades de un objeto en particular. Puede haber muchos interfaces para el mismo objeto. Esto ofrece la posibilidad de personalizar un objeto sin tener necesariamente que volver a desarrollarlo. Esto es especialmente importante cuando el objeto requiere una funcionalidad adicional que sólo necesitan un subconjunto de los clientes. En esta situación, el desarrollador podría *utilizar la misma implementación con distintos interfaces*. Un interface puede proporcionar la funcionalidad básica necesaria; el otro puede proporcionar la funcionalidad mejorada. Esto puede llevarse a cabo incluso aunque la implementación real pueda ser exactamente la misma, para mas detalles se recomienda la bibliografía de [Pinnock, 1998] y [Wang et al, 1997].

3.5.4 Integración con otros protocolos de red.

Como DCOM está basado en sencillos protocolos TCP/IP y RPC, los servidores de aplicación pueden llamarse desde cualquier punto de Internet utilizando TCP/IP básico u otros protocolos basados en TCP (como PPTP --Point to Point Tunneling Protocol--) para usuarios que conectan o HTTP para usuarios de Internet. Soportando el nivel más bajo de interoperabilidad de red, la comunicación en red de DCOM es transparente para las aplicaciones. Esto ofrece muchas opciones de red distintas para el desarrollador para que tome la decisión de elegir la mejor conectividad para la aplicación, segun [Pinnock, 1998] y [Wang et al, 1997].

3.5.5 Independencia en la ubicación y equilibrio de carga.

Aquí el concepto es que la aplicación cliente no se ocupa de averiguar dónde está situado el servidor de aplicaciones, sólo de que pueda conectarse a él. Como DCOM está dirigido hacia aplicaciones conectadas en red, el único criterio que se puede aplicar es que la aplicación exista realmente en la red, a continuación se ilustrara esto, por medio de un ejemplo.

Suponemos que el modo en que una máquina cliente "encuentra" a un servidor de aplicaciones es localizando las entradas de registro adecuadas en el archivo de registro del cliente. Simplemente cambiar la información de registro del Servidor A al Servidor B hará que el cliente cargue la aplicación en el Servidor B. Cuando una aplicación tiene muchos usuarios, éstos pueden dividirse en grupos de usuarios.

Por ejemplo imaginemos que existen dos grupos de trabajo, el Grupo 1 podría estar utilizando un servidor de aplicaciones de inventario que está funcionando en el Servidor A mientras que el Grupo 2 puede utilizar la misma aplicación funcionando en el Servidor B. Haciendo esto, la carga puede equilibrarse a lo largo de las distintas máquinas.

Este tipo de **equilibrio de carga** se llama *estático*, porque los usuarios siempre conectan con el mismo servidor, sea cual sea el número de usuarios que estén conectados. Esto está bien para situaciones sencillas, pero la mayoría de las veces la carga necesita equilibrarse dinámicamente. *Teniendo en cuenta el ejemplo de arriba, añadamos un poco de complejidad más realista*. En ciertosomentos del día, el Grupo 1 requiere utilizar mucho más el servidor de aplicaciones. En ese mismo momento, las necesidades del Grupo 2 son bajas.

Si se utiliza equilibrio de cargas estático, el Servidor 1 estará bastante ocupado, mientras que el Servidor 2 estará sobrecargado. Si hubiera una forma de pasar dinámicamente algunos usuarios del Grupo 1 al Servidor 2, en lugar de al Servidor 1, *basándose en el tráfico de red existente para cada servidor, la utilización global estaría equilibrada en los dos servidores, dando como resultado un mejor rendimiento.* "DCOM por sí solo no resuelve equilibrio de carga dinámica. No obstante, recomienda que los desarrolladores creen un objeto de referencia para ocuparse de este requisito de aplicación", ver mas detalles en [Pinnock, 1998] y [Wang et al, 1997].

3.5.6 Seguridad.

La política de seguridad predeterminada de COM distribuido protege al equipo remoto frente a la creación no autorizada de objetos. Los únicos objetos que se pueden crear de forma remota son los autorizados por el administrador del equipo remoto. La seguridad para la creación de objetos remotos es importante por la propia sencillez de COM distribuido. Lo único que determina si un cliente utiliza un componente ActiveX local o el mismo componente en ejecución en un equipo remoto, es la entrada para el componente del Registro de Windows del equipo cliente.

3.5.7 Configuración de la política de seguridad.

COM distribuido ofrece cuatro configuraciones de la política de seguridad para los equipos de red con componentes ActiveX, (El apendice A, presenta una referencia sobre ActiveX) . Puede establecer el nivel de control de acceso a un equipo de la red con el Administrador de conexiones. Para establecer la política de seguridad, debemos instalar y ejecutar el *Administrador de conexiones* en el equipo remoto.

La ficha Acceso del cliente del Administrador de conexiones permite seleccionar una de las siguientes configuraciones de política de seguridad del sistema para un equipo remoto [Wang et al, 1997].

En la tabla siguiente se enumeran en la columna Valor los valores de la configuración de preferencias de RemoteActivationPolicy.

Nombre	Valor	Descripción
No permitir ninguna creación remota	0	No permite que se cree ningún objeto.
Permitir creaciones remotas por clave	2	Sólo se puede crear un objeto si está activada la casilla de verificación **Permitir activación remota**. Esto modifica el CLSID del Registro de Windows para incluir el siguiente valor de subclave: **AllowRemoteActivation=Y**
Permitir creaciones remotas por ACL	3	Un usuario puede crear un objeto sólo si la Lista de control de acceso para el CLSID del Registro de Windows incluye al usuario. Sólo para Windows NT.
Permitir todas las creaciones remotas	1	Permite que se cree cualquier objeto. No recomendado fuera del entorno de desarrollo.

Nota: El Id. de clase (CLSID) de una clase pública (y, por extensión, de los objetos creados a partir de la clase) es un entero de 128 bits que identifica a la clase de forma única en el Registro de Windows de cualquier equipo.

3.5.8 Autentificación de datos

La autentificación de las llamadas a procedimiento remoto (RPC) hace referencia al nivel de integridad de los datos garantizado para la comunicación entre dos equipos a través de la red. Para los componentes ActiveX remotos que se ejecuten en cualquier sistema operativo de Windows, RPC proporciona siete niveles de autentificación, mostrados en la tabla siguiente [Wang et al, 1997].

Valor	Nombre	Descripción
0	Predeterminado	Utiliza el valor predeterminado de la red.
1	Ninguno	Sin autentificación.
2	Conectar	Se autentifica la conexión con el servidor.
3	Llamar	Sólo se autentifica al comienzo de cada llamada a procedimiento remoto, cuando el servidor recibe la petición. No se aplica a las secuencias del protocolo basadas en la conexión (las que empiezan con el prefijo "ncacn").
4	Paquete	Comprueba que todos los datos recibidos son del cliente esperado.
5	Integridad del paquete	Comprueba que no se ha modificado ninguno de los datos transferidos entre el cliente y el servidor.
6	Privacidad del paquete	Comprueba todos los niveles anteriores y codifica los valores de argumento de cada llamada a procedimiento remoto.

Los niveles están enumerados en orden creciente de autentificación. Cada nuevo nivel agrega una característica más a la autentificación proporcionada por el nivel anterior. Si la biblioteca RPC de tiempo de ejecución no admite el nivel especificado, actualiza automáticamente al siguiente nivel superior compatible.

Se debe evaluar con cuidado la necesidad de utilizar la autentificación de RPC, porque a medida què crece el nivel de autentificación de RPC se reduce el rendimiento. Puede especificar un nivel de autentificación para cada clase del componente ActiveX, de forma que no sea necesario aplicar a todo el servidor los niveles costosos, como la codificación.

Por ejemplo, un servicio de datos implementado como un componente ActiveX remoto podría tener una clase Logon que se utiliza para transmitir la información de usuario y contraseña, y esta clase podría requerir el tipo de autentificación Privacidad del paquete. Otras clases expuestas por el servidor podrían utilizar un nivel de autentificación inferior.

El nivel de autentificación se especifica en el Registro de Windows del equipo cliente, bajo el Id. de clase del objeto remoto. La subclave se llama **AuthenticationLevel**. Si esta subclave no existe, se utiliza None. Si el valor no es ninguno de los mostrados en la tabla anterior, se produce un error de ejecución de RPC. Puede elegir un nivel de autentificación predeterminado para un cliente que utilizará el servidor remoto y sustituirlo para las clases específicas que requieren una autentificación más estricta.

3.5.9 Seguridad frente a los fallos y tolerancia a errores.

DCOM proporciona soporte básico para la seguridad frente a los errores con las utilidades de "pinging". Esencialmente, esto significa que si un servidor de aplicaciones falla por cualquier razón, DCOM intentará reiniciar la aplicación. Aunque ésta es una función importante, no resuelve momentos en los que el servidor de aplicaciones no puede reiniciarse debido a diversas razones, incluyendo fallos en la red, fallos de hardware, fallos en la aplicación, etc.

La solución recomendada a un sistema de aplicaciones realmente tolerante a los fallos ha de utilizar la técnica, de cuando los clientes detectan el fallo de un componente, vuelven a conectar con el mismo componente de referencia que estableció la primera conexión. El componente de referencia tiene información sobre los servidores que ya no están disponibles y proporciona automáticamente al cliente una nueva muestra del componente funcionando en otra máquina, ver [Wang et al, 1997].

3.5.10 Caso: OLEnterprise.

OLEnterprise es un producto de Borland que provee un acceso dinámico a objetos OLE distribuidos para un acceso transparente suministrando las herramientas y utililerías para la conexión de las aplicaciones de escritorio con datos y servicios que residen remotamente utilizando mecanismos OLE, es decir que permite centralizar la lógica de la aplicación y el acceso a los datos en un servidor de aplicación.

OLEnterprise tiene tres componentes: el Explorador de Objetos (local y remoto), el Objeto Agente (servidor OLE de automatización) y el Controlador Remoto de Automatización OLE *(Object Factory)*.

El Objeto Agente.

Es una aplicación *in-process* de servidor OLE de automatización que se ejecuta como una Biblioteca de Enlace Dinámico (DLL) y se localiza en la máquina cliente, proveyendo un acceso transparente y dinámico a cualquier objeto OLE o RPC. Así la aplicación cliente se ve simplemente como cualquier otro servidor OLE de automatización, con la capacidad de devolver un objeto solicitado a un servidor remoto. Si el objeto remoto es OLE la solicitud es devuelta al *Object Factory* que vuelve a emitir la solicitud al sistema remoto; en cambio si el objeto remoto es *enterprise*, la petición se convierte dinámicamente en una llamada RPC solicitada por el Objeto Agente y devuelta al servidor apropiado [Koulinitch et al , 1997].

Un Objeto Agente se encuentra en la máquina cliente y tiene la capacidad de interpretar y |procesar transparente y dinámicamente cualquier solicitud de automatización OLE y convertirla en una llamada RPC para devolverla a cualquier servidor. En ambos casos la comunicación se maneja por el mecanismo RPC, que se llama RPC nativo (NRPC). Cuando un componente se ejecuta en una plataforma PC, NRPC usa el RPC de Microsoft. Si el objeto remoto es una automatización OLE, la solicitud es emitida al *Object Factory,* la cual a su vez emite una solicitud de automatización OLE al sistema remoto. El Objeto Agente es el servidor de automatización en la máquina cliente, y el Microsoft RPC es el mecanismo de comunicación para OLE Remoto que sirve como interface entre el Objeto Agente y el *Object Factory.*

El Object Factory.

Es un controlador OLE remoto (archivo *.EXE) que distribuye los servicios de automatización OLE (administrador de solicitudes remotas OLE) y se encuentra en el servidor que es responsable de procesar las solicitudes, a través de instancias de objetos que se localizan en clientes remotos [Koulinitch et al , 1997].

El Explorador de Objetos.

Tiene tres componentes principales: un visualizador para inspeccionar registros locales o remotos; un mecanismo de exportación de objetos OLE, para seleccionar servidores de automatización para el acceso remoto, y un mecanismo de importación para registrar servidores OLE remotos de automatización local. En la plataforma del servidor se generará y registrará localmente el servidor de automatización OLE usando sus utilerías estándar.

La referencia para invocar al servidor de automatización OLE es un programa identificado (ProgId) que se asocia a una clase identificada (CLSID), y ambos se crean cuando el servidor de automatización OLE se agrega a la base de datos de registros locales.

Por lo que la infraestructura *OLEnterprise* provee una alta posibilidad crecimiento, pues esta soportado por Microsoft RPC. Estos servicios permiten la localización de procesos y datos, la moderación en forma balanceada de la carga de trabajo en los servidores, la detección automática de las fallas en los servidores, la autorización de usuarios y la protección de datos, como implementación bajo la tecnología de DCOM y ActiveX, podemos considerarlo como una herramienta muy útil para desarrollar algunas aplicaciones distribuidas, pero este producto al estar bajo licencia de Microsoft DCOM, sus posibilidades de evolución están sin duda a la par de la tecnología de Microsoft.

Conclusiones COM/DCOM/ActiveX.

El núcleo del component Object Model es una especificación de cómo interactúan los componentes y sus clientes. Cómo especificación define un número de otros estándares para la interoperabilidad de los componentes de software:

- Un estándar binario para invocación de métodos entre componentes.
- Agrupación de métodos en interfaces.
- Una interfaz base Iunknown que provee de:
    - Una forma para que los componentes descubran dinámicamente las interfaces soportadas por otros componentes (QueryInterface).
    - Conteo de referencias para encapsular el tiempo de vida de un componente (addRef y Release).
    - Un mecanismo para identificar de una forma única a los componentes y sus interfaces (GUIDs).

La implementación que ha realizado Microsoft de DCOM y ActiveX ha supuesto un gran avance hacia el esfuerzo de la informática multinivel (multi-tier, - Ver Capitulo 4). No obstante, hay desafíos adicionales que quedan sin resolver por Windows y DCOM. Estos campos incluyen equilibrio de cargas, seguridad frente a los fallos, tolerancia a los errores.

3.6 Perfil de tecnología de sistemas de cómputo basados en objetos distribuidos.

Desde el punto de vista de objetos distribuidos, sabemos que cuando los componentes se distribuyen a través de una red, los clientes pueden ser servidores e inversamente, los servidores pueden ser clientes. Eso realmente no importa puesto que estamos hablando de la cooperación de objetos: el cliente solicita servicios de otro objeto, el objeto del servidor satisface la petición, y aunque los clientes y los servidores "viven" en sus propios mundos dinámicos, los objetos aparecen como si fueran locales, puesto que la red es la computadora.

El ORB proporciona los medios a los objetos, para localizar y activar a otros objetos en una red, sin importar el procesador o el lenguaje de programación usado para desarrollar objetos del cliente o del servidor. El ORB hace que estas cosas sucedan transparentemente al usuario.

De tal forma que el ORB es el *middleware* de cómputo para los objetos distribuidos que permite la interoperabilidad entre redes heterogéneas de objetos. A hora surge la siguiente cuestión, qué me ofrecen las diferentes alternativas de Middleware para cómputo distribuido basado en objetos (ORBs)? En esencia: *portabilidad entre plataformas y reuso de componentes*. O sea, que puedo codificar módulos software con distintos lenguajes de programación e instalarlos en distintas plataformas y, más tarde, establecer comunicación entre ellos: un módulo COBOL inserto en un RS/6000 bajo AIX podrá invocar un método en un módulo Java bajo una SparcStation en Solaris. Y la pregunta surge, ¿cómo se entienden distintos lenguajes, distintas implementaciones? Mediante un filtro intermedio, que en el caso de CORBA y DCOM son las interfaces IDL (Interface Definition Language). O sea, se trata de interfaces (sin implementación) codificados con unos lenguajes especialmente creado a tal fin y se asemeja mucho a C++. Así las invocaciones de métodos se realizan sobre declaraciones de métodos explícitos de interfaces IDL, para luego traspasar tales peticiones al código de implementación por debajo del interfaz.

He aquí un ejemplo:

```
Interface Político {
        Public void saluda ( );
};
```

Por su puesto que habrá que unir el interfaz a la implementación con algo más que meras palabras: con compiladores IDL. Y se notará, por último, que la conectividad no es automática, la gran ventaja es que los resultados típicos de métodos de análisis y diseño (estructurados u orientados-a-objetos) pueden ser expresados mediante IDL (que, por cierto, es, como CORBA y DCOM, han conseguido un importante apoyo industrial porque intenta crear normalizaciones en este campo del cómputo. La gran ventaja de usar IDL como lenguaje de especificación es que, de paso, estamos adelantado más de un paso para conectar nuestra solución software mediante CORBA o DCOM al resto del "universo" de las aplicaciones.

Finalmente podemos decir que CORBA y DCOM/ActiveX, son una buena plataforma para el desarrollo de sistemas abiertos, basados en objetos componente(componentes). En el fondo, su modelo de programación se basa en el descubrimiento y uso de las interfaces que ofrecen los objetos, por lo que las interacciones entre componentes quedan determinadas por las interfaces que conocen e implementan. Los objetos deben implementar unas interfaces especificas, según quieran ser *contenidos* o *contenedores*; para trabajar conjuntamente, solo necesitan saber que el otro objeto tiene las interfaces correspondientes. Los sistemas nuevos que implementan interfaces conocidas por los demás pueden integrarse inmediatamente al ambiente.

Capitulo 4

ARQUITECTURA DE CÓMPUTO DISTRIBUIDO BASADA EN OBJETOS COMPONENTE.

En la actualidad, dentro del área de cómputo distribuido y paralelo se tiende al desarrollo de entornos que permitan la reutilización efectiva, tanto de las unidades funcionales de un sistema como de la propia arquitectura o configuración del mismo. La idea básica es desarrollar patrones o esquemas de arquitecturas, de forma que el desarrollo de nuevos sistemas software se vea facilitado mediante la instanciación de alguno de estos patrones comunes.

La arquitectura propuesta en este trabajo esta diseñada para permitir a las aplicaciones ser distribuidas a través de múltiples niveles. En este capitulo se describe la arquitectura de objetos componente para desarrollo de cómputo distribuido, y se hace referencia de algunos otros trabajos relacionados; al final del capitulo se dan las conclusiones respectivas y se hace referencia del trabajo futuro que se podría abordar.

4.1 Antecedentes.

Las estrategias de las compañías proveedoras de sistemas de cómputo se pueden clasificar en dos tipos en el primero, tratan de formar consorcios y organismos a nivel, mundial que establezcan estándares para regular el desarrollo de nuevas tecnologías. Esto dentro de un marco que permita integrar esta gama de avances tecnológicos para formar soluciones más completas. En el segundo tipo, las compañías proveedoras crean arquitecturas propietarias que involucran el acomodamiento de sus productos para dar la impresión de que las empresas pueden adoptarlas como modelos generales.

Sin embargo, la estrategia de los clientes de esas empresas deben centrarse en construir una arquitectura de tecnología de la información (TI), propia y robusta, que permita *agregar* y/o *quitar* componentes de acuerdo a las necesidades de su negocio, así como a los avances de la tecnología.

Por otro lado es importante mencionar que aparte de los *problemas* propios que plantea el desarrollo de aplicaciones a partir de componentes software (como son la compatibilidad de interfaces, corrección o reutilización de los componentes)[1], el uso de los sistemas distribuidos introduce nuevas dificultades, como son la heterogeneidad de sus partes, los cambios dinámicos en su configuración, la evolución de sus componentes o los posibles retrasos y errores en las comunicaciones.

Actualmente, la mayoría de los esfuerzos se centran en el desarrollo de mejores sistemas que permitan tratar con estos problemas, pues como se mencionara en posteriores secciones, la *tecnología cliente/servidor*, por sí misma, no provee un modelo cognoscitivo uniforme para compartir la información entre sistemas y la gente, la lógica del negocio y de la interfaz se mezclan juntas dando por resultado redundancia – ver sección 4.2. Por lo que el diseño y el desarrollo cliente/servidor necesita incorporar la energía de la tecnología orientada a objetos (Objetos distribuidos), como se analizo en el capitulo 3, algunas de estas implementaciones ya se están logrando – CORBA y DCOM; pues el mezclar la *integridâd cognoscitiva y semántica* de objetos con el potencial de la distribución de arquitecturas cliente/servidor mantiene la gran promesa de distribuir objetos a través de una heterogénea red permitiendo a cada uno de sus componentes interactuar como una sola unidad, de acuerdo con [Cockburn et al , 1996] y [Jennings et al , 1998].

Esta tecnología provee de una infraestructura para abastecer a los componentes de servicios disponibles y así reunir los procesos cooperativos distribuidos de una forma universal y transparente. Sin embargo, aquellos requisitos propios de la aplicación que no queden cubiertos por el sistema han de ser incorporados a los componentes, lo que complica su diseño y dificulta su reutilización.

En este trabajo se trata de simplificar tanto a los componentes como a los propios sistemas, y para ello se ha tomado como base una *estructura "sistemas-propiedades-componentes"*, en la que los componentes pueden concentrarse solamente en sus labores computacionales, estos ofrecen únicamente la infraestructura mínima de creación e interconexión de componentes, mientras que el resto de los requisitos de la aplicación (*propiedades* como adaptabilidad, seguridad, reconfiguración dinámica, etc) son, implementados por los *controladores* – ORB middleware (CORBA, DCOM), entidades computacionales que son añadidas a los componentes para modificar su "comportamiento".

[1] Ver sección 2.3.2 del capitulo 2.

4.2 Conceptos básicos de Arquitecturas.

Definición.

Uno de los aspectos fundamentales en el diseño de cualquier sistema software de cierta complejidad es su estructura o arquitectura, representada por un conjunto de elementos o *componentes* computacionales y una serie de conexiones o interacciones entre estos elementos.

Podemos definir a la *arquitectura* como una *descripción de alto nivel de la organización de responsabilidades funcionales dentro de un sistema, que define la relación de los componentes (sub-sistemas, herramientas, o aplicaciones) del sistema*; sin embargo, una arquitectura no es una descripción de una solución específica a un problema o de un mapa itinerario para el éxito en diseño.

Una arquitectura no dirige a un diseñador a una solución acertada, de gran alcance o elegante a un problema específico. Cabe mencionar que la arquitectura por sí misma, sin un sistema o producto que la implemente, no es muy útil. Por lo tanto, un sistema o producto que implemente la arquitectura es lo que hace que esa arquitectura sea utilizable

4.2.1 Niveles de las Aplicaciónes.

Como se definió anteriormente, sabemos que una arquitectura dicta el modo en cual una aplicación es creada y como sus componentes son distribuidos a través del sistema. Primero debemos comprender los términos generales de los conceptos de *cliente y servidor*. Las definiciones generales de estos términos pueden ser expresadas como:

Cliente: Un consumidor de recursos.
Servidor: Un proveedor de recursos.

Podemos considerar que, todas las *aplicaciónes* fundamentalmente están compuestas de tres tipos esenciales de niveles o capas[2]:

- La capa de *la Presentación*: la presentación de información de puntos externo al sistema (También conocido como la interfaz de usuario). La capa *de la presentación* contiene la **lógica de la representación** de la información a un origen externo y obtiene entradas.

[2] Usaremos los términos de *capas o niveles* en forma indistinta.

En muchas casos el origen externo es un humano (*end user)* trabajando en una terminal o workstation, aunque el origen externo también puede ser provisto por un agente automatizado como un robot, un teléfono, o algún otro dispositivo de entrada. La *lógica de presentación* generalmente provee un menú de opciones para permitir al usuario navegar a través de las partes diferentes de la aplicación, y pueda manipular la entrada y campos en la presentación. Frecuentemente la capa *de presentación* también ejecuta una cantidad limitada de validación de datos de entrada.

- La capa de las *Reglas del Negocio*: Los procesos que implementan e imponen el contexto de los datos dentro del uso especificado de negocios, es decir que contiene la lógica de aplicación la cual gobierna, las funciones del negocio ejecutado por la aplicación. Estas funciones son invocados por la capa de presentación, o por otra función del negocio, según el flujo de trabajo del sistema. Las funciones de negocio generalmente ejecutan algún tipo de manipulación, análisis, y transformación de datos.

- La capa de *acceso a los datos*: Proporciona la vía para poder manipular, y recuperar la información acerca de los negocios con un sistema de almacenamiento de datos tal como los administradores de base de datos o sistemas de archivos, o con alguno otro tipo de origen de datos externo. Las funciones de acceso de datos generalmente son invocadas por una función de negocios, aunque en aplicaciones sencillas ellos sean invocados directamente por un componente de presentación.

