UNIVERSIDAD NACIONAL AUTONOMA
DE MEXICO

U.A.C.P. Y P. DEL C.C.H.
I.I.M.A.S.

UN SISTEMA PARA EL CONTROL DE DIALOGOS ENTRE FACILITADORES

T E S I S

QUE PARA OBTENER EL GRADO DE:
MAESTRO EN CIENCIAS DE LA COMPUTACION
P R E S E N T A :
LUIS ALFONSO GONZALEZ LUNA

MEXICO, D. F. ENERO 1997

Lo repito: basta que un libro sea posible para que exista. Sólo está excluido lo imposible. Por ejemplo: ningún libro es también una escalera, aunque sin duda hay libros que discuten y niegan y demuestran esa posibilidad y otros cuya estructura corresponde a la de una escalera.

J.L. Borges. *La Biblioteca de Babel*

Agradecimientos

Deseo agradecer en primer lugar a mi director de tesis, Dr. Víctor Germán Sánchez Arias, tanto por la orientación proporcionada como por los consejos y la paciencia demostrada a lo largo de este trabajo. También a mis demás sinodales, cuyas revisiones y sugerencias mejoraron este escrito. Especialmente al Dr. Christian Lemaître León, tanto por la orientación en áreas no familiares como por todos los apoyos de toda índole prestados durante este tiempo. A la Dra. Hanna Oktaba, por todo su esfuerzo, apoyo e interés para sus alumnos, que no pasan desapercibidos para nosotros. Al M. en C. Horacio Carvajal y al Dr. Armando Maldonado, cuyos comentarios permitieron mejorar la presentación de las conclusiones del trabajo e identificar más líneas de desarrollo futuro.

También agradezco a la Dirección General de Apoyo al Personal Académico de la UNAM por el apoyo prestado para la realización de este trabajo, y muy especialmente al Laboratorio Nacional de Informática Avanzada, A.C. (LANIA) y a su directora, Dra. Cristina Loyo Varela, por facilitar los medios para desarrollarlo. Asimismo, y aunque sea de una manera multitudinaria e insuficiente, agradezco a todas aquellas personas e instituciones que hacen libremente disponible su trabajo y aportaciones en medios como Internet, por compartir sus conocimientos y dar acceso inmediato a la información.

Finalmente, agradezco a todas las personas que durante este tiempo, me apoyaron y aguantaron de varias maneras: especialmente a mis padres, mi hermano, y los buenos amigos que afortunadamente tengo en Xalapa, Puebla y México.

Xalapa, Ver., a 20 de noviembre de 1997

Presentación

El presente trabajo se desarrolló en el contexto del proyecto "Redes cooperativas heterogéneas" (CONACYT 3345-A9308), dirigido en LANIA por el Dr. Christian Lemaître. Su objetivo es proponer e implementar un modelo para la comunicación entre sistemas cooperativos heterogéneos basados en facilitadores [Lemaître *et al*, 1993], para lo cual se revisan los mecanismos de comunicación empleados normalmente en el área de sistemas abiertos y las caracteristicas de la comunicación entre agentes inteligentes, comparándolos para ubicar las ventajas e inconvenientes de asociar cualidades mentales a los participantes.

En el primer capítulo del trabajo se presentan los sistemas abiertos heterogéneos y algunas de las arquitecturas y mecanismos empleados para construirlos, buscando las caracteristicas que los distinguen. Se ven los métodos empleados normalmente (RPC, DDE, y los ambientes basados en objetos), y los métodos basados en agentes y sus posibilidades (el modelo de agentes de Shoham, las sociedades de Hewitt, y el modelo de facilitadores desarrollado en LANIA), para comparar y obtener conclusiones.