En general podemos considerar, que todas las aplicaciones tienen *tres implementaciones de los niveles presentados arriba de alguna u otra manera*. Lo que hace definir aplicaciónes de un-, dos- o tres-niveles (one-, two-, three-tier) formas en las cuales estos *Niveles* se conceptualizan, como implementación dividida de niveles de la aplicación separadas unas de otras.

4.2.1.1 Las aplicaciones de un-nivel.

Las aplicaciónes con arquitectura de *un solo nivel* están basadas en un amplio marco de trabajo en el que todas las capas se combinan en un solo programa integrado que funciona sólo en una máquina. Éste es el tradicional entorno mainframe o miniordenador (monolíticos). El enfoque de un solo nivel proporciona una serie de ventajas significativas. Como la aplicación está centralizada en un entorno único, es fácil gestionar, controlar y asegurar estas aplicaciones.

Hay una serie de *notables desventajas* asociadas al método de un solo nivel. Como las aplicaciones de un solo nivel están confinadas a un solo procesador, la capacidad de dimensión puede ser una perspectiva costosa (pobremente escalable). Si la máquina actual en uso se sobrecarga, el único recurso es actualizarla a una máquina mayor. Esencialmente, las empresas están bloqueadas en su plataforma de hardware específica. Como resultado de ello, las empresas no pueden aprovechar las nuevas tecnologías.

Aplicación de Un-Nivel

En aplicaciones de Un-Nivel, es difícil hacer modificaciones a los sistemas de aplicación en respuesta a los requerimientos de cambios de los negocios. Por que la *lógica de aplicación* es solamente accesible a través de la *lógica de presentación* integrada.

4.2.1.2 Aplicaciones de dos-niveles.

Con la llegada de los ordenadores personales, las redes de área local, las bases de datos relacionales, potentes aplicaciones y herramientas de escritorio, el mercado de la informática ha pasado al reino de los *sistemas abiertos* y *cliente/servidor*. Los usuarios que toman las decisiones pueden generar sus propios informes y manipular datos de herramientas potentes en sus estaciones de trabajo de escritorio. La *arquitectura de dos niveles* permite al usuario realizar funciones que no han sido capaces de hacer antes.

La arquitectura cliente/servidor de *dos niveles* divide el proceso entre una estación de trabajo y un servidor.

Aplicación de Dos-Niveles

El método cliente/servidor de dos niveles proporciona ciertas ventajas significativas sobre el método de un solo nivel. Las herramientas de desarrollo GUI permiten el desarrollo y despliegue más rápidos de aplicaciones. Pasando buena parte del proceso de aplicación a las estaciones de trabajo, los sistemas de servidor no necesitan ser tan grandes. Los sistemas de servidor UNIX menores son notablemente más baratos que los grandes sistemas de mainframe.

A cambio de estas ventajas, el método de dos niveles pierde un poco de *seguridad, fiabilidad, capacidad de dimensión y control.* El modelo de dos niveles funciona con gran eficacia siempre que esté restringido al desarrollo de aplicaciones menores, al acceso de pocas bases de datos relacionales y no soporte una base de usuarios "grande".

Pero a medida que las aplicaciones se hacen más complejas, en términos de algoritmos procesados, número de bases de datos accedidas o número de usuarios soportados, el método de dos niveles empieza a caer. Sin los rigurosos controles de seguridad proporcionados por un entorno centralizado, cada aplicación cliente debe mejorar su propio proceso de seguridad. Por el momento, cliente/servidor sigue siendo todavía muy útil en aplicaciones de soporte de decisiones.

4.2.1.3 Aplicaciones de tres-niveles.

Hay un modo de lograr que los beneficios de cliente/servidor superen a sus inconvenientes. La *arquitectura cliente/servidor de tres-niveles* proporciona un entorno que soporta todos los beneficios, tanto del método de un solo nivel como del de dos niveles, y también soporta los objetivos de una arquitectura flexible.

Los tres niveles se refieren a las tres partes lógicas que componen una aplicación, no al número de máquinas utilizadas por la aplicación. El modelo de aplicación de tres-niveles, divide una aplicación en sus tres tipos de componente lógicos: *lógica de presentación, lógica del negocio y lógica de acceso a los datos*, donde puede haber cualquier número de cada uno de los tipos de capas dentro de una aplicación. Las capas de aplicación se comunican unos con otros utilizando una interface abstracta, que oculta la función subyacente realizada por la correspondiente capa.

Aplicación de Tres-Niveles

Las ventajas de una arquitectura de tres-niveles van más allá del ciclo de vida de una sola aplicación. En realidad, lo que se está creando no es una aplicación: *es un conjunto de módulos cliente y servidor que se comunican mediante interfaces abstractas y estandarizados y, cuando se combinan, se comportan como un sistema de aplicación integrado.* Cada módulo es realmente un objeto que puede reutilizarse y compartirse e incluirse en otros sistemas de aplicación.

Consideremos el siguiente ejemplo, de una aplicación estratégica dentro de un departamento de ventas debería poder visualizar información de inventario e interface con el sistema de entrada de pedidos.

Así que, la aplicación de ventas podría incluir componentes comerciales y de acceso a datos desde el inventario y desde aplicaciones de entrada de pedidos. Estos componentes podrían también reutilizarse en una aplicación de planificación estratégica para el departamento de fabricación.

La ventaja más obvia de una *arquitectura de tres-niveles* es la facilidad de mantenimiento. Como las funciones de aplicación están aisladas dentro de pequeños componentes de aplicación, la lógica de aplicación puede modificarse con mucha más facilidad que antes. Por ejemplo, una función que realiza una aplicación financiera es proyectar beneficios posteriores a los impuestos. Los algoritmos de esta función cambian periódicamente cuando cambia la normativa de los impuestos. Normalmente, cambiar la normativa de los impuestos requiere notables modificaciones en la aplicación financiera entera. Aislando estas reglas comerciales en un componente comercial autónomo, los algoritmos pueden cambiar para coincidir con la normativa de impuestos sin afectar de manera negativa en el resto de la aplicación.

Una ventaja más sutil de la *arquitectura de tres-niveles* resulta del hecho de que la lógica de aplicación ya no se enlaza directamente a las estructuras de base de datos o a un DBMS en particular. Los componentes de aplicación individuales funcionan con sus propias estructuras de datos encapsuladas, que pueden corresponder a una estructura de base de datos o pueden, ser una estructura de datos derivada de una serie de fuentes de datos distintas. Cuando se comunican los objetos de la aplicación, sólo necesitan enviar los parámetros de datos tal y como se especifican en la interface abstracta en lugar de en registros de la base de datos entera, reduciendo así el tráfico en la red. Los componentes de acceso a los datos son los únicos componentes de aplicación que comunican directamente con la base de datos. Podría ocurrir que una base de datos pudiera pasarse totalmente de un DBMS a otro sin afectar de manera negativa a toda la aplicación: sólo la lógica de acceso a los datos necesitaría modificarse.

Cabe aclarar que la técnica de diseño *cliente/servidor de tres-niveles*, no significa que se esté utilizando alguna plataforma de cómputo en particular, sino que más bien se tiene bien definido de la existencia de la lógica de presentación, separado de la lógica de negocio y del almacenamiento de los datos.

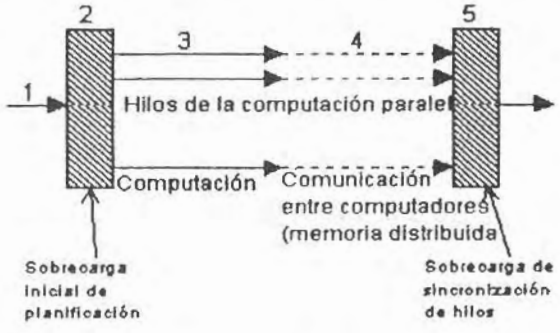
Uno de los componentes importantes de ésta infraestructura es el concepto de *middleware*, representado en la figura de arriba como un "*Menu de servicios*". Por lo tanto, debe identificarse cuál es el tipo de middleware que más le conviene a la empresa para crear un estándar de intercomunicación lógica aplicativa, que permita que los diferentes componentes sean fácilmente conectables (interoperabilidad) entre sí. Entre los diferentes tipos de middleware que existen están: ODBC, DCE RPCs, TPMs, MOMs, ORBs (CORBA, DCOM).

En la siguiente sección vamos a retomar nuestro estudio sobre los hilos, afín de integrar los conceptos de asíncrona del multihilado a la arquitectura que aquí se propone y que expondrá mas adelante.

4.2.2 Computación multihilo.

Como se describió en el capitulo 1, el concepto de *multithreading* o *multihilado* es una extensión de los conceptos de multitarea y multiproceso; también sabemos que los niveles de sofisticación, en asegurar la coherencia de los datos y preservar el orden de los eventos, se van incrementando progresivamente desde el modelo de programación simple, pasando por la multitarea, la multiprogramación, el multiproceso, hasta llegar al multihilado. Se necesitan desarrollar mecanismos especiales para gestión de memoria y protección con el fin de garantizar la corrección e integridad de los datos en las operaciones paralelas de los hilos. El multithreading o multihilado necesita que el procesador de la máquina esté diseñado para manejar múltiples contextos simultáneamente mediante un cambio de contexto base.

El modelo de computaciones paralelas multihilo fue descrito por Bell (1992), ver [Bacon, 1994] y se presenta en la siguiente figura:



Modelo de computación multihilo.

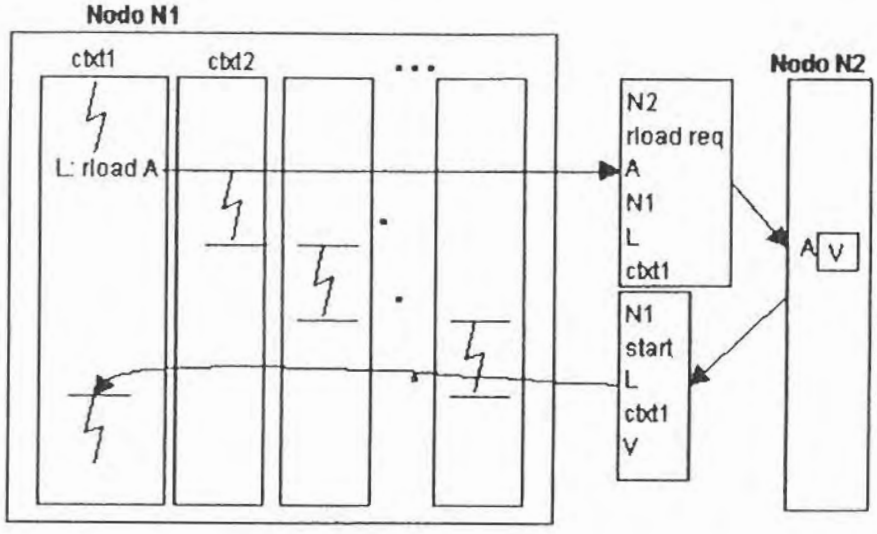
La computación comienza con un hilo secuencial (1), seguido por una planificación (2) donde los procesadores inician los hilos de la computación (3), a través de mensajes entre computadoras que actualizan las variables entre los nodos donde existe memoria distribuida (4), y finalmente mediante sincronización (5) antes del comienzo de la siguiente unidad de trabajo paralelo.

El periodo de sobrecarga de comunicación (3) inherente en las estructuras de memoria distribuida está normalmente distribuido en toda la computación y está solapado completamente. La sobrecarga del paso de mensajes debido a las llamadas *receive* y *send* en los multicomputadores puede reducirse mediante hardware especializado operando en paralelo con la computación. Las llamadas de paso de mensajes (4) y sincronización (5) son no productivas. Son necesarios mecanismos rápidos para reducir u ocultar estos retrasos.

4.2.2.1 Multihilado y asincronía.

La solución a los problemas de asincronía consiste en multiplexar varios hilos: cuando un hilo realiza una petición de carga remota, el procesador comienza a trabajar en otro hilo. Para ello, el coste de cambio de contexto de hilo debe ser mucho menor que la latencia de la carga remota, de lo contrario es preferible que el procesador espere la respuesta de la carga remota.

Cuanto mayor es la latencia entre nodos más hilos son necesarios para ocultarla eficientemente. Otro aspecto a tener en cuenta es que los mensajes deben llegar en orden. Así por ejemplo, supongamos que el hilo T1 realiza un carga remota; mientras llega el resultado podemos conmutar al hilo T2, que realiza otra carga remota. Entonces puede ocurrir que las respuestas no lleguen en el mismo orden, debido a que las peticiones pueden ser a nodos de distintas distancias, con distintos grados de congestión, nodos cuyas cargas se diferencian enormemente.



La solución del multihilado.

Una forma de abordar este problema es asociar a cada carga remota y cada respuesta un identificador del hilo apropiado, de tal forma que pueda ser reestablecido a la llegada de la respuesta. A estos identificadores se les llama *continuaciones sobre mensajes*.

Se necesita un gran espacio de nombres de continuación para poder manejar un número adecuado de hilos a la espera de respuestas remotas, Se puede realizar un estudio mas profundo sobre el tema, pero no es el objetivo de este capitulo, por lo que se recomienda ver [Gerlhof et al, 1994] y [Bacon, 1994].

El multihilo ofrece una solución para las cargas remotas y posiblemente para la sincronización de cargas. Normalmente, las dos aproximaciones se pueden combinar para resolver ambos tipos de problemas de acceso remoto.

4.2.2.2 Sistemas operativos con gestión de hilos.

La estructura del núcleo (kernel) de la mayoría de los sistemas operativos que dominaban el mercado hasta hace unos años, era una estructura monolítica que podía o no permitir la multitarea, pero no solía permitir el multiproceso real en múltiples procesadores, a pesar de que empezaban a aparecer nuevas arquitecturas multiprocesador. Normalmente, esto sistemas operativos se basaban en el concepto de proceso, como entidad planificable, con un solo hilo de control. Este era el caso de UNIX, que no permitía múltiples hilos de control dentro de un mismo proceso.

Con la aparición de nuevas arquitecturas multiprocesador y multicomputador era necesario que se desarrollaran nuevos sistemas operativos que pudiesen explotar las ventajas de estas nuevas arquitecturas, de cara a mejorar el rendimiento de las aplicaciones y de los sistemas. Estos nuevos sistemas operativos proporcionan soporte de hilos a nivel del kernel, soporte de multiprocesadores y en ocasiones de multicomputadores, convirtiéndose, en algunos casos, en los nuevos *sistemas operativos distribuidos* o al menos *sistemas operativos de red.*

La planificación de múltiples hilos en un conjunto de procesadores disponibles se llama *gang-cheduling* o *planificación en grupo.* Para implementar un kernel multihilo se emplean semáforos o bloqueos cíclicos para separar múltiples hilos con conjuntos diferentes de recursos kernel. El objetivo es maximizar el grado de paralelismo en las operaciones del núcleo. Los hilos no relacionados pueden ejecutarse en procesadores distintos sin contención de recursos, y la utilización de la CPU debe aumentar de forma notable, sin que la sobrecarga añadida para su gestión sea excesiva.

Existen muchos sistemas operativos actuales en cuyo desarrollo se tuvo en cuenta la implementación de hilos y su soporte por el kernel, mientras que en otros ha sido un añadido posterior (lo que en muchos caso ha obligado a rediseñar el sistema operativo). Algunos ejemplos son Mach, OSF/1, Amoeba, Windows NT/95, SunOS 5.x, Linux 2.x, OS/2, NeXTStep, AIX, IRIX. Los hilos se presentan como el paradigma de los nuevos sistemas y su influencia está siendo notable, tanto en plataformas hardware como software. Los nuevos sistemas operativos los han adoptado como parte integral en sus núcleos, y cada vez son más las aplicaciones multihilo que se escriben de forma específica para entornos distribuidos y paralelos. Además la aprobación del estándar POSIX 1003.1c parece que tiende a unificar los API de programación de los distintos fabricantes, lo que permitirá desarrollar aplicaciones más portables entre distintos sistemas.

4.3 Propuesta de Arquitectura para cómputo distribuido y paralelo.

Como ya hemos visto en los capítulos 2 y 3, las unidades que integran a los distribuidos y paralelos, pueden ser vistos como capas o niveles de *componentes*, los cuales abstraen sus propiedades, ofrecen una interfaz común para acceder a sus servicios, permiten ocultar su implementación y coordinarlos. Esto ocurre además independientemente del paradigma de programación escogido, ya sea mediante objetos, actores, lenguajes de coordinación, etc.

Pues bien, la idea común es la de ir incorporando al sistema nuevos componentes que añadan las funciones requeridas en cada caso. Este es un mecanismo conocido y aceptado en [Chandy et al, 1997] y [Jennings et al , 1998], y buenos ejemplos son las arquitecturas regidas por leyes para Linda [Minsky et al, 1995], el control de tiempo real para los actores, Ver apéndice A.

Cada uno de esos niveles (componente) actúa como un envolvente (*wrapper*) activo que captura y modifica la entrada y salida de datos del programa, y lo dota de la funcionalidad pedida. Sin embargo, las capas de componentes a las que nos referimos aquí no se comportan como simples filtros, pues su comportamiento no es pasivo: como resultado de la llegada de un mensaje, la capa puede enviar uno o más mensajes al sistema, esperar sus respuestas y construir con ellas el mensaje final que pasará al componente.

Una gran ventaja de este enfoque es la independencia que se consigue entre el sistema y cada una de las capas, pues cada una realiza su función de forma independiente e incremental, permitiendo así una gran modularidad y flexibilidad.

También permite "descargar" al componente de aquellas funciones que no son propiamente suyas. Por supuesto, las propiedades no tienen porqué implementarse en todos los componentes de un sistema, ni suponer que todos han de cumplirlas; según nuestra filosofía, cada componente es libre de escoger su propia configuración.

4.3.1 La Arquitectura Teórica.

La siguiente definición muestra los elementos que caracterizan a una propiedad P cualquiera.

Definición: Una propiedad P definida en un Sistema S está caracterizada por una cuaterna P = (C,B,E,R).

donde C es un conjunto de Componentes de S que implementan esta propiedad; B es un conjunto de Controladores, habiendo uno asociado a cada componente de C; E es un conjunto de Eventos de comunicación relativos a la propiedad P; y R es el conjunto de reglas o Estrategias que definen los posibles comportamiento de los controladores de B ante los eventos de E.

En esta definición, los *Controladores* son los procesos que implementan las capas. Ellos capturan todos los mensajes entrantes y salientes del componente (*Eventos,*) y los modifican según la *Estrategia* que implementen. Las *estrategias* se definen y especifican en forma global cuando se define una propiedad, y cada controlador escoge la estrategia apropiada cuando se particulariza para un componente concreto. El conjunto R contiene a todas las posibles estrategias para todos los eventos relativos a la propiedad.

En general, las propiedades no controlan la ocurrencia de los eventos, sino que sólo toman decisiones sobre lo que hacer con ellos. Y por supuesto, dos o más propiedades pueden componerse funcionalmente para formar nuevas propiedades bajo este esquema. La operación de composición permite encadenar unas propiedades a otras, y dota al conjunto (P) de todas las propiedades de estructura única.

4.3.2 La Arquitectura Computacional.

Para desarrollar la arquitectura teórica utilizaremos una arquitectura computacional que permita construirlo. El modelo escogido es muy simple y general, pues incorpora sólo la funcionalidad necesaria para nuestros propósitos.
El objetivo de esto es poder implementarlo sobre cualquier tipo de sistemas y paradigmas como (Linda [Minsky et al, 1995], Actores ver apéndice A, CORBA [Vinoski, 1997], DCOM [Wang et al, 1997]).

La arquitectura computacional, aquí propuesta está basada en procesos que se comunican a través de colas de mensajes (buzones). Cada componente es un proceso concurrente multihilado que tiene estados. Cada estado está compuesto por un conjunto de asignaciones de valores a las variables definidas en tal componente. La comunicación entre componentes es asíncrona, y se realiza mediante el envío de mensajes a los buzones.

Para cada componente existe definido su *dominio,* que no es sino la dirección de la máquina (o red de máquinas) en donde decide establecerse. Como nombres de dominio vamos a considerar los nombres de dominio de correo Internet, precedidos por el carácter "@". Ejemplos son "@angel.umar.mx" o "@utm.mx".

Cada buzón tiene una dirección única global (IP), que viene dada por una expresión de la forma "mb@dominio", siendo "mb" el nombre asociado a tal buzón, que debe ser único dentro del dominio "@dominio". Si se omite el dominio, por defecto se utiliza el del proceso que referencia al buzón.

Si sólo se indica el dominio, se hace referencia a *todos* los buzones que pertenecen a él en ese momento. Esto permite realizar la difusión de mensajes.

En esta arquitectura cada componente tiene asociado un buzón, y existen dos operaciones básicas de comunicación: *Envía* y *Recibe*. La primera envía un mensaje a un buzón dado, y la segunda permite al componente leer un mensaje de su buzón. El envío de mensajes no es bloqueante, tampoco la recepción.

Los mensajes están compuestos por campos, conteniendo entre otros la dirección del buzón destino (m.IPDestino), la dirección del buzón origen (m.IPOrigen), el "subject" (m.Subj) o campo de información libre que puede ser utilizado para resumir el objetivo del mensaje, un número de referencia del mensaje (m.Ref) y por último la información propia del mensaje (m.Info).

Es importante observar que el uso de este modelo permite aplicar también métodos de verificación y razonamientos formales, como se describe en [Chandy et al, 1997] tanto sobre los componentes como sobre sus composiciones.

4.3.3 Los Controladores - Brokers.

En este modelo los controladores son, como hemos mencionado anteriormente, procesos especiales que *envuelven* al componente modificando su comportamiento. Ellos capturan los eventos entrantes y salientes del componente mediante accesos a sus colas, y los tratan de acuerdo a la estrategia que implementen. Esta idea tiene analogía con los adaptadores para objetos [Tsichritzis et al, 1992], con el modelo para actores y agentes [ver Apendice B].

Para especificar los controladores vamos a utilizar un esquema genérico, que iremos derivando o *especializando* progresivamente: primero para especificar los controladores que implementan cada propiedad, y luego cada uno de estos habrá de especializarse en el momento de particularizarse a trabajar con un componente concreto.

Para especificar los controladores es necesario definir los tipos de mensajes que aceptan y las operaciones que soportan para manejarlos. Tres son las operaciones que serán invocadas por el sistema cuando se produzcan determinados eventos: *Entrega*, *Recibir* y *Respuesta*.

La primera permite capturar los mensajes salientes, y será invocada por el sistema cada vez que el componente desee enviar un mensaje a un buzón. La segunda función permite al controlador capturar los mensajes entrantes, y será invocada por el sistema cada vez que se intente encolar un mensaje en la cola de mensajes asociada al controlador. Por último, la función *Respuesta* permitirá al componente conocer cuándo un mensaje que se envía no recibe respuesta; si bien es cierto que una propiedad cualquiera no tiene porqué establecer eventos de fuera de tiempo (timeouts), en general los controladores sí deben saber tratar estas condiciones excepcionales, que no pueden olvidarse en los sistemas distribuidos y paralelos.

El sistema ofrece dos operaciones a los controladores para poder completar sus acciones: una vez tratado un mensaje entrante, el controlador lo depositará en la cola de mensajes del componente (EntraC). Análogamente para mensajes salientes, el controlador añadirá a la cola del componente (SaleC) el resultado del tratamiento que ha realizado sobre el mensaje.

Con todo esto, el esquema más simple que podemos tener de un controlador es el siguiente:

```
class Controlador_ORB {
   public void Entrega (Msg m) { // Captura mensajes salientes
   EntraC.Cola(m);        // y los deja pasar sin modificarlos.
 }
  public void Recibir (Msg m) { // Captura mensajes entrantes
   SaleC.Cola(m);              // y los pasa al componente directamente.
}
public void Respuesta (Msg m) { // Captura condiciones de timeout. }
                  // No hace nada al respecto.
}
```

A partir de este esquema general se podría ir "derivando" los controladores que implementan cada propiedad mediante sucesivas especializaciones.

Obsérvese que en el caso de tener varios controladores enlazados consecutivamente (por tratarse de una composición de propiedades), el fin de cada operación EntraC.Cola desencadenará una operación *Recibir* en el siguiente controlador y, de igual forma, el fin de cada operación SaleC.Cola hará que se invoque a la función *Entrega* del siguiente controlador. En cuanto a la ocurrencia de timeouts, la función que realiza su tratamiento será invocada en todos los controladores asociados a un componente dado.

4.3.4 Independencia de la Arquitectura.

Los sistemas distribuidos son, en muchos aspectos, un fiel reflejo de nuestro mundo real. Los servicios que necesitamos no están siempre disponibles, aparecen nuevos servidores de vez en cuando, y otros desaparecen. Nosotros funcionamos con dos tipos de informaciones: *las de confianza o conocidas, y las existentes en el mercado o desconocidas*. Las primeras son las que hemos utilizado para resolver problemas que hemos tenido. De la existencia de las segundas nos enteramos a través de preguntas que hacemos o de publicidad que recibimos. Cuando necesitamos algún servicio usamos una solución conocida, a menos que no funcione en ese instante o hayamos oído de otra mucho mejor. Incluso si no conocemos ninguna ruta alternativa.

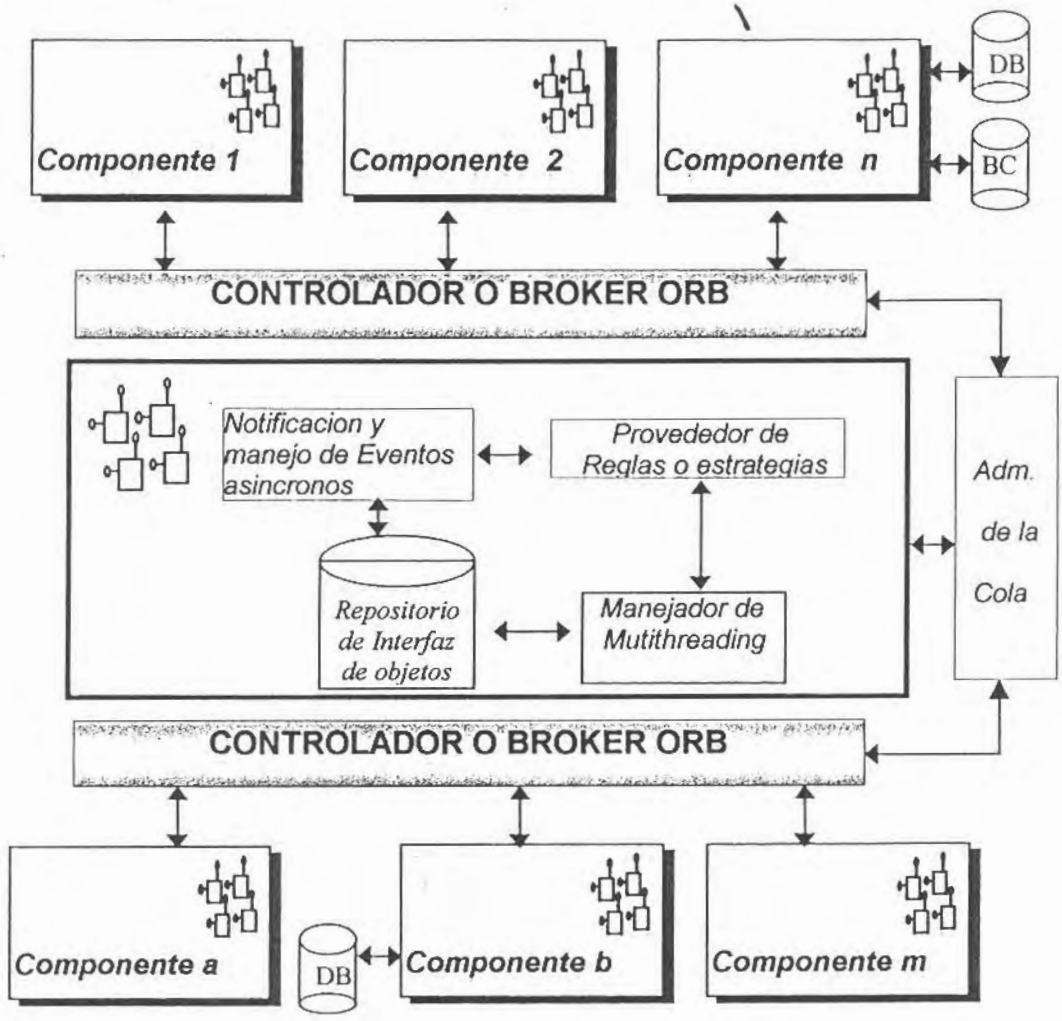
La facultad de buscar los servicios que necesitamos, junto con la capacidad de decidir cuál escoger en cada momento, nos permite tener *Independencia*. En este modo de funcionar nos apoyaremos para especificar los controladores de esta propiedad.

De acuerdo a nuestro modelo teórico, lo que sigue es una descripción de los elementos que componen la cuaterna que caracteriza la propiedad $P = (C, B, E, R)$.

Los eventos relativos a esta propiedad van a ser todos aquellos mensajes de petición de prestación de servicio a otros componentes del sistema, junto con la *publicidad* recibida sobre nuevos servicios. Las peticiones de prestación de servicio recibidas por el componente no se consideran eventos relativos a esta propiedad. Son las diferentes formas en las que el controlador de un proceso toma decisiones sobre los servicios: *cómo y a quién contratarlos y cómo mantener actualizada la información disponible*.

Cara a tomar una decisión, podemos seguir muy distintas estrategias según el carácter que queramos imprimir a nuestro componente. Así, un programa conservador intentará una solución (esto es, un par cliente/servidor) conocida siempre que pueda. Sólo decidirá cambiar de solución cuando ésta ya no le valga, haya dejado de funcionar, o exista otra de la que ha "oído" que la supera. Un programa más atrevido escogerá la opción que sea más atractiva entre todas las que conoce o ha oído hablar, no dando demasiado peso a su experiencia anterior.

Con la publicidad también existen varias alternativas: o bien se tira toda, y cuando no dispongamos de solución conocida entonces preguntamos a los componentes de nuestro dominio; o bien se va leyendo todo lo que llegue y actualizando nuestra lista de soluciones conocidas.



Arquitectura basada en componentes.

4.3.5 Mas sobre los controladores - Brokers.

En el caso de la Independencia, los controladores van a ser los encargados de buscar, gestionar y administrar los servicios que el componente necesita, y por tanto también los denominaremos Brokers. Al descargar estas labores en los controladores conseguimos que el componente sea independiente frente a los cambios en su entorno.

El uso de un controlador individual para cada componente permite su particularización y especialización de acuerdo a sus preferencias, y evita la dependencia de un servidor centralizado o común para muchos componentes. Obsérvese que nosotros no estamos en contra de estos servidores centralizados; si existen, el controlador debe utilizarlos para evitar duplicidad de trabajos y responsabilidades. Sin embargo, pensamos que los controladores deben ser implementados individualmente para conseguir la deseada independencia.

Para entender el concepto de controlador respecto a la propiedad de Independencia, una analogía clara aparece cuando pensamos en algunos gestores de servicios de nuestro mundo real: *asesores de inversiones, agencias de viajes o corredores inmobiliarios*. Otra idea interesante es la de la secretaria eficiente, que se ocupa de los vuelos y alojamientos de su jefe. Ella conoce sus preferencias y está al tanto de descuentos en hoteles, ofertas de viajes y huelgas aéreas para conseguir siempre las mejores combinaciones y precios, mientras que él puede concentrarse plenamente en sus reuniones.

Para llevar a cabo su misión, los controladores implementan los siguientes mecanismos:

- Mantenimiento de una lista ponderada de *Soluciones* conocidas.
- *Preguntas* a todos los componentes (broadcast).
- Recepción y distribución de *Publicidad* asíncrona.

El primero de ellos consiste en una lista de los servicios que el controlador conoce. Esta lista se mantiene en torno a una función que asocia a cada solución un peso (o coste) y que permitirá al controlador comparar soluciones y tomar decisiones sobre el servicio a contratar.

Respecto a las preguntas difundidas a todos los componentes, éstas son necesarias cuando el componente pide al controlador algún servicio no conocido por éste, aunque también pueden ser utilizadas por el controlador para actualizar periódicamente su lista de soluciones.

El sentido en el que utilizamos aquí la palabra difusión (*broadcast)* no implica que los mensajes han de llegar necesariamente a todos los componentes del sistema, ni que todos tengan que responder. De hecho, el componente no sabe cuántos otros componentes hay en un momento dado, ni tampoco le interesa ese dato.

Por otro lado, el concepto de *publicidad* aparece como la emisión y recepción de mensajes *asíncronos* que permiten estar al tanto de las novedades y cambios que se producen en el sistema. Cuando se recibe una *publicidad* se compara con la información que tenemos y se le asigna un peso, pudiendo incorporarse a la lista de soluciones mencionada anteriormente. Obsérvese que no se trata sólo de la correduría *(trading)* de los servicios, sino que la idea clave es aquí la capacidad de decisión necesaria (en forma de función de peso) que posee el controlador para elegir siempre la mejor opción.

4.3.6 Trabajos Relacionados.

La idea de este trabajo surge a partir de las ideas originales expuestas en [Hewitt et al, 1983], [Chandy et al, 1997], [Minsky et al, 1995] Los primeros citan el tiempo real, la asincronía y la autonomía como piezas claves en los sistemas abiertos y distribuidos, *ver Apendice B*, pero sólo tratan en su trabajo los dos primeros conceptos, ignorando la autonomía. Por otro lado en [Minsky et al, 1995] se presentan las arquitecturas regidas por leyes, esquema que se ha generalizado.

Respecto a los métodos de capas, éstos son conocidos y aceptados, también se cita el control de tiempo para actores. Sin embargo, las capas, que se manejan en la arquitectura, pasan a ser procesos más activos, con capacidad para interrogar a otros elementos del sistema y tomar decisiones dependiendo de las preferencias del componente, más al estilo de los adaptadores ORB´s. Este modelo de procesos que capturan y modifican los eventos de comunicación de sus componentes es completamente activo.

En cuanto a las propiedades definidas para los componentes, existe abundante bibliografía en el campo de la Inteligencia Artificial, en donde se discute acaloradamente sobre las posibles definiciones de *agentes autónomos* y de las propiedades que éstos deben cumplir [Jennings et al, 1998].

El seguimiento computacional de procesos y buzones escogido no trata de ser novedoso, sino más bien simple y general con el objeto de poder implementarlo sin dificultad sobre cualquier otro modelo o paradigma computacional, como puede ser CORBA, DCOM, o Actores. Y en cuanto a la propiedad de Independencia y la gestión y administración de los servicios que un componente necesita, el concepto de correduría de servicios *(trading)*, es conocido e incluso forma parte de CORBA o DCOM. Sin embargo, esta propiedad no sólo trata de eso, sino que su importancia reside en la capacidad de decisión que poseen los controladores.

4.4 Conclusiones.

Sea definido en el presente trabajo un conjunto de propiedades para los componentes de los sistemas distribuidos, junto con una arquitectura general que permite su definición, especificación e implementación.

El disponer de una arquitectura general no sólo ofrece una plataforma para definir formalmente las propiedades mencionadas, sino que permite razonar sobre ellas e incorporar nuevas propiedades fácilmente.

A hora sabemos que la computación distribuida basada en objetos es muy compleja y requiere una infraestructura sofisticada y robusta, ya que es necesario coordinar las aplicaciones en un ambiente heterogéneo, por ello hay que tratar con la interoperabilidad de los elementos que integran al sistema, proporcionar mecanismos para estructurar las interacciones, y dar flexibilidad en su definición.

El enfoque de los objetos distribuidos en este trabajo esta dado en términos del sistema de capas o niveles de componentes, que al exponer ***interfaces*** de uso se están ofreciendo *diálogos sobre temas particulares*, por lo que pudieran emplearse heurísticas de diseño orientado a objetos para la programación de protocolos. Correspondientemente, preguntar si un dialogo esta soportado es buscar una interfaz de uso en un objeto, por lo que los mecanismos de objetos distribuidos pueden emplearse en modelos de comunicación más elaborados. Por otro lado, los beneficios de un sistema multi-nivel pueden resumirse así:

- Reutilización y posibilidad de compartir objetos para que el tiempo de recodificación sea mínimo
- Interface abstracta para mayor flexibilidad.
- Mantenimiento de sistema más fácil.
- Los componentes concurrentes de un programa pueden ejecutarse de forma paralela en un multiprocesador.

Se ofrece una alta productividad tanto al diseñador(es), como al desarrollador(es) del sistema, a través de una Infraestructura de informática distribuida que permite transparencia, seguridad, capacidad de dimensión, fiabilidad y disponibilidad. Podemos afirmar que la próxima generación de sistemas de información está basada en modelos de componentes que ofrecen la flexibilidad en el rendimiento y escalabilidad.

Capitulo 5

IMPLEMENTACIÓN DE UN PROTOTIPO DE SISTEMA DE TOMA DE DECISIÓN DISTRIBUIDO.

En este capitulo se aborda la implementación basada en la arquitectura de objetos componente aquí propuesta, de un prototipo de sistema mínimo de toma de decisión distribuido de recuperación y presentación de información con mecanismos de inferencia basados en lógica difusa, pues se pretende tratar adecuadamente la imprecisión y la incertidumbre que suelen acompañar a los hechos y al conocimiento en la toma de decisiones, con relación al sistema orientado al desarrollo del estado de Oaxaca, en el nivel educativo y socioeconómico.

En las primeras secciones de este capitulo se expondrán los conceptos, técnicas y herramientas de diseño de lógica difusa, luego se explicaran el diseño del sistema de toma de decisiones en lógica difusa; cabe mencionar que la representación del conocimiento se basará en sentencias condicionales difusas. Posteriormente se explicara, como se adapta y distribuye el sistema difuso en la arquitectura de objetos componente, propuesta en este trabajo. Finalmente se mencionan las conclusiones y trabajo futuro.

5.1 Introducción.

La Inteligencia Artificial (IA) tiene como objetivo de estudio la comprensión y la construcción de entidades inteligentes. Dichas entidades son generalmente sistemas computacionales que tienen cierta capacidad de emular un comportamiento racional, a los que denominaremos "sistemas inteligentes". Podemos definir a un *Sistema Inteligente*, como una entidad capaz de decidir por sí misma que acciones llevará a cabo para alcanzar sus metas basándose en sus percepciones, conocimientos y experiencias acumuladas. Así que un sistema inteligente, de acuerdo con [Jennings et al, 1998], sera capaz de:

a) Ser autónomo y tomar las decisiones correctas al resolver un problema.
b) Poseer una motivación u objetivos bien definidos.
c) Aprender cosas nuevas, ya sea por ensayo-y-error, observación, razonamiento y/o instrucción.
d) Percibir y modificar interactivamente su entorno.

Durante muchos años la investigación sobre la inteligencia artificial se ha orientado hacía las aplicaciones independientes con un conocimiento determinado y con una meta especifica. Estas aplicaciones se desarrollan en un ambiente estático y sus actividades principales son las siguientes: *recopilar información, planear y ejecutar algún plan para lograr su meta*. Sin embargo, esta metodología ha resultado insuficiente debido a la inevitable presencia de un número de entidades que *cooperan en el mundo real* (Agentes, - Puede ver una rápida referencia de agentes computacionales, en el *Apéndice B)*. La Inteligencia Artificial Distribuida (Distributed Artificial Intelligence - **DAI**) es un subcampo de la IA, que intenta construir un modelo del mundo poblado por entidades inteligentes (agentes), que interactúan por medio de la cooperación o la coexistencia. Los sistemas reales, normalmente son tan complicados y contienen tanta información que es mejor reducirla a diferentes entidades cooperativas para lograr mayor eficiencia (modularidad, flexibilidad, y un tiempo más rápido de respuesta), para mas detalles pueden consultar [Jennings et al, 1998] y [Cockburn et al, 1996].

Podemos considerar a un sistema DAI, como una colección de procesos independientes, frecuentemente distribuidos en múltiples anfitriones o hosts enlazados por una red. Además, en los sistemas de redes modernos, es posible construir nuevos programas extendiendo los sistemas existentes - *reutilización*; un nuevo pequeño proceso debería tener la capacidad de enlazarse sin mayores problemas a las fuentes de información y herramientas existentes tales como los sistemas basados en conocimientos. Sabemos por [Chandy et al, 1997] y [Jennings et al, 1998], que durante el *proceso de la toma de decisiones* en conjunto cada *componente* debe obtener información acerca del ambiente externo y las conclusiones y decisiones derivadas por otros componentes.

5.2 Sistemas de apoyo a las decisiones.

Un *sistema de apoyo a las decisiones* es un sistema iterativo que da apoyo a los tomadores de decisiones, usando datos y modelos para resolver problemas no estructurados [Olson et al, 1992]. Estos sistemas están diseñados para proveer a los directivos información que puede usar para tomar decisiones en situaciones únicas, semiestructuradas y no estructuradas. A menudo, los ayuda a tomar una decisión a partir de varias alternativas. Esto incluye un conjunto de programas de cuantificación, modelación y simulación. Juntos, representan un grupo de herramientas matemáticas, estadísticas y gráficas. Esto permite al usuario manipular los datos en una amplia variedad de formas.

Podemos resumir el proceso de toma de decisiones, bajo los siguientes puntos de acuerdo con [Olson et al, 1992]:

- Una decisión se basa en la información (buena o mala) que poseamos.
- Es imprecindible que una decisión se apoye en una información que sea exacta, oportuna, completa, concisa y adecuada.
- No existe un sistema universal de soporte para la toma de decisiones.
- Mientras en un departamento puede prevalecer información objetiva y cuantitativa para sustentar sus decisiones, en otro departamento puede prevalecer el uso de información más incierta y subjetiva.
- Mientras que para algún departamento, su "asistente" es una simple hoja electrónica de cálculos, para otro departamento, tal vez sea un sofisticado sistema experto.

El área de la inteligencia artificial que mas desarrollo ha tenido son los *sistemas expertos* que se centran en una área muy delimitada (e.g., diagnóstico médico), y donde la experiencia y juicio del *experto humano*[i] es extraída, modelada y representada en la *base de conocimiento, según* [Ignizio, 1991].

La tarea de un experto humano es la de resolver problemas dentro de un dominio específico de conocimiento. Un sistema experto es un programa basado en el conocimiento que no intenta imitar la estructura de la mente humana ni los mecanismos para inteligencia general sino más bien, llegar a soluciones que sean muy parecidas a las del experto humano. No debemos confundirnos con sistemas de modelado cognoscitivo que pretenden seguir el mismo proceso que el humano para la resolución de problemas.

De acuerdo con [Russell et al, 1996] y [Negrete, et al, 1996], un sistema experto *es un programa de computadora que utiliza conocimiento y métodos de inferencia para resolver problemas en un dominio específico del conocimiento y que requieren de una pericia humana significativa.*

Los primeros sistemas expertos que se construyeron (y los más conocidos) son los sistemas basados en reglas. El conocimiento en estos sistemas está estructurado en forma de reglas del tipo *si-entonces* (if-then). Esta clase de sistemas son muy eficientes para resolver problemas donde ***no existe información incompleta*** **ó donde el conocimiento es categórico**. *MACSYMA* y *XCON* [Russell et al , 1996] y [Negrete, et al, 1996], son dos sistemas de este tipo.

[i] No necesariamente se requiere de un experto humano para adquirir el conocimiento, en ocasiones hay varios expertos o simplemente no existe tal experto, y se tiene que obtener información de libros y registros estadísticos de procesos realizados.

El primero encuentra soluciones a integrales y el segundo configura equipo de cómputo. En ninguno de estos sistemas requerimos o necesitamos representar incertidumbre. No tiene sentido hablar de soluciones de integrales con 70% de certeza ó de cables y drives con un 56% de probabilidad de que se conecten juntos.

Sin embargo sabemos que el conocimiento humano está expresado generalmente de manera incierta. Un experto resuelve problemas donde el conocimiento que se tiene no es del todo seguro ni tampoco se tienen todos los datos necesarios para la solución del problema, por lo que el *experto humano* debe hacer algunas suposiciones y entonces así ser capaz de ofrecer soluciones a los problemas presentados.

Debido a que los primeros sistemas expertos fueron los sistemas basados en reglas, éstos fueron los primeros que se adaptaron para la representación de conocimiento incierto. *Cabe aclarar que en general se tienen varios significados cuando hablamos de incertidumbre.* En algunas ocasiones nos referimos a ella cuando tenemos errores en las observaciones. En otras ocasiones no se tiene toda la información acerca del problema y cuando llega más información entonces puede darse el caso de poder cambiar las conclusiones a las que se había llegado, es decir, podemos retractarnos de las conclusiones a la llegada de nueva información. El otro significado de incertidumbre es cuando tenemos vaguedad en algún concepto (*como el de opinar acerca de la estatura de una persona*), mas detalles, ver en [Negrete, et al, 1996] y [Cockburn et al, 1996].

En el desarrollo del presente trabajo, aclaramos que el significado que le damos a incertidumbre es aquel cuando tenemos vaguedad en algún concepto (*como el de opinar acerca de la estatura de una persona*).

Por otro lado sabemos que, el campo de la medicina es un buen ejemplo para ilustrar el problema de representar conocimiento incierto. Un médico decide si ciertos síntomas se encuentran presentes en el paciente y si esos síntomas están asociados con cierta enfermedad ó enfermedades. El especialista nunca está del todo seguro de sus conclusiones aunque esto no quiere decir que no resuelva los problemas que se le presenten (o por lo menos una buena parte de ellos).

Para que un sistema experto pueda razonar bajo incertidumbre, se le debe dotar de ciertas capacidades. Muchos han sido los enfoques para incorporar incertidumbre en sistemas expertos, teniendo todos sus detractores y defensores. Algunos de estos enfoques son, de acuerdo con [Negrete, et al, 1996] y [Russell et al, 1996]:

- Teoría matemática de la evidencia (Dempster-Shafer)
- Lógica no-monótona
- Lógica difusa (fuzzy logic)

La manera de representar incertidumbre en sistemas expertos basados en reglas es utilizando la *teoría matemática de la evidencia*. En algunos casos tenemos la evidencia de que una regla implica cierta conclusión con cierta probabilidad. Si queremos determinar la probabilidad de cada proposición en la regla pero no podemos obtener los valores de probabilidad necesarios para hacerlo, entonces recurrimos a las llamadas *funciones de creencia*. Estas *creencias* en las proposiciones nos dicen con qué grado de certeza podemos creer en el peso de la evidencia que a partir de una regla se le asigna a la conclusión. Para más detalles ver [Negrete, et al, 1996].

Otro enfoque es el de las *lógicas no-monótonas*. Es un método no numérico para representar incertidumbre. Un humano llega a una conclusión a partir de sus creencias ó conjunto de creencias dependiendo de la información que tenga acerca de un determinado problema. Conforme tenga mayor información, las creencias en las conclusiones pueden cambiar. Es decir, hay veces en las que el humano debe retractarse de sus creencias para así poder plantear otra solución al problema. No sucede así en la lógica clásica que es monótona, esto es, partimos de un conjunto de axiomas (los cuales asumimos que son verdaderos) y a partir de ellos inferimos sus consecuencias (teoremas), siendo éstos invariantes ante la introducción de nuevos conocimientos ó creencias, un estudio mas detallado acerca de estas teorías, puede obtenerse en [Negrete, et al, 1996], [Herrera et al, 1998] y [Russell et al, 1996].

Otra aproximación para representar conocimiento incierto en sistemas expertos, basado en reglas es otra extensión de la lógica clásica llama *lógica difusa* (*fuzzy logic*). Como mencionamos anteriormente, en el enfoque de la teoría matemática de la evidencia, asignamos un número real a cada proposición indicando así la cantidad de certeza con la cual creemos que esa proposición es verdadera en un caso determinado. Pero en la lógica fuzzy asignamos una función de *membresía* a cada proposición.

Esta función asocia un número real entre 0 y 1 que significa, no el grado de certeza de la proposición, sino *el grado de pertenencia* de esa proposición en un subconjunto fuzzy, mas detalles en [Herrera et al, 1998].

Brevemente ilustraremos esto con un ejemplo. Tenemos la siguiente regla:

Si el jugador es muy alto **entonces** existe una gran posibilidad de que juegue como defensa (en baloncesto por supuesto).

Podría quitar la palabra *gran posibilidad* y reemplazarla por un *número real* entre 0 y 1 y así estaríamos hablando de incertidumbre en la conclusión de la regla tal como la consideraría la teoría de la evidencia, la teoría de la probabilidad ó el enfoque mediante factores de certeza. Pero si notamos la premisa *el jugador es muy alto* no es nada exacta pero tampoco expresa incertidumbre alguna. Más bien *expresa el punto de vista del entrenador* que nos dice si para él, el jugador es muy alto ó no.