En un sistema abierto heterogéneo, el factor común entre los participantes es la capacidad de comunicación, asi que se busca resaltar su papel a través de un modelo de comunicación expresivo y general que esté al alcance de todos. Con ese fin, en el segundo capítulo se ven teorías de comunicación surgidas de las áreas de filosofia del lenguaje y procesamiento del lenguaje natural que consideran los aspectos mentales de los participantes. Se presentan la teoria de actos de habla (que reconoce el papel de las intenciones en la comunicación), el *Agent Communication Language* (ACL, que usa los actos de habla para plantear un lenguaje de comunicación entre agentes), y la teoria de estructura del discurso de Grosz y Sidner (que provee un modelo estructurado para el procesamiento e interpretación de los mensajes recibidos).

Las conclusiones obtenidas se emplean en el tercer capítulo para elaborar un diseño del sistema de diálogos de los agentes facilitadores, estructurando el intercambio de actos de habla. Se presentan los niveles que forman al sistema, su funcionalidad y las decisiones de diseño adoptadas, comparándolo con otros desarrollos como KAPI (una implementación que ayuda a emplear KQML, uno de los componentes de ACL). También se presenta el diseño de un nivel de acciones del facilitador, que interactúa con el sistema de diálogos y se empleó para su validación.

Finalmente, se presentan los resultados obtenidos con la implementación y validación del sistema, y las conclusiones obtenidas durante el trabajo. Se describen las interfaces de programación desarrolladas, una aplicación que utiliza el sistema, y las posibilidades de desarrollo futuro del trabajo.

Contenido

Capítulo 1

Sistemas Abiertos

La importancia creciente de los sistemas de cómputo en las empresas y su proliferación en otros campos han forzado a los proveedores de equipos a diseñar sistemas "abiertos", cuyos componentes de hardware y software son tratados como cajas negras que tienen interfaces de uso estándares y bien definidas. Estos componentes son intercambiables con otros que cumplan las mismas especificaciones, permitiendo:

- integrar soluciones con múltiples proveedores;
- reducir los costos de soporte y mantenimiento;
- introducir nuevas tecnologías sin modificar otras partes del sistema;
- la interconexión con otros sistemas, creando un ambiente común de trabajo (como en las redes de computadoras).

El área de la interconexión es especialmente interesante, ya que causa expectativas mayores entre los usuarios: una vez que dos sistemas están en un ambiente común, se hace deseable la interacción entre ellos, a menudo en formas no previstas por los diseñadores de cada sistema. Por ejemplo:

- En los sistemas de interfaz gráfica al usuario (GUIs como X-Window, Windows y Macintosh), como las aplicaciones tienen una interfaz semejante y el usuario las ve compartiendo el mismo espacio en pantalla, se refuerza su expectativa de intercambiar datos entre ellas y verlas funcionar conjuntamente.
- Las redes de cómputo y los sistemas distribuidos dentro de una organización permiten el acceso a los datos independientemente de su localización, pero aún hace falta adaptar o adecuar los datos para que sean útiles en los sistemas que podrían aprovecharlos: un banco que disponga en su red de cómputo de un sistema experto para la asignación de créditos y otro para la evaluación de proyectos querrá emplear los datos del evaluador durante la asignación de créditos de la forma más sencilla posible, a pesar de que los sistemas hayan sido desarrollados independientemente.

En ambos casos, la interacción que en realidad se desea es la **cooperación**: los sistemas en el ambiente común deben poderse emplear juntos para alcanzar fácilmente una meta. Los participantes en el ambiente (programas clientes, servidores, basados en conocimiento, o los mismos seres humanos) requieren un componente de cooperación hacia los demás sistemas interconectados. Tradicionalmente, este componente de cooperación se ha implementado a través de mecanismos específicos (como los programas convertidores de archivos y la intervención de operadores humanos), pero conforme

aumenta la complejidad de las aplicaciones y la cantidad de recursos disponibles en una red, estas soluciones muestran limitaciones:

- incrementan el trabajo de administración (ya que al añadir un nuevo sistema al ambiente hay que crearle sus convertidores y darlo a conocer a los usuarios);
- los usuarios necesitan saber qué sistemas pueden ayudarles, cómo utilizarlos, y cómo combinarlos para obtener los resultados que buscan.