En este caso el *subconjunto fuzzy* al que hacíamos referencia anteriormente sería ***muy alto*** y la *función de membresía* podría estar definida como sigue:

$$\text{muy alto}(X) = \begin{cases} 0 & \text{si la altura}(X) < 1.70 \text{ mts.} \\ (\text{altura}(X) - 1.70 \text{ mts}) / .50 \text{ mts.} & \text{si } 1.70 \leq \text{altura}(X) \\ 1 & \text{si la altura}(X) > 2.10 \text{ mts.} \end{cases}$$

Además, si tenemos varios jugadores:

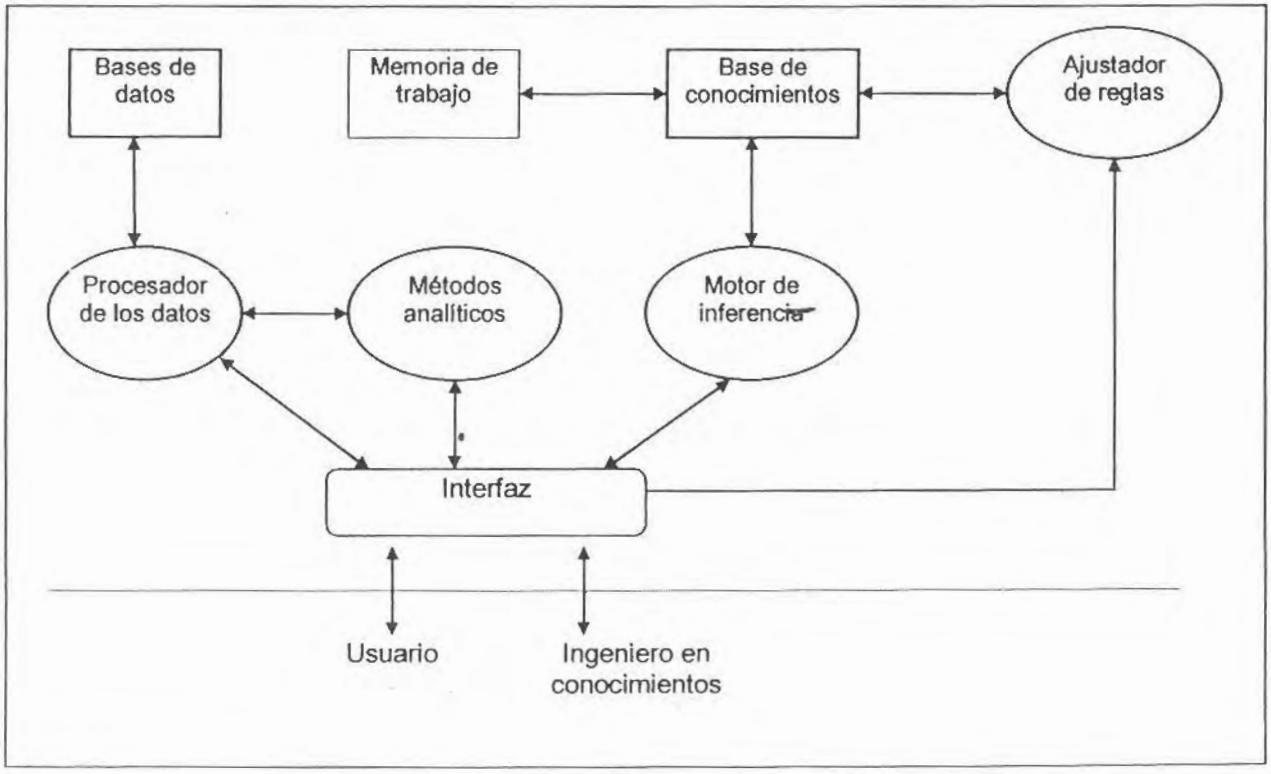
	Altura(mts)	grado de altura
Pancho	1.60	0.0
Carlos	1.75	0.1
Juan	2.00	0.6
Pedro	1.90	0.4
Manuel	1.85	0.3
Luis	2.15	1.0

Los números de la tabla, nos indican el grado de membresía de una proposición. Por ejemplo, el grado de membresía de la proposición *Carlos es muy alto* es de 0.1. Esto quiere decir que 0.1 indica el grado con el cual Carlos pertenece a la categoría *muy alto*.

Cabe señalar que la función ó funciones de membresía son definidas subjetivamente por el experto humano (en este caso por el entrenador). Esta metodología puede extenderse para manejar incertidumbre de varias maneras.

Una de ellas consiste en asociar con cada proposición ***p*** un valor $\mu_t(p)$ que indicará el grado de pertenencia de ***p*** al conjunto de las proposiciones verdaderas, [Cordón et al, 1997]. En la siguiente sección, vamos se va ha describir con mayor detalle, los conceptos básicos de la lógica difusa.

Por otro lado, actualmente los sistemas de apoyo a las decisiones han empezado a incluir sistemas expertos en su arquitectura. Cuando un sistema experto es incrustado en la arquitectura de un sistema de apoyo a las decisiones, se le denomina *Sistema de apoyo a las decisiones híbrido*. Esto permite combinar modelos cuantitativos y analíticos (i.e., algoritmos), y modelos cualitativos (i.e., heurística). La estructura de un sistema de apoyo a las decisiones híbrido se representa en la Figura 1.1[i].



Estructura de un sistema de apoyo a las decisiones híbrido

[i] Para un conocimiento detallado de cada uno de los componentes de la estructura de un sistema de apoyo a las decisiones híbrido, v. [Ignizio, 91] , en su capitulo 2, *Background.*

El principal enfoque de la representación del conocimiento del prototipo implantado en este trabajo de tesis, es el *sistema basado en reglas*. Se ha elegido este modelo debido a la flexibilidad y facilidad de representación del conocimiento. Además, que el ambiente de programación para sistemas basados en lógica difusa que se ha utilizado en la implementación del prototipo está basado en el modelo del sistema basado en reglas.

El prototipo a implementar en esta tesis, *no es un sistema experto*; pues se basa en elementos de los sistemas de apoyo a las decisiones y de herramientas provistas por la tecnología de sistemas expertos (i.e., es un *sistema de apoyo a las decisiones híbrido*).

5.3 Lógica Difusa como solución al problema de decisión: Teoría y Tecnología.

La lógica difusa ofrece una solución práctica y barata para el control de sistemas complejos o mal definidos. En oposición a su nombre, este método ofrece una rigurosa estructura para la solución de muchos tipos de problemas de decisión y control. Todo lo que realmente se necesita es un conocimiento práctico del comportamiento total del sistema. La Lógica Difusa trata de los principios formales del razonamiento aproximado, en el que, el razonamiento preciso, es un caso particular. Esto es, la característica esencial de la Lógica Difusa, al contrario de lo que sucede en la clásica, permite modelar de alguna manera el *razonamiento impreciso* que, por otro lado, es la base del pensamiento humano, [Herrera et al, 1998], [Gulley, 1997].

Un Sistema de Lógica Difusa puede entenderse como un sistema no lineal de múltiples entradas concretas y múltiples salidas concretas, cuya estructura interna se muestra en la figura 5.2, ver mas abajo. La idea básica es poder dar una respuesta aproximada a una pregunta, en función de unos hechos previamente almacenados y que pueden ser inexactos, incompletos o poco fiables, de acuerdo con [Siler, 1996] y [Herrera et al, 1998].

Vamos a considerar, los grandes inconvenientes que plantea la Lógica clásica para llevar a cabo este tipo de *razonamiento, y estos son, de acuerdo con [Gulley, 1997] y [Siler, 1996]*:

- En Lógica Bivaluada, una proposición *p* es verdadera o falsa. En Lógica Multivaluada, una proposición puede ser verdadera, falsa, o tener un valor de verdad intermedio de entre los de un conjunto finito de posibles valores de verdad. En Lógica Difusa, se permite que los valores de verdad sean subconjuntos difusos definidos, generalmente, en el intervalo [0;1].

- En Lógica Bivaluada, los predicados han de ser, necesariamente, precisos, en el sentido de que no pueden describir subconjuntos difusos sobre el universo del discurso. En Lógica Difusa, los predicados pueden ser precisos (*madre, mortal, hombre*) o bien difusos (*alto, cansado, guapo,...)*.

- En las Lógicas Bivaluada y Multivaluada, solo se permiten dos cuantificadores: alguno y todos.

- En Lógica Difusa, pueden utilizarse numerosos cuantificadores (*la mayoría, pocos, muchos, normalmente, alrededor de 4*), que son vistos como números difusos que expresan la cardinalidad de algún conjunto difuso.

- La Lógica Difusa permite representar y manipular modificadores (difusos o no) de predicados, tales como: *ligeramente, mucho, un poco, bastante,...*

- En Lógica Bivaluada, una proposición puede ser cualificada asociándole un valor de verdad verdadero o falso, mediante un operador modal como *posible o necesario* o bien mediante un operador intencional como *creo, se, pienso, etc.*

- En Lógica Difusa, hay tres tipos de cualificación de predicados: Cualificación de la verdad, por ejemplo "No es muy cierto que Pedro sea alto", Cualificación de la probabilidad, por ejemplo *"Es poco probable que Pedro sea alto" y Cualificación de la posibilidad, por ejemplo "Es prácticamente imposible que Pedro sea alto"*.

En resumen, la Lógica clásica resulta restrictiva en cuanto a que:

- No ofrece los mecanismos adecuados para representar simbólicamente sentencias cuyo significado es impreciso.

- No ofrece un mecanismo de inferencia propicio para llevar a cabo el razonamiento aproximado.

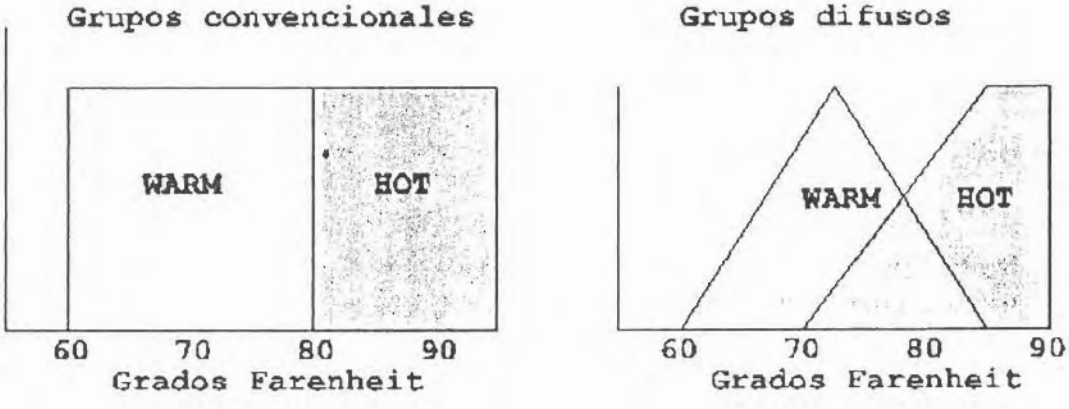
El objetivo de las secciónes que siguen, es exponer de manera clara los fundamentos de la lógica difusa ya que es la herramientas fundamental del prototipo del sistema de toma de decisión, a implementar.

5.3.1 ¿ Qué es la lógica difusa?

A mediados de 1960, el profesor Lofti Zadeh de la Universidad de California en Berkeley, reconoció que la naturaleza bivalente de la lógica booleana no contaba con los muchos grados de grises encontrados en el mundo real. Para contar con éstas infinitas graduaciones entre verdadero y falso, Zadeh expandió la idea de un conjunto clásico a lo que él llamó un conjunto difuso. A diferencia de la lógica booleana[i], la lógica difusa[ii] es multivaluada, ver [Gulley, 1997].

En vez de que un elemento sea 100% esto o aquello, o una proposición sea falsa o verdadera, la lógica difusa trabaja con grados de membresía, *cosas que pueden ser parcialmente falsas o parcialmente verdaderas al mismo tiempo.*

Por ejemplo, 80ª Farenheit ¿es tibio o caliente? En lógica difusa y en el mundo real "ambos" pueden ser la respuesta. Como se puede ver en la gráfica difusa, 80 grados es parcialmente tibio y parcialmente caliente en la representación del conjunto difuso[iii].



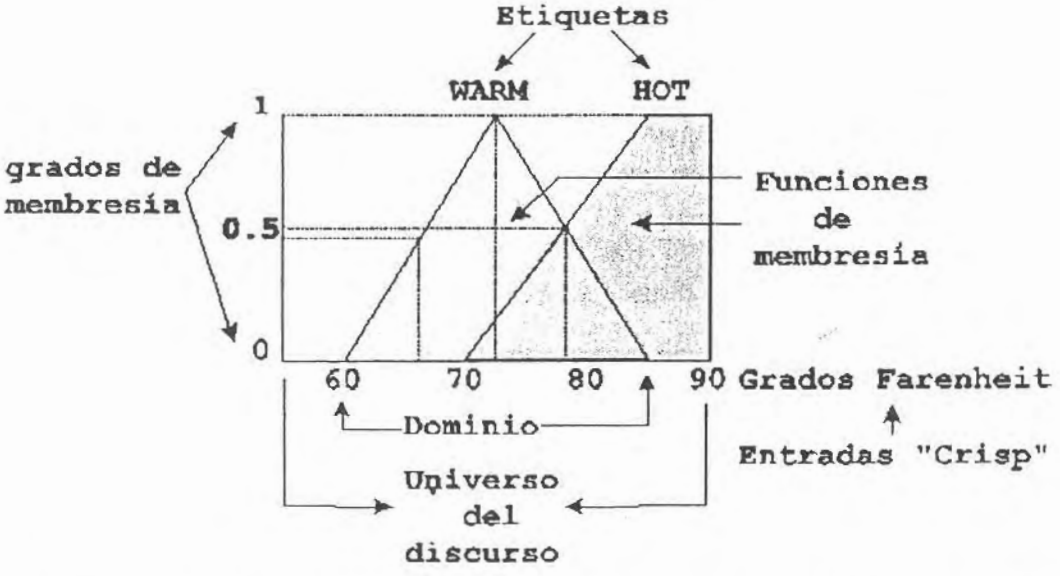
[i] Lógica Booleana (lógica clasica). Nombrada en honor a George Boole (1815-1864). Es un álgebra combinacional que trata variables a través de operadores como AND, OR, NOT, IF, THEN, etc.. Una variable en lógica Booleana toma el valor de verdadero o falso, pero no ambos.

[ii] Lógica Difusa (teoría del conjunto difuso). Es un grupo de principios matemáticos para el modelado de información basado en grados de membresía. Inicialmente propuesto por Lofti Zadeh en 1965, pero con raíces en la lógica multivaluada de Lukasiewicz.

[iii] Conjunto difuso (función de membresía). Es un conjunto con límites difusos. Un elemento en un conjunto difuso se manifiesta a sí mismo por un grado de membresía, que va desde 0 a 1.

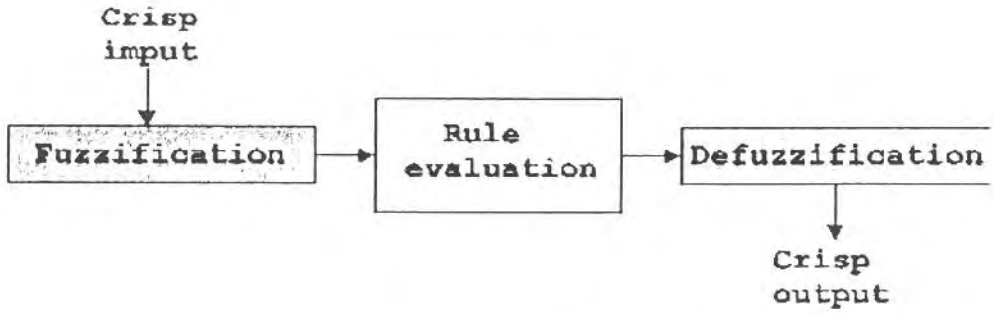
Como podemos ver en la figura anterior, la transición de conjunto a conjunto en la representación convencional (booleana) es instantánea. En la lógica difusa sin embargo, la transición puede ser gradual. En la lógica clásica usando los conjuntos convencionales, 79.9 grados debería ser clasificado como tibio y 80.1 como caliente. Pequeños cambios en el sistema causarían entonces significativamente diferentes reacciones. En un sistema difuso, un pequeño cambio en la temperatura causaría un cambio más aceptable en el funcionamiento del sistema. *La lógica difusa no solo reconoce cortes claros y contrastes blanco-negro, sino infinitas graduaciones entre ellas*. Esto puede parecer vago, pero la lógica difusa elimina cualquier incertidumbre asignando números específicos a éstas graduaciones, de acuerdo con [Cordón et al, 1997] y [Gulley, 1997].

A continuación mencionaremos algunos de los conceptos básicos asociados con la lógica difusa. Veamos la siguiente figura, ver [Siler, 1996]:



- **Grado de membresía:** Grado al cual un valor real (crisp) es compatible con una función de membresía.
- **Etiquetas:** Nombre descriptivo para identificar una función de membresía.
- **Función de membresía**: Define un conjunto difuso graficando entradas reales con su grado de membresía.
- **Entradas "crisp"**: Entradas distintas y claramente definidas.
- **Scope/Domain**: Ancho de una función de membresía.
- **Universo del discurso**: Rango de todos los posibles valores aplicables a una variable del sistema.

El utilizar la lógica difusa para la obtención de una solución *crisp* a un problema específico involucra tres pasos: la *fuzzificación*, *la evaluación de las reglas y la defuzzificación* (no se encuentra traducción), ver figura que sigue.



Los pasos anteriormente mencionados se describen a grosso modo a continuación, la referencia completa se encuentra en [Gulley, 1997], [Siler, 1996] y [Herrera et al, 1998]:

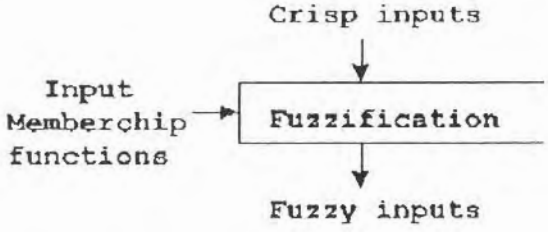
Fuzzification: En este paso, se toma un valor real (crisp variable) y se combina con funciones de membresía almacenadas para producir entradas difusas. Es natural que obtengamos múltiples entradas difusas que correspondan a una sola entrada real, dado que cada entrada real puede tener grados de membresía parciales en múltiples conjuntos difusos.

Evaluación de reglas: (Inferencia difusa) En este paso, la técnica llamada inferencia min-max es generalmente usada para calcular un valor numérico representando la veracidad para cierta acción consecuente basada en un conjunto de reglas trayendo el consequente. El resultado de la evaluación es una salida difusa para cada tipo de acción consecuente.

Defuzzification: Es el paso final en el procesado de lógica difusa en el cual el valor esperado de una variable de salida es derivado del aislamiento de un valor real del universo del discurso (rango de valores que puede tomar la entrada del sistema) de los conjuntos difusos de salida. Es decir, es el proceso de combinar todas las salidas difusas en un resultado real específico que puede ser aplicado para cada salida del sistema.

5.3.2 Fuzzificación y funciones de membresía.

El primer paso en el proceso de la lógica difusa involucra una transformación de dominio llamada fuzzificación (del inglés fuzzification[i]). Las entradas crisp son transformadas a "entradas difusas"[ii]. Por ejemplo, una entrada crisp de 78ºF se podría transformar a "tibio" en términos difusos. Para hacer esto, las funciones de membresía primero deben ser determinadas para cada entrada. Una vez que las funciones de membresía están asignadas, la fuzzificación toma un valor en tiempo real, como la temperatura, y lo compara con la información de la función de membresía almacenada para producir entradas difusas, como se ilustra en la figura que sigue.



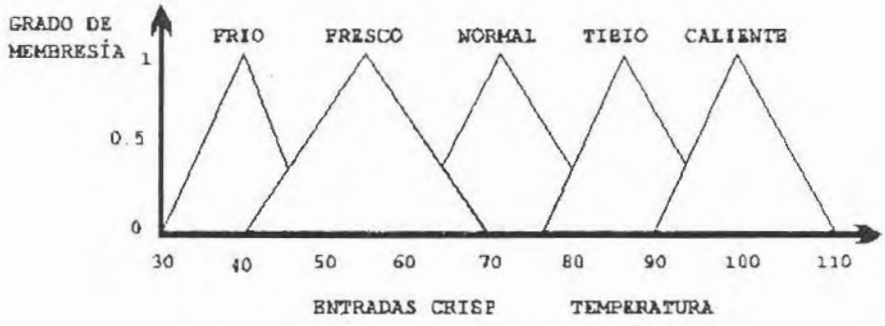
Para ilustrar el proceso de fuzzificación, veamos un sistema de riego casero, [Siler, 1996]. El controlador difuso en este sistema utilizará dos entradas (temperatura del ambiente y humedad del suelo) para calcular la duración de riego. El primer paso en la fuzzificación es asignar etiquetas en el universo del discurso a cada entrada crisp. Por ejemplo, para la temperatura podemos asignar etiquetas como *frío, fresco, normal, tibio y caliente*; y para la humedad, etiquetas como *seco, normal y húmedo*, ver figura que sigue. Después, se establecen las funciones de membresía para dar un significado numérico a cada etiqueta. *Cada función de membresía identifica el rango de valores de entrada que corresponda a la etiqueta. A diferencia con la lógica booleana, la función de membresía de una etiqueta no define límites donde la etiqueta cambia abruptamente, sino que hay una región donde los valores de entrada cambian gradualmente desde completamente aplicable a totalmente inaplicables*. Las salidas difusas también tienen funciones de membresía, que se comentan más ampliamente en la evaluación de reglas y la defuzzificación, según se describe en [Siler, 1996].

[i] Fuzzification : Es el primer paso en el proceso de lógica difusa. Se toma un valor crisp, como la lectura de temperatura y se combina con las funciones de membresía almacenadas para producir salidas difusas. Usualmente hay muchas entradas difusas correspondientes a un valor crisp dado que pueden tener diferentes grados de membresía en múltiples conjuntos difusos.

[ii] Entradas difusas : Transformación de entradas crisp en términos de etiquetas de función.

Una función de membresía[i] de entrada es creada especificando un número (grado de membresía[ii]) para cada posible valor de entrada para una etiqueta dada, por ejemplo, ver figura de abajo, [Siler, 1996]:

Etiqueta	Dominio
FRIO	30°-47°
FRESCO	40°-70°
NORMAL	60°-84°
TIBIO	75°-98°
CALIENTE	90°-110°



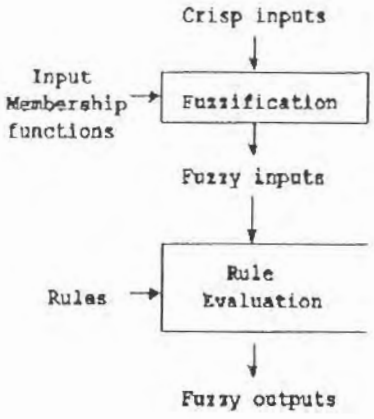
Describir las entradas crisp en términos difusos permite al sistema responder de mejor forma a cambios graduales en la temperatura.

5.3.3 Evaluación de reglas.

De acuerdo con [Siler, 1996] y [Cordón et al, 1997], el siguiente paso en el proceso de lógica difusa es llamado *evaluación de reglas*, donde el procesador difuso usa reglas lingüísticas para determinar qué acción de control debe ocurrir en respuesta a un conjunto dado de valores de entrada. La *evaluación de reglas* (llamada también inferencia difusa) aplica las reglas a las entradas difusas que fueron generadas en el proceso de fuzzificación, entonces evalúa cada regla con las entradas que fueron generadas del proceso de fuzzificación, ver figura que sigue.

[i] Función de membresía : Es una función matemática que da significado numérico a un conjunto difuso.

[ii] Grado de membresía : Es el grado al cual un valor de variable es compatible con un conjunto difuso.



Las reglas difusas son generalmente proposiciones implicativas que describen la acción a ser tomada en respuesta a varias entradas difusas, de acuerdo con [Siler, 1996] y [Cordón et al, 1997].

Por ejemplo:

> **Si** el suelo está húmedo **y** la temperatura caliente, **entonces** el tiempo de riego es corto.
> **Si** el carro se mueve rápido **y** el pavimento está seco, **entonces** aplicar completamente frenos.
> **Si** el agua del baño está demasiado caliente, **entonces** incrementar gradualmente el flujo de agua fría.

Aunque las reglas aparentan ser expresadas en un lenguaje libre y natural, están confinadas a un conjunto predeterminado de *términos lingüísticos*, y una sintaxis precisa, [Herrera et al, 1998] y [Cordón et al, 1997]; La sintaxis es:

> **SI** antecedente 1 **Y** antecedente 2 .... **ENTONCES** consecuente 1 **Y** consecuente 2 ...
>
> donde **Y** (**AND**) es uno de los operadores lógicos permisibles.

Es importante notar que las reglas siguen el sentido común del comportamiento del sistema y están escritas en términos de las etiquetas de la función de membresía lingüística.

Para un sistema de dos entradas y una salida, las reglas pueden ser representadas en una matriz, por ejemplo, en el sistema de riego, ver [Siler, 1996] y [Herrera et al, 1998]:

ANTECEDENT 1

	COLD	COOL	NORMAL	WARM	HOT
WET	SHORT	SHORT	SHORT	SHORT	SHORT
MOIST	SHORT	MED	MED	MED	MED
DRY	LONG	LONG	LONG	LONG	LONG

ANTECEDENT 2

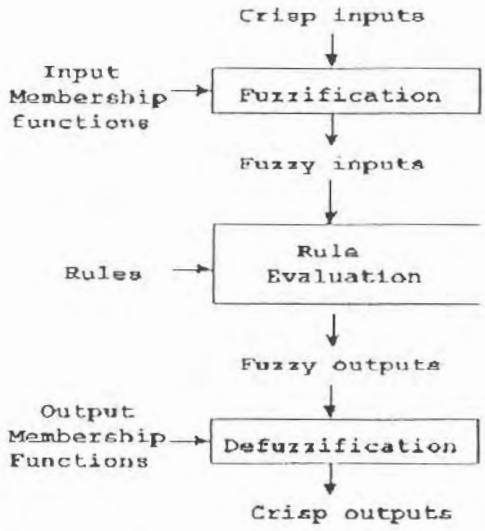
El Siguiente paso en el proceso de evaluación de reglas es evaluar la relevancia o grado de membresía de cada antecedente. Para encontrar dicha relevancia se extiende una línea vertical de referencia a través de la entradas crisp (eje x) y se encuentra el valor y donde intercepta las funciones de membresía.

Una vez que la relevancia ha sido determinada para cada antecedente, el siguiente paso es encontrar el grado de verdad (fuerza de la regla) para cada regla. Como los antecedentes están conectados por un operador Y (AND), la fuerza de la regla asume el valor más pequeño de los antecedentes. El valor mínimo es entonces el grado de verdad de esa regla, ver [Siler, 1996] y [Herrera et al, 1998]. El siguiente paso es determinar la salida difusa comparando las fuerzas de la regla de todas las reglas que especifican el mismo etiqueta consecuente. La salida difusa estará *determinada por la máxima fuerza de regla de todas las reglas que involucren la misma acción de salida.* Habrá una salida difusa por cada etiqueta de función de membresía de salida.

Además, si múltiples reglas se aplican para una función de salida, aquella que más se "aplique" será usada, es decir, si 2 o más reglas tratan de afectar la misma salida, la regla que es más verdadera dominará, ver detalles en ver [Siler, 1996] y [Herrera et al, 1998].

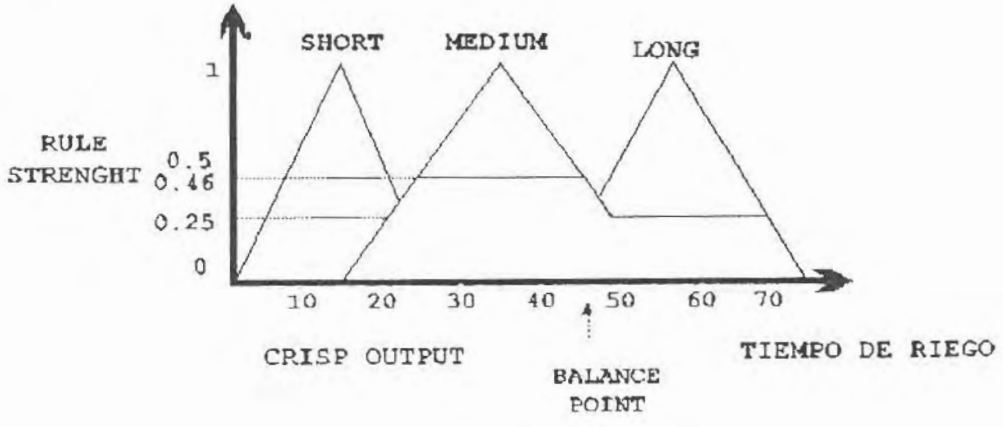
5.3.4 Defuzzificación.

En la defuzzificación, todas las salidas difusas significativas serán combinadas en un resultado específico y comprensible para la variable de salida, ver figura de abajo. En este proceso, todos los valores difusos de salida efectivamente modifican su respectivas funciones de membresía de salida. *Al igual que en la evaluación de reglas, tomando el valor más grande de fuerza de la regla para cada consecuente, las reglas que son más verdaderas dominan*, de acuerdo con [Gulley, 1997] y [Siler, 1996].



Una de las técnicas más utilizadas para la defuzzificación es llamada el método del centro de gravedad (COG) o del centroide, puede ver tablas, que se presentan en el apéndice C de funciones soportadas por el Fuzzy Logic ToolBox.

En este método, cada función de membresía de salida sobre el valor indicado por su salida difusa respectiva es truncada. Entonces las *funciones de membresía resultantes son combinadas y se calcula el centro de gravedad total.* Tal corte se denomina corte lambda. El siguiente paso es encontrar el punto de balance del centro de gravedad del área compartida Esto representa la salida defuzzificada. Por ejemplo, si las salidas difusas para nuestro sistema de riego son: SHORT : 0, MEDIUM: 0.46, LONG: 0.25, entonces:



El Método de defuzzificación de COG puede también ser aplicado a funciones de membresía de salida únicas (de dominio tendiendo a 0). Una función de éste tipo es representada por un punto individual en el espacio y de masa cero. El truncado tiene por resultado la reducción en su altura, un estudio mas afondo de estos conceptos, puede hallarlo en [Gulley, 1997], [Herrera et al, 1998] y [Siler, 1996].

5.3.5 Como observar comportamiento del sistema.

Este paso consiste en la prueba del modelo, al aplicarle varias combinaciones de entradas y observar el conjunto de salidas. Las características del modelo que se necesitan verificar son:

- Valores de salida. Los valores defuzzificados regresados por el modelo deben ser checados.
- Superficie de control. El mapeo de entradas a salidas se transforma en una superficie de control. En un sistema de dos entradas y una salida, la superficie de control puede ser visualizada en tres dimensiones.
- Simulación del tiempo. El comportamiento del modelo fuera del tiempo debe ser observado para la sensibilidad y la estabilidad.

El aprovechamiento de la verificación del modelo dependerá de la herramienta usada. Las herramientas de alto nivel proveen ambientes ricos, interactivos y gráficos para observar el comportamiento del sistema, aislando problemas potenciales y modificando parámetros del sistema. Las herramientas como el Fuzzy Logic ToolBox de MathWorks, provee una interface amigable para modificar reglas y funciones de membresía y desplegar mapeos de entrada/salida en dos dimensiones.

Lo primero que se debe hacer es disponer de entradas a valores que producen valore de salida conocidos. Para cada combinación de entradas, verifique que:

- Las salidas tengan sentido. Si no es así, identifique las reglas y funciones de membresía que están causando el error.
- La(s) regla(s) identifican correctamente el comportamiento deseado.
- Las etiquetas para los valores de la entrada asociados tiene la forma correcta y están bien traslapados.

Después de examinar las salidas, se debe intentar romper al sistema, probando exhaustivamente varias combinaciones de entrada. También debemos poner particular atención a las siguientes regiones:

- Valores de entrada en los finales extremos de los dominios de cada función de membresía.
- Valores de entrada correspondientes al traslape de funciones de membresía.
- Valores de entrada con algunos grados de verdad con tantos antecedentes como sea posible.

Después de un gran número de simulaciones, se debe verificar que la actividad de las variables de entrada y de salida es robusta y bien distribuída alrededor de sus respectivos universos del discurso. Además, las reglas crean una superficie de control de salida centrada acerca del 80% de espacio de salida consecuente. Asimismo se debe verificar que la superficie de control conforme las especulaciones del diseñador. La falta de conformidad es usualmente debida a uno de tres problemas en el siguiente orden.

- Reglas especificadas incorrectamente.
- Funciones de membresía incorrectamente definidas.
- Operadores difusos inapropiados.

Finalmente, asegúrese de que el rango de valores de salida a través del rango de valores de entrada está correctamente definido. La incompatibilidad entre conjuntos de entradas y salidas difusas puede causar problemas.

5.3.6 La Herramienta de Desarrollo Difuso - Fuzzy Logic ToolBox.

La Herramienta que se utilizo para implementar nuestro sistema de toma de decisiones es el *MathWorks Fuzzy Logic Toolbox,* que es una colección de funciones construidas sobre el ambiente de computo numérico de *MathWorks* MATLAB ver 5.0, cuyo propósito es brindar un ambiente de trabajo adecuado para el diseño, simulación e implementación de Sistemas de Lógica Difusa, para usar dentro del ambiente de desarrollo de MATLAB, o integrarlo a un modulo de simulación diseñado en SIMULINK, e incluso podemos construir aplicaciones en lenguaje C/C++, que puede operar en forma autónoma sobre el sistema de lógica difusa (*en el caso de el sistema de toma de decisiones aquí propuesto en particular a si se realizo*), construido bajo esta herramienta, dentro de otras características ofrece una interfaz gráfica de usuario, como entorno de desarrollo, lo que facilita los procesos que intervienen en la realización de nuestro sistema. A continuación se mencionan las características más significativas de diseño que soporta el Fuzzy Logic ToolBox.

Fuzzy Logic ToolBox permite que el usuario seleccione distintos parámetros del Sistema de Lógica Difusa en los siguientes aspectos:

Tipo de Conjuntos Difusos
Tipo de Difusores
Tipo de Concresores
Operador AND
Operador de Composición
Operador de Unión e Intersección
Operadores de Implicación

Las Tablas 1 a la 6 del apéndice C, se resumen estas opciones, clasificándolas en Tipo de Conjuntos Difusos, Tipo de Difusores, Tipo de Concresores, Opciones Matemáticas, Normas e Implicaciones, una descripción mas detallada la puede encontrar en [Gulley, 1997].

5.3.7 Redes neuronales y lógica difusa.

Las redes neuronales y la lógica difusa son tecnologías complementarias. Los ingenieros que toman esto en cuenta pueden resolver problemas complejos en un menor tiempo y desarrollar los productos más robustos con mayor potencial de crecimiento, como se menciona en [Gulley, 1997]. Las redes neuronales están altamente conectadas con sistemas no-lineales que pueden "aprender" una respuesta característica basada en datos de entrenamiento. Esta adaptabilidad es útil cuando se desarrollan sistemas complejos o en sistemas que deben cambiar su comportamiento con condiciones cambiantes de entrada. Las redes neuronales pueden también generalizar resultados de datos de entrada. Esta habilidad es importante cuando se trabaja en condiciones incompletas o ruidosas, de acuerdo con [Gulley, 1997] y [Russell et al, 1996].

Las redes neuronales y las unidades de lógica difusa pueden ser combinadas para formas sistemas híbridos ,que capturan lo mejor de ambas tecnologías. La capacidad de aprendizaje de las redes neuronales podría ser usada en el diseño cíclico de sistemas de lógica difusa para generar automáticamente las reglas y ajustar las funciones de membresía, para un sistema difuso, cabe mencionar que la herramienta aquí utilizada para implementar nuestro sistema de toma de decisiones (*MathWorks Fuzzy Logic Toolbox)*, dentro de su mecanismo de inferencia, esta integrado el sistema de inferencia de redes neuronales adaptativas, con funciones "backpropagation" - *Adaptive-Network-Based Fuzzy Inference Systems* - (ANFIS), no se efectuara ningún estudio de evaluación de esta técnica de inferencia basada en redes neuronales, sin embargo la bibliografía para un posterior estudio esta disponible en [Jang, 1993] y [Jang et al, 1995].

5.4 Prototipo de sistema de toma de decisiones distribuido orientado al desarrollo del estado de Oaxaca basado en la arquitectura de objetos componente.

Los esfuerzos en la implementación y construcción de este prototipo de aplicación están dirigidos a reafirmar y comprobar que la arquitectura, en este trabajo propuesta, se adaptara y funcionara correctamente; de tal forma que se integren las tecnologías de objetos componente distribuidos (componentes distribuidos), Internet, procesamiento de datos y "conocimiento" de manera distribuida; todo ello para analizar la situación actual con criterios de estimar la influencia de diferentes factores en aspectos económicos y educativos del estado de Oaxaca.

Sabemos, según [INEGI, 1997] y [INEGI, 1994], uno de los tantos problemas, a los que se enfrenta una gran cantidad de sectores de la investigación, analistas, planeadores, inversionistas, gobierno estatal, municipal y federal, etc. es la obtención de información verídica, concentrada, analizada y sistematizada que se encuentra distribuida en el estado de Oaxaca, esto trae como consecuencia perdida de tiempo, errores que se cometen al transcribir la información, alta inversión en recursos humanos y materiales al consultar las fuentes de *información escritas* que existen en diversos centros de información. En la mayoría de los casos, los usuarios que requieren de esta información se encuentran repartidos trabajando en sitios diferentes. Considerando que estos sitios están conectados por medio de una red de comunicaciones, ya sea local(LAN) o a gran distancia (Internet). Como parte del alcance, se considera a Internet como vía, ya que representa la red WAN más importante que existe, debido a que la gran mayoría de las computadoras en el mundo están conectadas a ella y además hoy muestra un crecimiento explosivo.

5.4.1 Descripción general del prototipo de implementación a diseñar y construir.

El prototipo a diseñar e implementar, trata de la creación de un sistema distribuido de toma de decisiones, para recuperación y presentación de información orientada al desarrollo del estado de Oaxaca en el nivel educativo y económico, basado en la arquitectura de objetos componente, es decir los módulos que integran al modelo de decisión serán componentes que van a proveer de las interfaces necesarias para interactuar con otros, cabe mencionar que estos componentes adoptan toda la funcionalidad en la arquitectura descrita (ver capitulo 4), como punto importante se debe aclarar que cada componente provee de un acceso concurrente, por medio de múltiples hilos de ejecución; por otro lado el problema a resolver es simple y consiste en que si consideramos que existen ciertos niveles en los índices de la economía, educación, y población en cada región, municipio, distrito o municipio, y necesitamos decidir alguna situación, a quien, o donde *Invertir, apoyar*, etc; necesitamos contar con un mecanismo de consulta que determine en términos de variables lingüísticas que regiones del estado son "las más pobres" o que municipios tienen un nivel educativo "muy bueno", "bastante malo" o "excelente"; sabemos que por lo regular este tipo de expresiones, tienen implícita cierta imprecisión o vaguedad, si en la toma de decisiones consideramos *factores sociales y humanísticos* en función de determinar en que situación económica o educativa, se encuentra cierto lugar del estado de Oaxaca, pues se cree que existe mucha subjetividad entre decir si una región es "pobre" a decir que es "muy pobre", para ello se eligió a la lógica difusa para modelar este problema, de tal manera nos permita manejar con precisión ciertos rangos de incertidumbre en la toma de decisiones; a la vez se integrara con aplicaciones de objetos componente, que estarán distribuidas, y cada una va ha contar con las responsabilidades que se presentan abajo:

Si consideramos que el nivel de educación esta dado por el número de *personas alfabetas* que saben leer y escribir, y que los niveles económicos de cada entidad municipio esta en función del numero de la población económicamente activa, entonces podemos suponer que existe una base de datos en algún sitio A, que se encarga de *contener datos concentrados* sobre el estado de Oaxaca, por medio de algún mecanismo de bases de datos; la tabla de base de datos contiene los siguientes campos:

REGION: Representa, valores numéricos enteros, correspondientes a las regiones ocho regiones del estado de Oaxaca.

DISTRITO: Representa, valores numéricos enteros, correspondientes a los 30 distritos, presentes en el estado de Oaxaca.

MUNICIPIO: Representa, valores numéricos enteros, correspondientes a los 570 municipios presentes en el estado de Oaxaca.

NOMBRE: Representa, valores tipo string o cadena, correspondientes a los nombres de los municipios, presentes en el estado de Oaxaca.

ECONOMICO: Representa, valores numéricos reales, correspondientes a los ingresos económicos por municipios en el año de 1995, presentes en el estado de Oaxaca.

PACTIVA: Representa, valores numéricos reales, correspondientes a la población económicamente activa por municipios en el año de 1995, presentes en el estado de Oaxaca.

NESCOLAR: Representa, valores numéricos reales, correspondientes a la población alfabeta por municipios en el año de 1995, presentes en el estado de Oaxaca.

NPOBLACION: Representa, valores numéricos reales, correspondientes a la población registrada, según censo de 1995 por municipios en el año antes citado, presentes en el estado de Oaxaca.

En el sitio A, aparte de la colección de datos, requerimos de una unidad que encapsule y publique en forma de un objeto componente (componente), los servicios que el sitio A ofrece, pues aparte de la simple recolección o contención de datos, es necesario tener un servicio que extraiga de la base de datos, una colección de registros, basados en una búsqueda por medio de variables lingüísticas, como parte del proceso de defuzzificación puestos registros contiene la información necesaria que se presentara al usuario final y que con base en esta información tome las decisiones adecuadas.

Por otro lado, suponemos que en otro lugar, digamos el sitio B, existen *analistas de datos,* quienes van jugar el papel de extractores del conocimiento (expertos), quienes podrán generar el sistema de toma de decisiones, que van a integrar la base de conocimientos del sistema, a una herramienta de análisis y diseño automatizada de sistema de toma de decisiones, como el Fuzzy logic ToolBox.

Podemos identificar otro sitio C, donde se va ha establecer un componente controlador(ver capitulo 4), este elemento es el responsable de manipular, los servicios provistos por el sistema total, de tal forma que servirá como un bus entre el componente del cliente y los servidores disponibles en el sistema (ORB), es decir que proveen las interfaces necesarias para consulta y recuperación de datos en forma asíncrona, respecto al componente que hace las consultas, como al que provee el servicio requerido.

5.4.2 Diseño del Sistema de Toma de decisiones en MathWorks Fuzzy Logic ToolBox.

En esta sección vamos a describir paso a paso el diseño de nuestro sistema de toma de decisiones basado en Lógica Difusa, en primera vez, deberemos ejecutar la aplicación MATLAB y enseguida escribir en la línea de comandos el comando fuzzy, ver figura que sigue.

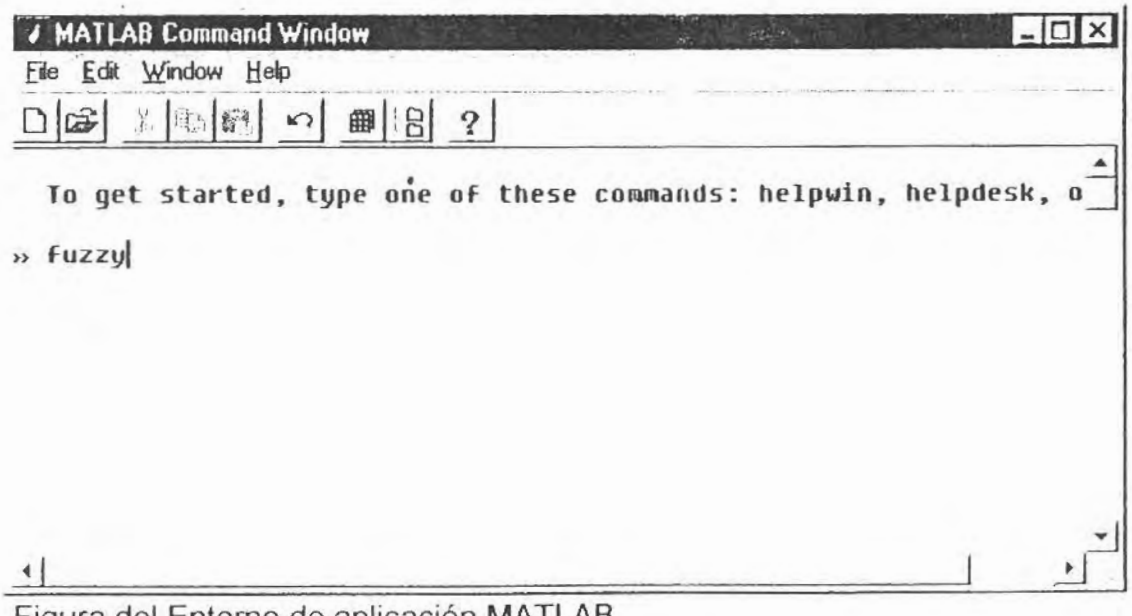


Figura del Entorno de aplicación MATLAB.

En seguida aparecerá una ventana como la mostrada en la figura 5.1; esta ventana representa los *Universos de Entrada y Salida, así como la de la Base de Reglas*, como podemos ver, se muestra que éstos no han sido diseñados aún; en este procedimiento se ha creado un nuevo espacio de trabajo, por medio del editor conocido como FIS(editor de sistemas de inferencia difusos), aun sin un nombre que lo identifique - Untitled.

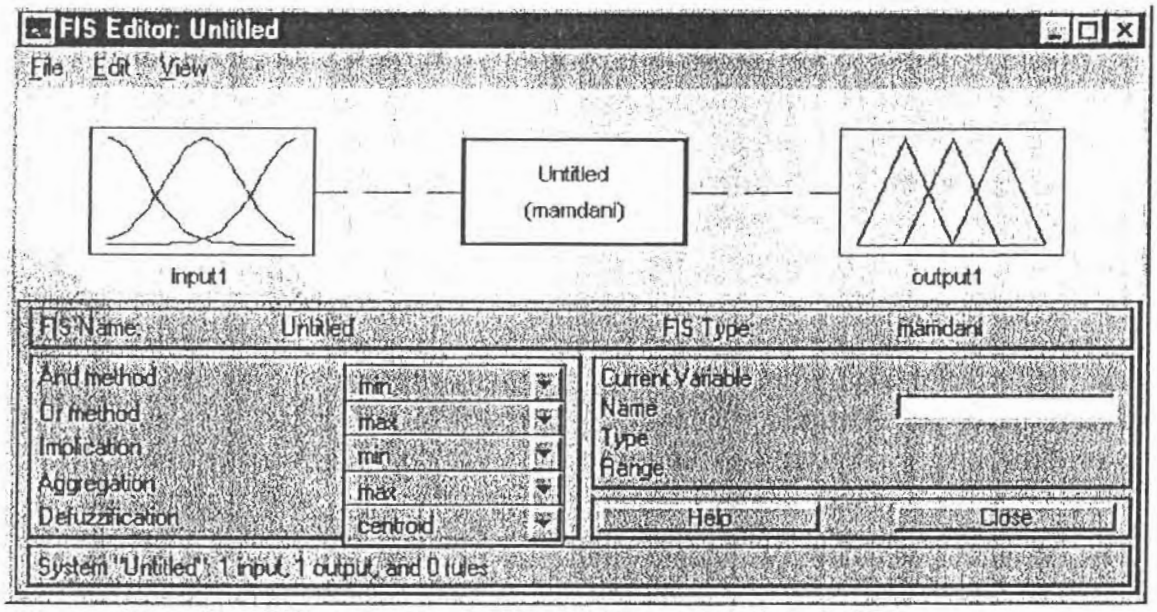


Figura del editor FIS, para un nuevo sistema de lógica difusa.

Ahora efectuaremos un proceso de cinco pasos, para diseñar y verificar nuestro sistema:

1. Diseñar el Universo de Entrada
2. Diseñar el Universo de Salida
3. Diseñar la Base de Reglas
4. Prueba del sistema, mediante simulación en el fuzzy logic toolbox

El orden en el que se deben efectuar estos pasos sólo está regido por las siguientes reglas:

- *El diseño de la base de reglas debe ser posterior al diseño de los Universos de Entrada y de Salida.*
- *Algunos cambios en los Universos de Entrada o de Salida (como el número de Valores Lingüísticos de alguna variable) implican una redefinición de la Base de Reglas.*

A continuación se presentan los cinco pasos arriba mencionados.

Paso 1: Diseño del Universo de Entrada

El diseño del Universo de Entrada consiste en la definición de las Variables Lingüísticas que lo forman, y de los difusores que acompañan a éstas. Fuzzy logic toolbox permite efectuar estas definiciones.

A partir del nuevo espacio de trabajo, abierto, podemos trabajar sobre las opciones y contenidos que presenta la forma de la figura 5.1, haciendo uno o doble 'click' en la figura que representa al universo de entrada(s), universo de salida(s), así como la de la Base de reglas. En el primer caso de un 'click', se harán activos los contenidos que requiera, el componente activado, con la posibilidad de asignar y/o modificar los valores que le correspondan.

El menú *Edit-Add Input*, permite definir *variables* de entrada, después de seleccionar esta opción, podemos asignar los correspondientes valores, que deba tener dicha variable, por ejemplo a la variable disponible por default es "input1", podemos renombrarla y asignarle la etiqueta "Economico", además al hacer un doble 'click', sobre la variable elegida, podemos ejecutar el editor de funciones miembro, correspondiente a la variable elegida, se pueden asignar los difusores según convenga, en nuestro caso se opto por una función triangular; enseguida se presentara la figura que aparece:

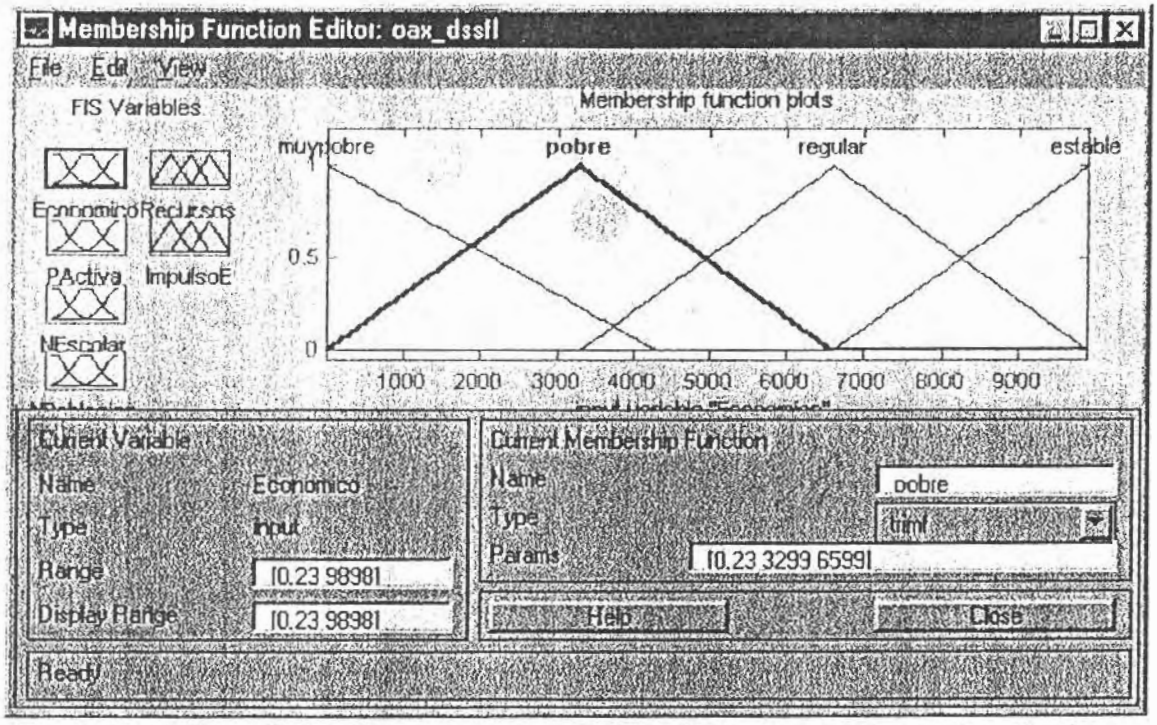


Figura de Editor de las funciones de membresia de los universo de entrada y salida

Consecutivamente se pueden ir añadiendo, variables y definiendo, sus propiedades, para nuestro caso estas las variables del universo de entrada, son: Economico, Pactiva, Nescolar y NPoblacion al final nuestro sistema de entradas, se ve como la figura de arriba.

Cabe mencionar que para la variable Npoblacion, se mezclaron varios difusores, como se menciono en una de las secciones de conceptos de lógica fuzzy, debido a los diferentes grados de membresía de la variable Npoblacion, ver figura de abajo.

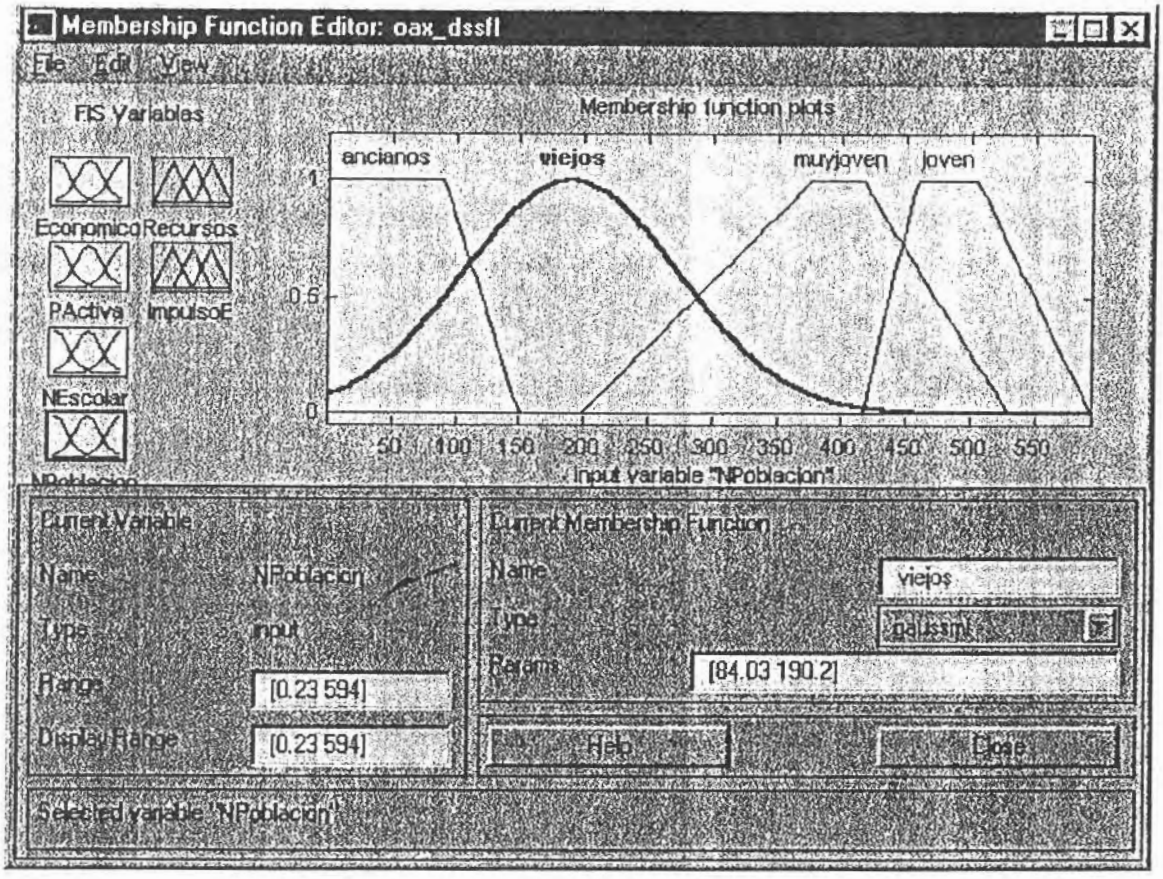


Figura de Editor de las funciones de membresía de universo de entrada, usando varios tipos de difusores.

Paso 2: Diseño del Universo de Salida:

El diseño del Universo de Salida consiste en la definición de las Variables Lingüísticas que lo forman, y de los concresores que acompañan a éstas. Fuzzy logic toolbox, permite efectuar estas definiciones, que son del todo análogos a empleados para definir el Universo de Entrada, se deben usar concresores en lugar de difusores, de hecho el proceso del Paso 2, para la herramienta aquí utilizada, se realiza en forma simultanea, como se ha mostrado en las figuras de arriba.

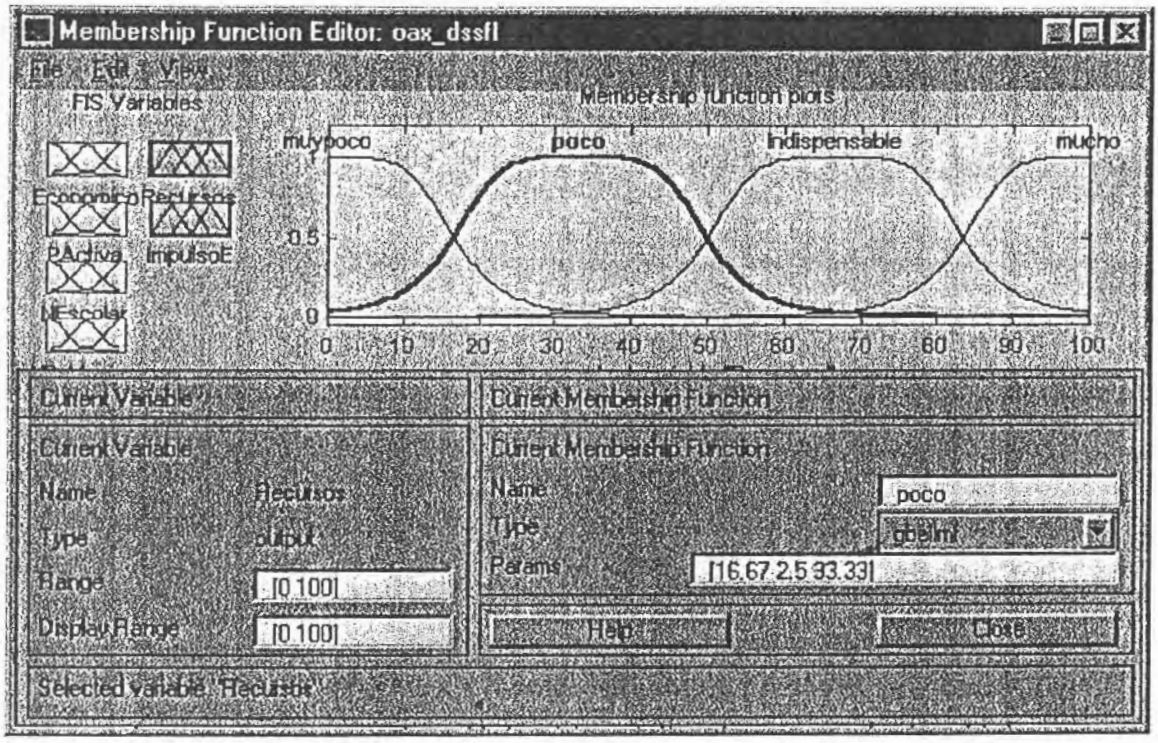


Figura del universo de salida para la variable Recursos

Consecutivamente se pueden ir añadiendo, variables al universo de salida e ir definiendo sus propiedades, las variables del universo de salida para nbuestro sistema son Recurso e ImpulsoE.

Paso 3: Diseño de la Base de Reglas

El diseño de la Base de Reglas consiste en la definición de un conjunto de reglas, cada una de las cuales está conformada por un Antecedente y un Consecuente; éstos, a su vez, están definidos por términos de la fórma *Variable Lingüística i es Valor Lingüístico j*; en donde *Variable Lingüística i* es una de las variables que forman parte del Universo de Entrada (si se trata de un Antecedente) o de Salida (si se trata de un consecuente), y *Valor Lingüístico j* es uno de los Valores que puede tomar *Variable Lingüística i*; adicionalmente, cada término del Antecedente puede estar acompañado de un Modificador Lingüístico (que por defecto es 1.0), cuyo significado será el de adicionar el calificativo *muy* (si es mayor que 1.0) o *poco* (si es menor que 1.0). Enseguida, se muestra la figura, de nuestra definición de las reglas para el sistema de toma de decisiones, (solo algunas reglas), ver figura de abajo.

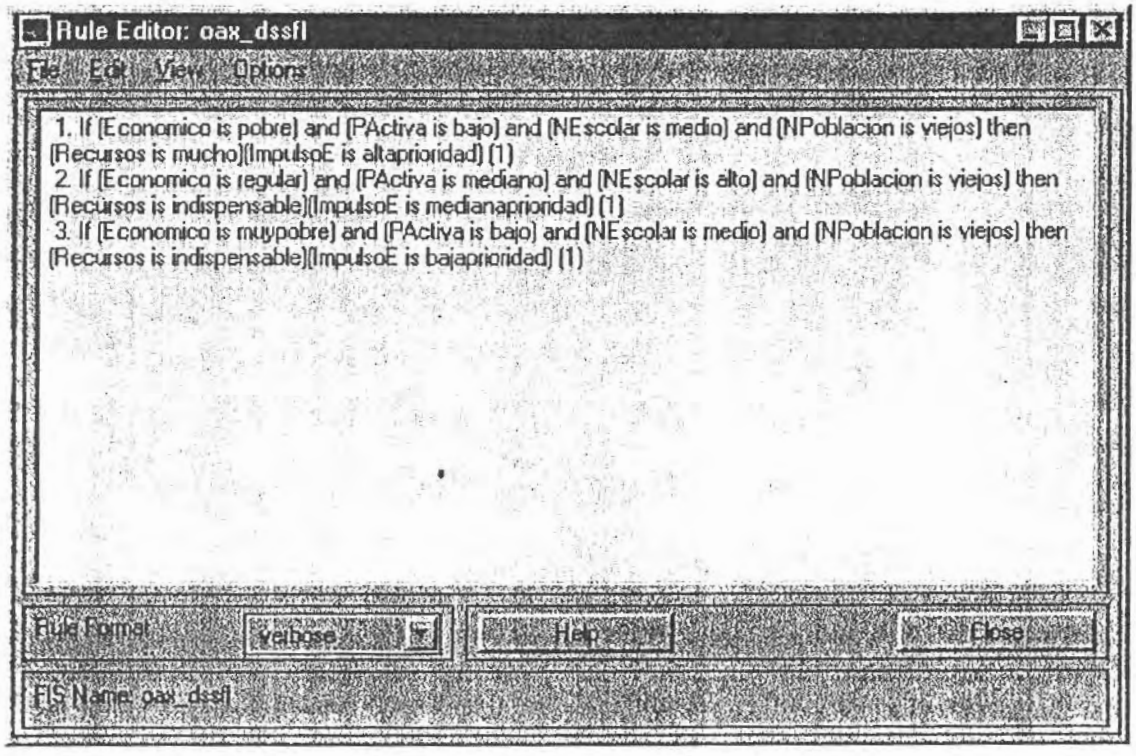


Figura del editor de las reglas del sistema de inferencia (solo unas cuantas)

Paso 4: Prueba del sistema, mediante simulación en el fuzzy logic toolbox.

Para, verificar el funcionamiento de nuestro sistema, es necesario realizar lo siguiente, seleccioné en el editor FIS, la opción del menú View_view-rules (ctrl_4) En esta figura se muestra las variables que conforman el universo de entrada y las variables que forman el universo de salida, por medio del mouse, sobre las gráficas sensitivas que presenta el visualizador de las reglas, podemos variar los valores de las variables de entrada sobre sus respectivas función o funciones de membresía, de tal forma que forma automática, podemos ver se asignan valores a las variables del universo de salida. En la figura que sigue, podemos apreciar un ejemplo de este mecanismo.

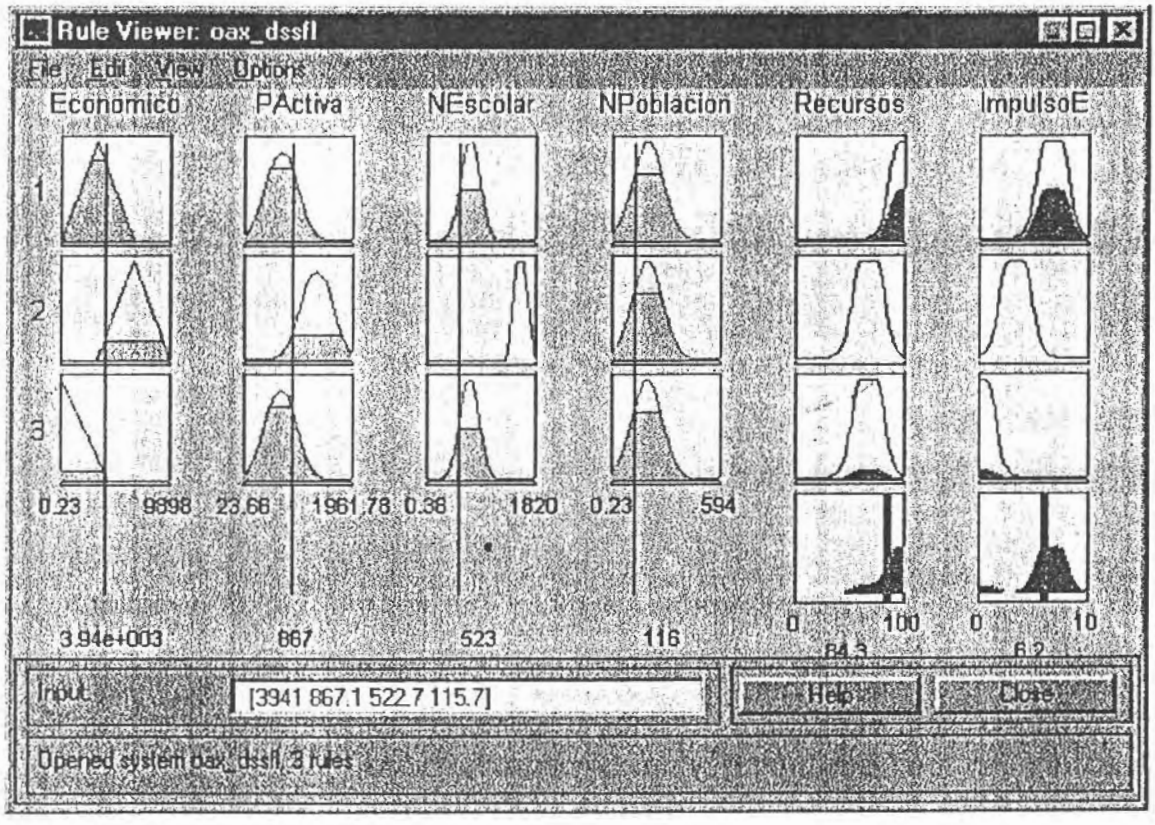


Figura del visualizador de reglas, para verificación del sistema.

5.4.3 Interacción del sistema.

El componente interfaz de usuario establece una conexión asíncrona con el componente Controlador, enviado la consulta que necesite ser procesada, este fue diseñado e implementado para accesar a el sistema desde una aplicación ejecutable en red, o desde una interfaz http-DCOM, en Internet bajo computadoras PC en la plataforma Windows 95, la forma que se le presenta al usuario en la interacción con el sistema de recuperación de información es una vista interactiva, basada en componentes ActiveX de interfaz visual: *Agente de interfaz de usuario, Mapa sensitivo del estado de Oaxaca, Formas y menúes desplegables*. De tal forma que su interacción con el sistema sea lo más agradable y eficaz posible.

Por otro lado tenemos que el componente controlador, es capaz de analizar peticiones, descomponer la consulta en varias subconsultas, de acuerdo con la petición hecha, es decir el controlador localiza, ejecuta, y coordinar estas consulta/respuestas con los demás componentes que forman el sistema distribuido y regresar el resultado de esa petición hecha.

El componente controlador, es capaz de recibir múltiples peticiones externas y este puede asignar *múltiples hilos* de ejecución para cada tarea que se le delegué, de tal manera que el procesamiento se efectúa de manera semi-paralela, en múltiples computadoras que componen al sistema distribuido.

Consultas posibles

El usuario podrá efectuar algunas de las siguientes consulta lingüísticas al sistema distribuido de toma de decisiones, como preguntar por los municipios o distritos que:

Su nivel económico y educativo sea "muy malo".
Nivel económico sea "bueno".
Nivel económico sea "muy bueno" y su nivel educativo sea "regular".
Y otras consultas parecidas, en términos de variables lingüísticas

Pruebas

Las pruebas que se realizaron, fueron diseñadas para ejecutarse, bajo la tecnológica de componentes ActiveX y el ORB de Microsoft DCOM. Se realizaron, unas pruebas, instalando, el componente de la maquina de inferencia ANFIS, en el servidor de Windows NT, para un mejor rendimiento. El componente de contenedor de datos de MS Access 7.0, en una PC, corriendo, bajo Windows 98. El componente o componentes de usuario, bajo dos PC, corriendo Windows 95, con Internet Explorer 4.1.

5.5 Conclusiones y Trabajo futuro.

En el presente trabajo se definieron un conjunto de propiedades para los sistemas distribuidos, basados en objetos componente, junto con una arquitectura general que permite su definición, especificación e implementación. Podemos afirmar que definitivamente el disponer de una arquitectura no sólo ofrece una plataforma para definir las propiedades mencionadas, sino que permite razonar sobre ellas e incorporar nuevas propiedades fácilmente, también podemos decir que una arquitectura basada en objetos componente de cómputo distribuido puede facilitar la definición de cómo se "comportan" los sistemas, los subsistemas, y las herramientas o las aplicaciones en un entorno de cómputo distribuido.

El enfoque de los objetos distribuidos en este trabajo esta dado en términos del sistema de capas o niveles de componentes, que al exponer *interfaces* de uso se están ofreciendo *diálogos sobre temas particulares*, por lo que en un trabajo futuro pudieran emplearse heurísticas de diseño orientado a objetos para la programación de protocolos de coordinación, y correspondientemente, preguntar si un dialogo esta soportado es buscar una interfaz de uso en un objeto, por lo que los mecanismos de objetos distribuidos pueden emplearse en modelos de comunicación y coordinación más elaborados. Cabe mencionar, que los beneficios de un sistema basado en componentes pueden resumirse así:

- Reutilización y posibilidad de compartir objetos para que el tiempo de recodificación sea mínimo
- Interface abstracta para mayor flexibilidad.
- Mantenimiento de sistema más fácil.
- Los componentes concurrentes, basado en el multihilado - multithreading de un programa pueden ejecutarse de forma paralela en un multiprocesador.

También se ofrece una alta productividad tanto al diseñador, como al desarrollador del sistema, a través de una Infraestructura de informática distribuida que permite *transparencia, seguridad, capacidad de dimensión, fiabilidad y disponibilidad*, pues los desarrolladores de aplicaciones pueden ser capaces de tratar problemas distribuidos sin necesidad de aprender enormes API's o librerías de programación ni los mecanismos de comunicación existentes y su complejidad inherente.

Por otro lado, también cabe considerar que uno de los problemas más difíciles que han enfrentado las ciencias exactas - la matemática, la lógica, etc. - es el calculo de lo que no es exacto, esto es, como manejarse sin perder su carácter duro aceptando las variables naturales de la realidad.

APENDICE A

ActiveX EN SISTEMAS DISTRIBUIDOS E INTERNET.

ActiveX, es un conjunto de tecnologías desarrolladas por Microsoft para unificar las diferentes tecnologías de información dentro de Internet, como lo es el *cómputo distribuido, bases de datos distribuidas, y reuso de componentes software.*

Para el entorno distribuido Microsoft constituye un conjunto de estándares de interfaz y diseño conocido como *Componentes ActiveX*. Los estándares ActiveX son fundamentalmente extensiones de la tecnología conocida anteriormente como OLE(Object Linking and Embedding), en cuanto a la comunicación a través de una red, los componentes ActiveX pueden responder a solicitudes de clientes mediante un mecanismos remoto: COM distribuido, para plataformas como Digital UNIX, Open MVS, Microsoft Windows 95 y Windows NT.

Un componente ActiveX es cualquier código binario que hace sus objetos visibles para otras aplicaciones de acuerdo con los estándares de ActiveX, basados en el Modelo de objetos componentes (COM), ver [Pinnock, 1998] y [Wang et al, 1997].

Como la finalidad de ActiveX es compartir objetos, los componentes ActiveX implican una relación *cliente-servidor* en la que el cliente solicita objetos y el servidor los proporciona. Sin embargo, la distinción entre clientes y servidores es confusa porque el mismo componente puede ser (y normalmente lo es) tanto cliente como servidor. Por esta razón, se denominan "componentes ActiveX" en lugar de "clientes ActiveX" y "servidores ActiveX".

Ambos componentes de una relación cliente-servidor pueden estar contenidos en un único equipo o pueden ejecutarse en distintos equipos conectados por una red.

Un componente ActiveX es *local* cuando se ejecuta en el mismo equipo que el proceso cliente y es *remoto* cuando se ejecuta en un equipo diferente.

Así que podemos identificar los siguientes tipos de componentes ActiveX, que dan soporte al desarrollo de software basado en componentes y en forma especial al computo distribuido:

Local en proceso (in-process).

En este caso, el código de origen para el módulo de clase se compila en una biblioteca (archivo .DLL) binaria independiente y se carga en un proceso de aplicación host como componente en proceso. Esto proporciona un rendimiento de acceso a las interfaces del componente que casi no se puede distinguir del caso de despliegue interno de integración total descrito anteriormente, pero que también proporciona un modelo de empaquetamiento que permite la actualización del componente sin tener que *recompilar* la aplicación host (cliente). Puesto que el componente en proceso es un archivo binario independiente, otras aplicaciones cliente lo pueden reutilizar de forma más efectiva que si se hubiera compilado junto con la aplicación host original [Pinnock, 1998] y [Wang et al, 1997].

Local fuera de proceso (out of process).

En este caso, el código de origen para el módulo clase se compila en un componente binario (archivo EXE) fuera de proceso e independiente, al que llama una aplicación cliente ubicada en otro espacio de proceso. Esto proporciona un rendimiento más lento de acceso a las interfaces del componente que en los dos casos anteriores, pero permite ejecutar el componente por separado, en su propio espacio de proceso protegido y con su propio subproceso de ejecución independiente. Esta independencia puede resultar muy útil si múltiples aplicaciones compartirán al mismo tiempo el componente o si éste va a efectuar tareas relativamente largas que una aplicación cliente que llame puede no desear bloquear mientras espera a que se completen. (Las interfaces asíncronas de componentes se tratan en la ultima parte de este apéndice).

Remoto fuera de proceso.

En este caso, el código de origen para el módulo clase se compila en un componente binario (archivo EXE) fuera de proceso e independiente, al que llama una aplicación cliente ubicada en otro espacio de proceso o en otro equipo. Esto proporciona un rendimiento más lento de acceso a las interfaces del componente que en los tres casos anteriores, pero permite que el componente aproveche el

ancho de banda y los recursos de un equipo servidor externo compartido, potencialmente más poderoso. La ubicación de un componente en un sistema centralizado de esta manera también puede simplificar en gran medida la actualización y administración de las tareas, gracias al modelo de despliegue limitado. Este modelo también permite que múltiples procesos procedentes de equipos cliente compartan al mismo tiempo un componente [Pinnock, 1998] y [Wang et al, 1997].

Remoto en proceso.

En este caso, el código de origen se compila en una biblioteca binaria independiente (archivo DLL) y se carga en un proceso host de un *equipo remoto* como componente en proceso. Aunque el cliente remoto debe recibir una referencia al componente en proceso indirectamente desde el proceso host, todas las referencias al componente procedentes del cliente después de ese momento son directas y el rendimiento del acceso es, por tanto, comparable al del despliegue remoto fuera de proceso. *Este modelo se basa en el despliegue remoto fuera de proceso, pero tiene la ventaja adicional de no requerir la carga de todo un proceso para ejecutar el componente.* Esto puede reducir enormemente el tiempo de conexión y los recursos del sistema operativo necesarios para ejecutar múltiples componentes, al tiempo que permite que los programadores desplieguen componentes mucho más pequeños que los que, de lo contrario, desplegarían si cada componente necesitara su propio espacio de proceso [Pinnock, 1998] y [Wang et al, 1997].

Con base a estas posibilidades ofrecidas por el estándar ActiveX, esta tecnología resulta idónea para la creación de sistemas distribuidos basados en componentes. Puesto que la interfaz de comunicación binaria entre componentes es idéntica para las ejecuciones remotas y local, se puede mover un componente entre equipos sin volver a compilar el componente o sus clientes.

En lo que se refiere a Internet, la tecnología ActiveX, da soporte a los siguientes elementos:

- ActiveX Controls: objetos interactivos programables por medio de eventos.
- ActiveX Documents. Permiten ver documentos que no son de HTML como por ejemplo Word o Excel.
- Active Scripting Controls: Simulan el comportamiento de algunos de los controles de ActiveX o Java Virtual Machine: es el código que permite al Internet Explorer 3.0 correr los applets de Java e integrados con los controles de ActiveX.

- ActiveX Server Framework: provee las funciones de seguridad, acceso a base de datos y otras, a traves de sus servicios para Internet, como el Internet Information Server (IIS), el servidor de transacciones (MTS),etc.

En cuanto a seguridad se refiere los controles de ActiveX creados por Microsoft o por otros desarrolladores autorizados cuentan con una firma electrónica contenida dentro de los archivos que serán transferidos por lo que garantiza que estos provienen de una fuente confiable, aunque se ha descubierto agujeros importantes en el esquema de seguridad de ActiveX, los cuales Microsoft se encuentra actualmente intentando reparar [Wang et al, 1997], para mas información sobre los recursos ofrecidos por Microsoft para internet, puede consultar la pagina principal de Microsoft en la sección de Tecnología, www.microsoft.com/dev/.

Invocación asíncrona.

Cuando un cliente realiza una llamada a un método de un componente, se bloquea hasta que vuelve la llamada. Es decir, el cliente no puede ejecutar código mientras está esperando. Esto se conoce como procesamiento síncrono. Al utilizar procesamiento asíncrono, puede liberar al cliente para realizar otras tareas mientras está esperando.

En el procesamiento asíncrono, la llamada al método que inicia una tarea vuelve instantáneamente, sin proporcionar ningún resultado. El cliente sigue con sus tareas, mientras el componente trabaja en la tarea. Cuando la tarea se ha completado, el componente avisa al cliente de que el resultado está listo.

El procesamiento asíncrono es útil también cuando los clientes necesitan obtener notificaciones de sucesos interesantes; por ejemplo, cambios en los valores de la base de datos o la llegada de mensajes. Un cliente indica a un componente que desea recibir notificaciones cuando ocurran ciertos sucesos y el componente envía notificaciones cuando estos ocurren. Ambos escenarios dependen de las notificaciones asíncronas. Mientras la aplicación cliente se está ocupando de sus tareas, llega una notificación de que una petición asíncrona se ha completado o de que ha ocurrido un suceso de interés.

Invocación asíncrona con devolución de llamada (CallBack).

La implementación del procesamiento asíncrono mediante métodos de devolución de llamada consta de dos partes. La primera parte es responsabilidad del autor de un componente.

El desarrollador de acuerdo con [Wang et al, 1997], deberia:

1. Definir las tareas o las notificaciones que se van a realizar.
2. Proporcionar una o más clases de creación externa para administrar las tareas o las notificaciones. Esta clase administradora también puede hacer el trabajo o se puede proporcionar una clase colaboradora para realizar el procesamiento real.
3. Crear una biblioteca de tipos que contenga la interfaz (o interfaces) que deben implementar los clientes para recibir notificaciones. Esta interfaz debe incluir todos los métodos a los que llamará el componente para notificar al cliente.
4. Proporcionar a la clase administradora los métodos a los que los clientes pueden llamar para iniciar las tareas o para solicitar notificaciones.

Nota Cada uno de estos métodos debe tener un argumento declarado como el tipo de interfaz definido en el paso 3, de forma que el cliente pueda pasar la interfaz que contiene la implementación del método apropiado de devolución de llamada.

5. Escribir código para iniciar la tarea o el proceso de vigilar los sucesos de interés.
6. Escribir código para llamar al método apropiado de devolución de llamada cuando se complete la tarea o cuando se observen los sucesos de interés.

Sugerencia: Si un cliente proporciona varios servicios asíncronos, puede que desee agrupar las devoluciones de llamadas relacionadas en interfaces distintas. Esto se debe a que una clase de un cliente debe implementar todos los métodos en una interfaz, tanto si los utiliza como si no. Por tanto, tener una interfaz grande con todos los métodos de devolución de llamada es un inconveniente para los clientes.

La segunda parte es responsabilidad del programador que utiliza el componente. El programador debe:

1. Crear una clase pública que implemente la interfaz definida por el autor del componente.
2. Escribir código para controlar las notificaciones en los métodos de devolución de llamada que utilizará el cliente.

3. Escribir código para solicitar una instancia de la clase administradora del componente.
4. Escribir código para llamar a los métodos que inician tareas o que solicitan notificaciones.

Los componentes ActiveX, soportan el manejo síncrono y asíncrono de eventos o invocaciones, también dan soporte a varios modelos de multihilado, con base ala arquitectura de Win32, para las versiones de Windows de 32-bits, para un estudio más amplio sobre el manejo de hilos consultar [Asche, 1996]. Para una referencia mas completa de Componentes ActiveX, puede consultar los siguientes sitios: http://msdn.microsoft.com/developer/ y http://www.microsoft.com/msdn/ o la bibliografia a la que se ha hecho referencia en es apéndice.

APENDICE B

MODELOS DEL COMPUTO DISTRIBUIDO Y PARALELO EN SISTEMAS ABIERTOS: Actores y Agentes.

La intensión de este apéndice es la de mostrar, los trabajos que se han y están realizando con relación a modelos y arquitecturas de computación en el área de sistemas distribuidos, paralelos e inteligencia artificial distribuida; pues se cree que los modelos de actores y agentes son dos de las teorías más fuertes e importantes en las áreas anteriormente mencionadas.

Sistemas abiertos.

Las Arquitecturas multiprocesador requieren herramientas para explotar el paralelismo. Una solución común consiste en indicar explícitamente, indicando las acciones a ejecutar simultáneamente, usando primitivas especificas. En ese caso se dice que el paralelismo es explícito, [Jennings et al, 1998]. La estructura general del programa sin embargo permanece secuencial y los correspondientes lenguajes son usualmente extensiones de lenguajes como C O Pascal. Este esquema es conveniente para arquitecturas paralelas con control centralizado, pero pobremente adaptado a redes de procesadores destinados a trabajar tan independientemente como sea posible [Jennings et al, 1998]. Por lo tanto Carl Hewitt considera que un sistema distribuido debería tener las siguientes características, [Hewitt et al, 1983]:

- Flexibilidad y capacidad para evolucionar: nuevas entidades deben ser fácilmente incorporadas en el sistema. Sin embargo, se torna difícil conocer algo sobre las entidades existentes a un tiempo determinado.
- Control descentralizado: en orden de evitar el embotellamiento.
- Cooperación remota, sin links rígidos entre componentes: ellos serán perjudiciales para la heterogeneidad del sistema, el cual deberá ser capaz de disparar subsistemas para cooperar.
- Conocimiento incompleto: cuando el mundo que será modelado es de tamaño y complejidad importantes, no es posible representar toda la información que lo caracteriza. Esto no será realista en el caso de asumir que ninguna solicitud será satisfecha.

- Extensibilidad e inconsistencia: si es necesario buscar repartir información a un resultado dado al tiempo (t), no es cierto que esto será el mismo resultado al tiempo (t+1).
- Cooperación basada en negociación: cada entidad tiene una conocimiento parcial del sistema completo y puede solo tener una limitada influencia sobre las otras entidades. El deberá negociar con ellos para completar su tarea.

Un sistema distribuido como estos, será entonces un sistema abierto. Numerosas arquitecturas son insatisfactorias considerando esas propiedades. Por ejemplo, *sistemas de pizarrón*, en ellas la información es compartida en lugar de ser comunicada, presentando embotellamiento, por ejemplo, la estructura de datos comunes a las cuales todas las fuentes de conocimiento deben tener acceso. No existe tal limitación con un sistema de actores, a partir de que no hay conocimiento global y las comunicaciones son sincrónicas. El paralelismo es intrínseco o implícito. Cada actor es una entidad independiente llevando solamente información local. El comportamiento total del sistema es, sin embargo, *difícil de apreciar.*

MODELO DE ACTOR.

En Inteligencia Artificial, es recomendado el uso de programas de propósito general que "piensen" sobre la integridad del problema a tratar, utilizando una base de conocimientos separada para ello. La validez de un modelo como el mencionado, en algunas oportunidades es rechazada, pues requiere de la recopilación de toda la pericia necesaria, para resolver el problema de esa base de conocimientos, ver [Hewitt et al, 1983] [Jennings et al , 1998].

Una alternativa propuesta por Marvin Minsky, especificada por en su libro " The Society of Mind " es crear especialistas cooperadores, conteniendo cada uno, una pericia restringida pero altamente especificada. Luego esas sociedad de especialistas se comunican entre si, para resolver cada uno la tarea que le compete para completar la resolución del problema integralmente, de acuerdo con [Jennings et al, 1998].

El modelo de actor pertenece a esa clase de ideas. Este modelo fue desarrollado en el MIT por Carl Hewitt y su grupo de investigadores, ahora llamado Message Passing Semantics Group, como una continuación de el Proyecto PLANNER.

La interface de un actor es llamada su "intencion" o "proposito" y define un compromiso entre el actor y el mundo exterior. Un actor solo conoce la intención o el propósito de sus conocidos (aquaintances) que pueden ser considerados como una abstracción de la representación fisica del actor. Un actor es mejor definido por su comportamiento, que por su estructura interna, [Hewitt et al, 1983].

Envío de mensajes.

El modelo de actores es completamente uniforme. Solo un tipo de evento puede ocurrir: un actor envía un mensaje a otro. Mas precisamente un actor mensaje, es trasmitido a un actor objetivo a través de un actor mensajero. Uno de los conocidos del actor mensajero es un actor sobre (envelope) que contiene el mensaje a ser trasmitido. Un mensaje también contiene una continuación, denotando el actor al cual el resultado del mensaje será trasmitido. Por defecto, la continuación es el remitente, pero también puede ser otro actor existente, como también un actor especialmente creado para ese propósito.

En el primer caso, el mensaje enviado posee la misma semántica como en los lenguajes basados en clases. En el segundo caso, son posibles comunicaciones sincrónicas: el actor que envía el mensaje no posee que esperar por una respuesta y puede continuar su actividad, mas detalles se presentan en [Hewitt et al, 1983] y [Sheremetov, 1997].

Comunicaciones.

Durante su existencia, un actor recibe mensajes que son procesados de acuerdo con su estado actual, comunicando información a sus conocidos (acquaintances) y creando nuevos actores. Solo un actor serializado puede cambiar su propio estado. Este procesa un mensage a la vez y explícitamente especifica su nuevo estado antes de procesar otro nuevo mensaje. Los mensajes como llegan son mientras tanto colocados en una lista de espera (queue). Como para el caso de actores no serializados, ellos pueden conceptualmente procesar un numero arbitrario de mensajes en paralelo (asincrónicos), ver [Sheremetov, 1997].

A modo de ejemplo veamos al ejemplo que se muestra a continuación, que es un administrador de existencias, en donde define un actor "articulo", con un conocido (acquintance) "cantidad".

Este fue particularmente influenciado por varios formalismos basados en el lambda-calculo, por el trabajo contemporáneo sobre SmallTalk-80 de Alan Kay, e indirectamente, por algunas ideas que adelantaron a Simula. El modelo de actores es bastante más teórico y no da indicaciones precisas respecto a su implementación, según se menciona en [Jennings et al , 1998].

Noción de actor.

Un actor puede ser visto como un experto viviendo en sociedad y comunicándose con otros expertos para resolver problemas. Cada experto puede por si mismo ser una sociedad de expertos más elementales. Como en el mundo, conocimiento y comportamiento son distribuidos entre los actores, quienes independientemente y al mismo tiempo llevan a cabo sus tareas.
En la forma de una clase, un actor encapsula dentro de la misma entidad una pequeña cantidad de conocimiento especifico y el significado para explotarlo, de acuerdo con [Hewitt et al, 1983] y [Sheremetov, 1997].

Esa pequeña cantidad de conocimientos son:

- Datos locales, llamados "conocidos" (acquaintances), que son los otros actores directamente conocidos por él, ellos corresponden a la instancia variable de una clase.

Y su significado para explotarlo es:

- Un comportamiento, que define la acción que el actor puede emprender durante su existencia, de acuerdo a los eventos surgidos. Como opuesto a una clase, el comportamiento no es definido a través de métodos pero si a través de scripts únicos, quienes filtran los mensajes recibidos de otros actores y activan la parte apropiada del comportamiento.

Este modelo provee un mecanismo de no-herencia. Un actor es una entidad autónoma que lleva dentro de si todo el conocimiento requerido para llevar a cabo tareas especificas. Esto puede ser comparado a una computadora que trabaja en la forma especificada script. Esta maquina puede posiblemente tener varios procesadores que le permitan ejecutar tareas simultáneamente.

Para proteger este conocido (aquintances) durante un movimiento dado en las existencias, el actor " articulo " es serializado. Para recibir un mensaje " Add " que incluye el " number " conocido de valor " qa ", el actor " articulo " modifica su estado utilizando la primitiva "become" y se convierte en un nuevo actor con "quantity" incrementado en un valor de "qa". Una vez que esta operacion ha sido completada, la siguiente operación en el conocido "quantity" puede ser ejecutado sin el riesgo de inconsistencia. El cambio de estado es llevado a cabo justo antes de enviar el mensaje "completion" indicando a la continuación "c" que la operación inicial de suma ha terminado. Este mensaje puede ser enviado concurrentemente con el procesamiento del siguiente mensaje. Como muestra el ejemplo el valor de un conocido nunca se modifica por una asignación pero si puede ser modificado por un cambio de estado, ver [Hewitt et al, 1983].

Se definen las acciones del actor por su repertorio (escenario) único que contiene una descripción de su comportamiento que abarca las siguientes alternativas de actividades: modificación de estado, actualización de la lista de destinatarios, el envío de mensajes, y la creación de nuevos actores. El proceso de comunicación esta basado en nombres no repetidos con que cuentan los actores de acuerdo a los papeles que ellos desempeñan. Un sistema actor evoluciona por una serie de efectos que resultan de los eventos que ocurren en el sistema, por ejemplo, del procesamiento de los mensajes.

Cuando un actor obtiene acceso a nueva información la propaga a los actores que conoce, los cuales a la vez, la propagan y posiblemente deduce nueva información de ella. De esta manera el conocimiento se distribuye entre todos los actores, de acuerdo con [Sheremetov, 1997].

Actores y clases

La abstracción de datos y la encapsulación son los fundamentos de los lenguajes basados en actores, así como los lenguajes basados en clases. Una clase, como un actor, agrupa datos locales y comportamientos. El comportamiento es descrito por métodos para el primero y por scripts para el segundo. Sin embargo, una clase principalmente se diferencia a partir de un actor en tres aspectos:

Una clase es cerrada a un tipo de datos que describe las características comunes para un juego de instancias modeladas sobre la estructura de datos descripta por la clase. Un actor es un prototipo único e independiente que tiene sus propios datos locales y comportamientos. Es creado por copia diferencial de otro actor y no mantiene relación particular con el.

Una clase no es activa pero tiene el comportamiento de sus instancias, que son activadas por transmisiones de mensajes sincronicas. Un actor es una entidad autónoma, que es activada por transmisión de mensajes asincrónicos. Una vez activa, un actor posee su propia vida y puede ser asimilado para procesar.
Visto que clases son ordenadas por jerearquias en un grafo de herencia, los actores coexisten al mismo nivel, sin discriminación. Delegación puede, sin embargo, establecer una relación de dependencia mutua entre algunos actores, los cuales pueden de ese modo compartir comportamientos.

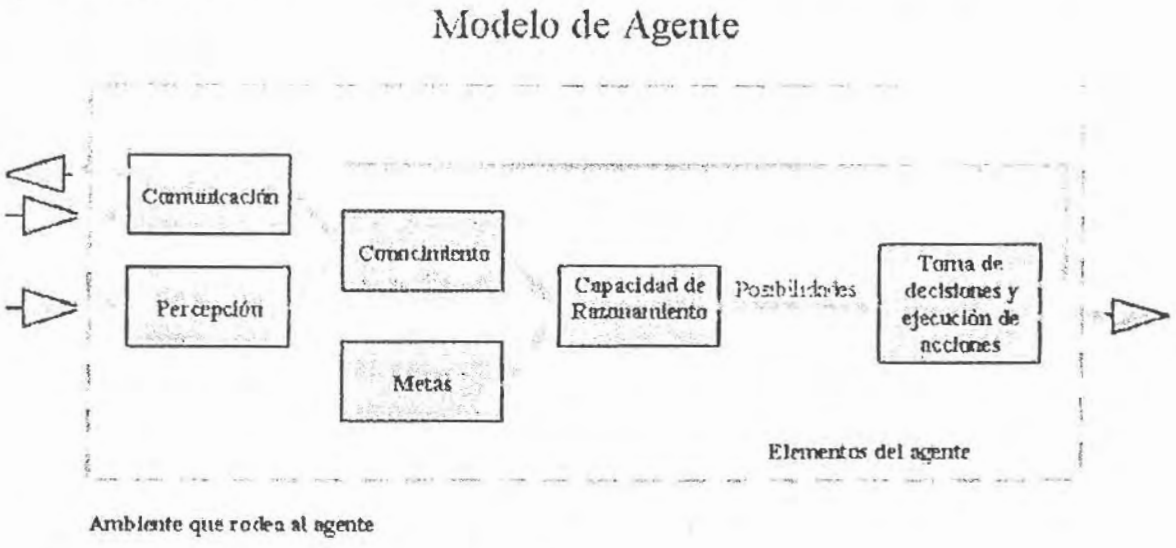
El paralelismo intrínseco de un modelo basado en actores es algo que perturba los hábitos tradicionales de programación. Por ejemplo, el uso y diseño de herramientas para examinar y depurar software es particularmente difícil: como la ocurrencia de un suceso en un sistema de actores solo parcialmente ordenados, sus ordenes cronológicos no corresponden a sus ordenes causales.
Muchos lenguajes basados en actores no son usualmente muy distribuidos y se encuentran lejos de ser productos bien terminados. Un caso opuesto seria por ejemplo SmallTalk-80. Sin embargo, la interesante pendiente para las arquitecturas paralelas y redes de procesadores podría tenerlos a prometedor futuro, y sin estar completamente definidas a campos de la Inteligencia Artificial, para realizar un estudio mas profundo se recomienda la bibliografía de [Hewitt et al, 1983] y [Jennings et al , 1998].

MODELO DE AGENTES.

Se le puede considerar al concepto de agencia como uno de los conceptos más importantes y emocionantes en los 90's tanto en la Inteligencia Artificial como en las tendencias principales de la Ciencia de la Computación, y en particular de sistemas distribuidos y paralelos. Existe un número de conceptos sobre un agente dependiendo del punto de vista sobre el procesamiento del conocimiento de manera distribuida. Desde el punto de vista del procesamiento distribuido se define el agente como *un proceso de software autosuficiente ejecutando en forma concurrente, que encapsula algún estado y tiene la capacidad para comunicarse con otros agentes por medio de la transmisión de mensajes, [Cockburn et al , 1996]* .

Definición y propiedades de agentes.

Como se describe en [Jennings et al , 1998] y [Cockburn et al , 1996], un sistema basado en agentes puede definirse como una red de "solucionadores de problemas" que trabajan juntos para dar solución a problemas que están más allá de sus capacidades individuales. Estos solucionadores de problemas, que se les llama "agentes", son autónomos y pueden ser homogéneos o heterogéneos en su naturaleza dependiendo de sus capacidades en la solución de problemas, ver figura que sigue.



En esta figura, podemos identificar que:

- Los agentes tienen conocimiento acerca del problema que tratan de darle solución. Pueden obtener este conocimiento preguntando información al usuario, percibiendo variables del medio ambiente que los rodea, o por medio de la comunicación con otros agentes.
- Tienen metas que deben alcanzar en el transcurso del tiempo.
- Son capaces de analizar un conjunto de posibles soluciones para lograr sus metas dependiendo de su capacidad de razonamiento.
- Tienen la capacidad de tomar decisiones para seleccionar la mejor opción.

En general el término de agente se usa para nombrar a un sistema (hardware o software) que tiene las siguientes propiedades, las cuales sirven a su vez para definir un modelo genérico de agentes.

Vamos a identificar y mencionar, las características o propiedades que los agentes, que en realidad es la parte medular de la teoría de agentes computacionales, esto lo visualizaremos, por medio de la figura que se muestra abajo, ver [Cockburn et al , 1996].

- Autonomía: Los agentes actúan sin la intervención directa de las personas, y tiene algún tipo de control sobre sus acciones y estado interno. Esto significa que debe poseer la capacidad de razonamiento, basado en el conocimiento, para generar posibles cursos de acción, y que de acuerdo a sus metas, sea capaz de tomar decisiones y ejecutar acciones acordes a sus metas.

- Sociabilidad: Los agentes interactúan con otros agentes mediante algún mecanismo de comunicación. Por lo anterior, se han desarrollado lenguajes de comunicación de agentes.

- Cooperación: En consecuencia de la sociabilidad, la cooperación entre agentes es una razón de ser para tener múltiples agentes para resolver problemas.

- Son reactivos: Los agentes perciben su ambiente (que puede ser el mundo real, un usuario vía interfaz gráfica, un conjunto de otros agentes, Internet, o la combinación de éstos) y responder a los cambios de éste. Lo anterior significa que los agentes deben tener elementos que instrumenten la capacidad de percibir variables del medio ambiente.

- Son proactivos: Los agentes no solamente actúan en respuesta a su ambiente, sino que son capaces de tener comportamiento orientado a metas al tener iniciativa.

- Aprendizaje: Los agentes, para ser inteligentes, requieren tener la propiedad de poder aprender del ambiente que les rodea.

- Veracidad: Se supone que un agente no debe comunicar en forma deliberada información falsa.

- Benevolencia: Se supone que los agentes no tienen metas conflictivas, y por lo tanto harán siempre lo que se les pide que hagan.

- Racionalidad: Se supone que un agente actuará para alcanzar sus metas en la medida que sus creencias, conocimientos y capacidad de razonamiento se lo permitan.

- Mobilidad: Algunos agentes pueden tener la habilidad de viajar en una red de computadoras (p.ej. en Internet –softbots)

Perspectivas de sistemas multiagentes.

De acuerdo a los tipos de agentes, es posible presentar ideas sobre trabajos de investigación a desarrollar en un futuro sobre sistemas basados en teoría de agentes, para un estudio mas detallado sobre la teoría de agentes, es posible consultar [Cockburn et al , 1996] y [Jennings et al , 1998], quienes presentan un panorama muy amplio y detallado sobre esta materia, sin embargo en la red internet, puede hallar un sinfín de información sobre esta materia.

Agentes colaborativos:

Diseño de metodologías y herramientas que permitan desarrollos más rápidos de sistemas basados en agentes colaborativos.
Mayor investigación en la teoría de coordinación en agentes.

Agentes de interfaz:

Demostrar que el conocimiento aprendido mediante agentes de interfaz puede realmente usarse para reducir cargas de trabajo.
Realización de experimentos con diferentes técnicas de aprendizaje de máquina en diferentes dominios para determinar cuáles técnicas de aprendizaje son mejores para cuáles dominios y porqué lo son.

Análisis del efecto de diferentes mecanismos de aprendizaje de máquina en las respuestas de los agentes.
Extender las capacidades de los agentes de interfaz para que sean capaces de negociar con otros agentes.

Agentes móviles:

Investigar problemas relacionados con la transportación de los agentes: ¿Cómo le hace un agente móvil para ir de un lugar a otro?
Investigar problemas relacionados con la autentificación: ¿Cómo se le hace para asegurar que el agente es realmente quien dice ser, y que éste verdaderamente representa a quien dice representar? ¿Cómo se puede asegurar que el agente móvil no se ha infectado con virus al navegar por varias redes?
Investigar problemas relacionados con la privacidad: ¿Qué se podría hacer para evitar que alguien pudiese leer el código de el agente de alguien, y ejecutarlo en nombre de otra persona?
Investigar problemas relacionados con el desempeño de las redes si en éstas viajan cientos, miles o millones de agentes móviles.
Investigar problemas relacionados con la interoperabilidad: ¿Cómo implementar servicios de directorios para que los agentes puedan localizar máquinas de información o servicios específicos?

Agentes de información/internet:

Sus perspectivas son muy semejantes a las de los agentes viajeros y de interfaz.

En general un sistema basado en agentes tiene grandes ventajas con respecto a un sistema centralizado y monolítico:

- Solución de problemas con mayor rapidez, debido al aprovechamiento de procesamiento paralelo.

- Comunicación mínima, pues se transmite solamente soluciones parciales de alto nivel a otros agentes en lugar de tener que enviar datos básicos a un sistema central.

- Mayor flexibilidad, pues se tienen agentes con diferentes habilidades que en forma dinámica cooperan entre sí para resolver problemas.

- Mayor confiabilidad, pues otros agentes pueden tomar las responsabilidades de los agentes que llegasen a fallar en su operación.

APENDICE C

OPCIONES DE DISEÑO FUZZY EN MATHWORKS FUZZY LOGIC TOOLBOX.

Fuzzy Logic ToolBox, permite que el usuario seleccione distintos parámetros del Sistema de Lógica Difusa; las Tablas 1 a 6 resumen estas opciones, clasificándolas en Tipo de Conjuntos Difusos, Tipo de Difusores, Tipo de Concresores, Opciones Matemáticas, Normas e Implicaciones.

Implicación	**AND**	**Composición**	**Unión/Intersección**
Mínimo	Mínimo	Mínimo	Máximo
Producto	Producto	Producto	Suma Acotada
Kleene-Dienes	Producto Acotado	Producto Acotado	Suma Drástica
Lukasiewcz	Producto Drástico	Producto Drástico	Mínimo
Zadeh	Familia Tp	Familia Tp	Producto
Estocástica	Familia Hamacher	Familia Hamacher	Producto Acotado
Goguen	Familia Sugeno	Familia Sugeno	Producto Drástico
Gödel	Familia Frank	Familia Frank	Familia Tp
Aguda	Familia Yager	Familia Yager	Familia Hamacher
	Familia Dubois-Prade	Familia Dubois-Prade	Familia Sugeno
			Familia Frank
			Familia Yager
			Familia Dubois-Prade

Tabla 1. Opciones de la Máquina de Inferencia

Tipo	Descripción[1]
Tipo L	Conjunto Difuso con función de pertenencia $f(x)=\begin{cases}1, x<a\\ (b-x)/(b-a), a>x>b\\ 0, x>b\end{cases}$
Triangulo	Conjunto Difuso con función de pertenencia $f(x)=\begin{cases}0, x<a\\ (x-b)/(a-b), a>x>b\\ (c-x)/(c-b), b>x>c\\ 0, x>c\end{cases}$
Pi	Conjunto Difuso con función de pertenencia $f(x)=\begin{cases}0, x<a\\ (x-b)/(a-b), a>x>b\\ 1, b>x>c\\ (d-x)/(d-c), c>x>d\\ 0, x>d\end{cases}$
Tipo Gamma	Conjunto Difuso con función de pertenencia $f(x)=\begin{cases}0, x<a\\ (x-a)/(b-a), a>x>b\\ 1, x>b\end{cases}$
Tipo Z	Conjunto Difuso con función de pertenencia $f(x)=\begin{cases}1, x<a\\ 1-2((x-a)/(b-a))^2, a>x>(a+b)/2\\ 2((x-b)/(b-a))^2, (a+b)/2>x>b\\ 0, x>b\end{cases}$
Campana	Conjunto Difuso con función de pertenencia $f(x)=\begin{cases}0, x<a\\ 2((x-a)/(b-a))^2, a>x>(a+b)/2\\ 1-2((x-b)/(b-a))^2, (a+b)/2>x>b\\ 1-2((x-b)/(c-b))^2, b>x>(b+c)/2\\ 2((x-c)/(c-b))^2, (b+c)/2>x>c\\ 0, x>c\end{cases}$
	Conjunto Difuso con función de pertenencia

[1] En esta tabla, a,b,c,d son reales definidos dentro del Universo de Discurso de la Variable Lingüística que contiene al ConjuntoDifuso, que satisfacen a<b<c<d.

PiCampana	$f(x)=\begin{cases}0, x<a\\ 2((x-a)/(b-a))^2, a>x>(a+b)/2\\ 1-2((x-b)/(b-a))^2, (a+b)/2>x>b\\ 1, b>x>c\\ 1-2((x-c)/(d-c))^2, c>x>(c+d)/2\\ 2((x-d)/(d-c))^2, (c+d)/2>x>d\\ 0, x>d\end{cases}$
Tipo S	Conjunto Difuso con función de pertenencia $f(x)=\begin{cases}0, x<a\\ 2((x-a)/(b-a))^2, a>x>(a+b)/2\\ 1-2((x-b)/(b-a))^2, (a+b)/2>x>b\\ 1, x>b\end{cases}$
Singlenton	Conjunto Difuso con función de pertenencia $f(x)=\begin{cases}0, x\neq x_0\\ 1, x=x_0\end{cases}$

Tabla 2. Tipos de Conjuntos Difusos Disponibles

Tipo	Descripción
Triangulo	difusor que genera Conjuntos Difusos tipo Triangulo
Pi	difusor que genera Conjuntos Difusos tipo Pi
Campana	difusor que genera Conjuntos Difusos tipo Campana
PiCampana	difusor que genera Conjuntos Difusos tipo PiCampana
Singlenton	difusor que genera Conjuntos Difusos tipo Singlenton

Tabla 3. Tipos de Difusores Disponibles

Tipo	Descripción
Mínimo	T_Norma definida por la relación $f(x,y)=min(x,y)$
Producto	T_Norma definida por la relación $f(x,y)=xy$
Producto Drastico	T_Norma definida por la relación $f(x,y)=\begin{cases}x, y=1\\ y, x=1\\ 0, x\neq 1 AND y\neq 1\end{cases}$
Producto	T_Norma definida por la relación

Acotado	$f(x,y) = max(0, x+y-1)$
Familia Tp	T_Norma definida por la relación $f(x,y) = 1-\left[(1-x)^p + (1-y)^p - (1-x)^p * (1-y)^p\right]^{1/p}$ $p \in \Re$
Familia Hamacher	T_Norma definida por la relación $f(x,y) = \frac{xy}{p+(1-p)(x+y-xy)}$ $p \geq 0$
Familia Sugeno	T_Norma definida por la relación $f(x,y) = min(1, x+y+p-xy)$ $p \geq -1$
Familia Frank	T_Norma definida por la relación $f(x,y) = \log_p\left(1+\frac{(p^x-1)(p^y-1)}{p-1}\right)$ $p > 0$
FamiliaYager	T_Norma definida por la relación $f(x,y) = 1-min(1,((1-a)^p+(1-b)^p)^{1/p})$ $p \geq 0$
Familia Dubois-Prade	T_Norma definida por la relación $f(x,y) = \frac{xy}{max(x,y,p)}$ $0 \leq p \leq 1$
Máximo	T_Norma definida por la relación $f(x,y) = max(x,y)$
Suma Acotada	T_Norma definida por la relación $f(x,y) = min(1, x+y)$
Suma Drástica	T_Norma definida por la relación $f(x,y) = \begin{cases} x, y = 0 \\ y, x = 0 \\ 1, x \neq 1 AND y \neq 1 \end{cases}$

Tabla 4. Tipos de Normas Disponibles

Tipo	Descripción
Producto	Implicación de Ingeniería definida por la relación $f(x,y) = xy$
Mínimo	Implicación de Ingeniería definida por la relación $f(x,y) = min(x,y)$
Kleene Dienes	Implicación Lógica definida por la relación $f(x,y) = \max(1-x,y)$
Lukasiewicz	Implicación Lógica definida por la relación $f(x,y) = min(1,1-x+y)$
Zadeh	Implicación Lógica definida por la relación $f(x,y) = max(min(x,y),1-x)$
Estocástica	Implicación Lógica definida por la relación $f(x,y) = max(1-x,xy)$
Goguen	Implicación Lógica definida por la relación $f(x,y) = min(1,\frac{y}{x})$
Godel	Implicación Lógica definida por la relación $f(x,y) = \begin{cases} 1, x \le y \\ y, x > y \end{cases}$
Aguda	Implicación Lógica definida por la relación $f(x,y) = \begin{cases} 1, x \le y \\ 0, x > y \end{cases}$

Tabla 5. Tipos de Implicaciones Disponibles.

Tipo	Descripción[2]
Primer Máximo	Concresor definido por la expresión $y^* = \inf_{y \in U}\{y \in U \mid \mu(y) = \sup \mu_B(y)\}$
Último Máximo	Concresor definido por la expresión $y^* = \sup_{y \in U}\{y \in U \mid \mu(y) = \sup \mu_B(y)\}$
Media De Máximos	Concresor definido por la expresión $y^*_1 = \inf_{y \in U}\{y \in U \mid \mu(y) = \sup \mu_B(y)\}$ $y^*_2 = \sup_{y \in U}\{y \in U \mid \mu(y) = \sup \mu_B(y)\}$ $y^* = \frac{(y^*_1 + y^*_2)}{2}$
Centro De Gravedad	Concresor definido por la expresión $y^* = \frac{\int_U y \mu_B(y) dy}{\int_U \mu_B(y) dy}$
Altura	Concresor definido por la expresión $y^* = \frac{\sum_{i=1}^{m} \overset{i}{y}\, \mu_{Bi}(\overset{i}{y})}{\sum_{i=1}^{m} \mu_{Bi}(\overset{i}{y})}$ $\overset{i}{y} = centroAltura(i)$ *centroAltura(i) es un parámetro del Conjunto Difuso de la Variable de Salida, que corresponde al Consecuente de la regla i; dicho parámetro intenta representar el centro del Conjunto Difuso.*

Tabla 6. Tipos de Concresores Disponibles.

Para una referencia más completa de funciones, para Conjuntos Difusos, Difusores, Concresores, Opciones Matemáticas, Normas e Implicaciones, puede consultar la bibliografía de [Gulley, 1997], y los manuales en línea, en formato PDF que vienen incluidos en el MathWorks MATLAB ver. 5.0.

[2] En esta tabla se supone que el Motor de Inferencia produce m Conjuntos Difusos para la Variable de Salida en cuestión, que está definida sobre un Universo de Discurso U; cada uno de los m conjuntos tiene una función de pertenencia $u_{Bi}(y)$, $i=1,2,..m$. Si el Concresor efectúa la Unión o la Intersección de los m conjuntos, el resultado es un Conjunto Difuso B que tiene función de pertenencia $u_B(y)$.

Bibliografía y Referencias.

[1] [Tanenbaum, 1993] Andrew S. Tanenbaum. ***Sistemas Operativos Modernos.*** Prentice-Hall Hispanoamericana S.A., México,1996.

[2] [Watson, 1998] Mark Watson. ***Creating JavaBeans.***, Morgan Kaufmann Publishers, Inc. 1998.

[3] [Brockschmidt, 1993] K. Brockschmidt. ***Inside OLE.*** Redmond, Washington: Microsoft Press, 1993.

[4] [Russell et al , 1996] Russell, Norving. ***Inteligencia Artificial, Un enfoque Moderno.***Prentice Hall, 1996.

[5] [Martin et al , 1997] Richard J. Martin, Glenn E. Weadock. **Bulletproffing Client/Server systems.** McGraw-Hill 1997.

[6] [Hwang et al , 1988] Kai Hwang, Faye A. Briggs. **Arquitectura de computadoras y procesamiento paralelo.** McGraw-Hill 1988.

[7] [Koulinitch et al , 1997] Anatoli S. Koulinitch, Francisco Espinosa, Carlos Benavides. ***Arquitectura basada en objetos de computación distribuida en la configuración de sistemas distribuidos.*** *Publicacion cientifica TEMAS* 2, Universidad Tecnologica de la Mixteca, - UTM, Mexico, 1997.

[8] [Sheremetov, 1997] Leonid B. Sheremetov. ***Marcos de comunicación para la resolución de problemas distribuidos en ingeniería concurrente.*** *TEMAS* 1, UTM, Mexico, 1997.

[9] [Pinnock, 1998] Jonathan Pinnock. ***Wrox Press, Developer's Journal, Professional DCOM Application Development.*** Volume 3 Issue 4 July/August 1998.

[10] [Wang et al, 1997] Y. M. Wang, O. P. Damani, and W.-J. Lee. ***Reliability and Availability Issues In Distributed Component Object Model (DCOM) (Position paper)***, in Proc. International Workshop on Community Networking (CN4), Sept. 1997. http://akpublic.research.att.com/~ymwang/papers/CN4.prn.doc

[11] [Yang et al, 1996] Zhonghua Yang and Keith Duddy.
In ACM Operating Systems Review, Vol.30, No. 2, April 1996. CORBA: A Platform for Distributed Object Computing (A State-of-the-Art Report on OMG/CORBA)). School of Computing and Information Technology Griffith University Nathan Queensland 4111 Australia.
http://www.cs.utah.edu/~machiraj/res/papers/corba.html

[12] [Vinoski, 1997] Steve Vinoski, ***CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments***, to appear in IEEE Communications Magazine, Vol.14, No.2, Feb.1997. http://www.cs.wustl.edu/~schmidt/vinoski.ps.gz

[13] [Bacon, 1994] Jean Bacon, University of Cambridge. ***Concurrent Systems: An Integrated Approach to Operating Systems, Database, and Distributed Systems***. Addison-Wesley 1994.

[14] [Cordón et al, 1997] O. Cordón, M.J. del Jesus, F. Herrera. ***A Proposal on Reasoning Methods in Fuzzy Rule-Based Classification Systems.*** Computer Science and Artificial Intelligence. Granada University, November 1997. ftp://decsai.ugr.es/pub/arai/tech_rep/ga-fl/tr-97126.ps.Z

[15] [Herrera et al, 1998] F. Herrera, E. Herrera-Viedma. ***Linguistic Decision Analysis: Steps for Solving Decision Problems under Linguistic Information: Multi-purpose decision making, linguistic modeling.*** Computer Science and Artificial Intelligence. Granada University February 1998. ftp://decsai.ugr.es/pub/arai/tech_rep/viedma/TR-98104.ps.Z

[16] [Siler, 1996] PhD. William Siler. Birmingham, AL 35217, USA. ***Building fuzzy expert systems Handbook 1996.*** http://users.aol.com/wsiler/manual.exe

[17] [Cockburn et al , 1996] D. Cockburn and N. R. Jennings: ***ARCHON: A Distributed Artificial Intelligence System for Industrial Applications, in Foundations of Distributed Artificial Intelligence*** (eds. G. M. P. O'Hare and N. R. Jennings) Wiley, 1996. ftp://ftp.elec.qmw.ac.uk/pub/isag/distributed-ai/publications/FOUND-DAI-ARCHON.ps.Z

[18] [Jennings et al , 1998] N. R. Jennings and M. J. Wooldridge. ***Applications of Intelligent Agents in Agent Technology: Foundations, Applications, and Markets*** (eds. N. R. Jennings and M. Wooldridge). 1998 ftp://ftp.elec.qmw.ac.uk/pub/isag/distributed-ai/publications/agt-technology.pdf

[19] [INEGI, 1994] **Anuario Estadistico del estado de Oaxaca**. Edicion 1994. INEGI, Gobierno del estado de Oaxaca.

[20] [Expersoft, 1997] Expersoft Corporation 1997. ***Bridging the Gap Between CORBA and COM.*** http://www.expersoft.com/Products/CORBAActiveX/white.htm

[21] [Asche, 1996] Asche, Ruediger. ***Win32 Multithreading Performance.*** Microsoft Developer Network Technology Group. Development Library. Enero 1996. http://www.microsoft.com/win32dev/base/threadli.htm

[22] [Jackson, 1990] Peter Jackson. ***Introduction to Expert Systems***. Addison-Wesley,

[23] [Sucar, 1994] L. Enrique Sucar. Structure and Parameter Learning in Probabilistic Networks. ***Memorias de la XI Reunión Nacional de Inteligencia Artificial,*** Soc. Mex. de Intel. Artif., Universidad de Guadalajara. Limusa, 1994.

[24] [Gulley, 1997] Ned Gulley. ***Fuzzy Logic ToolBox User's Guide.*** The MathWorks, Inc. 1997.

[25] [Gerlhof et al, 1994] Carsten A. Gerlhof, Alfons Kemper. ***A Multi-Threaded Architecture for Prefetching in Object Bases.*** Proc. of the 4th Intl. Conf. Extending Database Technology (EDBT), Lecture Notes in Computer Science (LNCS), Cambridge, England, March 1994, Springer-Verlag.
http://dodgers.fmi.uni-passau.de/~gerlhof/papers/edbt94.ps

[26] [Tsichritzis et al, 1992] Dennis Tsichritzis, Oscar Nierstrasz and Simon Gibbs. ***Beyond Objects: Objects.*** IJICIS (International Journal of Intelligent & Cooperative Information Systems), vol. 1, no. 1, 1992.
http://cuiwww.unige.ch/OSG/publications/OO-articles/beyondObjects.ps.Z

[27] [Chandy et al, 1997] K. Mani Chandy, Joseph Kiniry, Adam Rifkin, and Daniel Zimmerman . **A Framework for Structured Distributed Object Computing**
http://www.infospheres.caltech.edu/papers/framework/framework.ps

[28] [Chandy et al, 1996] K. Mani Chandy, Paolo A.G. Sivilotti, Joseph R. Kiniry
A Cottage Industry of Software Publishing: Implications for Theories of Composition. Department of Computer Science, California Institute of Technology, m/c 256-80,
http://cuiwww.unige.ch/OSG/publications/OO-articles/objectOrientedInterop.ps.Z

[29] [Nierstrasz et al, 1992] Oscar Nierstrasz, Simon Gibbs and Dennis Tsichritzis . **Component-Oriented Software Development.** Genova University, Switzerland.
http://cuiwww.unige.ch/OSG/publications/OO-articles/componentOrientedSoftDevt.ps.Z

[30] [Nierstrasz , 1993] Oscar Nierstrasz. **Composing Active Objects**. University of Geneva, Switzerland. In: Research Directions in Object-Based Concurrency, ed. G. Agha, P. Wegner, A. Yonezawa, MIT Press, 1993, pp. 151-171.
http://cuiwww.unige.ch/OSG/publications/OO-articles/composingActiveObjects.ps.Z

[31] [Bruun et al, 1996] Bent Bruun Kristensen, Daniel C. M. May. **Component Composition and Interaction.** Proceedings of International Conference on Technology of Object-oriented Languages and Systems (TOOLS PACIFIC 96), Melbourne, Australia, 1996, available at: http://www.cs.auc.dk/~bbk/research/hook43.ps

[32] [Minsky et al, 1995] Naftaly Minsky and Jerrold Leichter. **Law-Governed Linda as a Coordination Model,"** *Object-Based Models and Languages for Concurrent Systems*, P. Ciancarini, O. Nierstrasz and A. Yonezawa (Ed.), LNCS 924, Springer-Verlag, 1995, pp. 125-146. http://www.cs.rutgers.edu/~minsky/public-papers/linda-paper.ps

[33] [Hewitt et al, 1983] Carl Hewitt and Peter de Jong .**Analyzing the Roles of Descriptions and Actions in Open Systems.** ftp://publications.ai.mit.edu/ai-publications/500-999/AIM-727.ps

[34] [Olson et al, 1992] David Olson L, James F. Courtney, Jr. **Decision support Models and Sysytms**. -- Singapur: Maxwell Macmillan International; 1992.

[35] [Negrete, et al, 1996] Jose Negrete, Pedro Pablo Gonzalez, Alejandro guerra. **Pericia Artificial: Un aprendizaje constructivista de Sistemas Expertos**. Mexico: Universidad Veracruzana 1996.

[36] [Ignizio, 1991] James P. Ignizio. **Introduction to expert systems: The development and implementation of rule-based Expert Systems**. Singapur: McgrawHill, 1991.

[37] [Jang, 1993] J.-S. Roger Jang, **ANFIS: Adaptive-Network-Based Fuzzy Inference Systems**, IEEE Transactions on Systems, Man, and Cybernetics, Vol. 23, No. 03, pp 665-685, May 1993. http://www.cs.nthu.edu.tw/~jang/ps/anfis.ps.Z

[38] [Jang et al, 1995] J.-S. Roger Jang and C.-T. Sun, **Neuro-Fuzzy Modeling and Control**, The Proceedings of the IEEE, Vol. 83, No. 3, pp 378-406, March 1995. http://www.cs.nthu.edu.tw/~jang/ps/review.ps.Z

[39] [INEGI, 1997] **Anuario Estadistico del estado de Oaxaca**. Edicion 1998. INEGI, Gobierno del estado de Oaxaca.