Por ello, es necesario buscar otras maneras de tener cooperación entre aplicaciones.

¿Qué mecanismos hay disponibles para lograr la interacción entre aplicaciones? Dentro de una misma máquina, los sistemas de interfaz gráfica al usuario como Macintosh, X-Window y Windows han desarrollado mecanismos como los portapapeles o *clipboards*, protocolos de comunicación entre aplicaciones (DDE en Windows, Apple Events en Macintosh), y sistemas de soporte basados en objetos que pueden integrarse para crear ambientes orientados a documentos (OpenDoc, COM/OLE en Windows, SOM en OS/2)[1]. Para ambientes distribuidos, se han empleado sistemas basados en mensajes, llamadas a procedimientos remotos (RPC), y objetos distribuidos (CORBA).

Por otro lado, se pueden obtener respuestas diferentes a la búsqueda del ambiente cooperativo si se adopta un punto de vista basado en intenciones. Los sistemas computacionales que están disponibles en un ambiente común sirven algún propósito específico (de otra forma, no estarían allí), y otro tanto puede decirse de los usuarios (que buscan obtener resultados empleando los servicios ofrecidos por los sistemas o por otros usuarios en el caso de una red). Ambos tipos de participantes tienen en común la posesión de algún tipo de conocimiento, y las intenciones de emplearlo de determinada manera: haciéndolo disponible a otros, realizando actividades con él, o ambas cosas. Este enfoque de **agentes**, donde se habla de conocimiento, intenciones y modelos mentales, ofrece un nivel de abstracción conveniente para diseñar estrategias cooperativas y otras interacciones inteligentes entre los participantes, independientemente de que sean sistemas computacionales o seres humanos.

En este capítulo se muestran algunos de los protocolos empleados actualmente para la comunicación entre aplicaciones, comentando sus ventajas y limitaciones. Después se introduce el modelo de agentes de Shoham como paradigma de la programación orientada a agentes, junto con el modelo de sistemas abiertos de Hewitt como muestra de organización de una sociedad basada en sistemas cooperativos de esta naturaleza. Finalmente se presenta la arquitectura de agentes facilitadores, desarrollada en LANIA y a la cual pertenece este trabajo.

---

[1] Los sistemas basados en objetos son especialmente interesantes porque son extensibles a ambientes distribuidos, y en buena parte complementan los sistemas presentados en este trabajo; por su importancia, se les discute en las conclusiones de la tesis.

1.1 Interacción entre aplicaciones

En esta sección se presentan algunos de los mecanismos existentes que ofrecen servicios básicos para el desarrollo de sistemas abiertos, permitiendo a una aplicación ofrecer servicios o interactuar con otras en mayor o menor medida, sin necesidad de conocerlas de antemano. Los mecanismos que se presentan aquí son DDE, RPC, y KQML, y los ambientes de objetos distribuidos CORBA y DCOM.

1.1.1 Remote Procedure Calls (RPC)

Las llamadas a procedimientos remotos o RPC [Corbin, 1991; Lockhart, 1994] estructuran los sistemas en una arquitectura cliente/servidor, donde una aplicación ofrece servicios a las demás registrando procedimientos que pueden ser invocados desde la misma máquina o a través de la red. Hay dos protocolos incompatibles que usan el mismo nombre (Open Network Computing RPC de Sun, y Distributed Computing Environment RPC de la Open Software Foundation), pero sus principios son los mismos: por cada servicio que emplea, un cliente tiene un procedimiento local llamado *stub* que invoca al procedimiento remoto, codificando y transmitiendo al servidor los parámetros en un formato de representación externa de datos, y recibiendo y descodificando los resultados de la ejecución (ver figura 1.1).

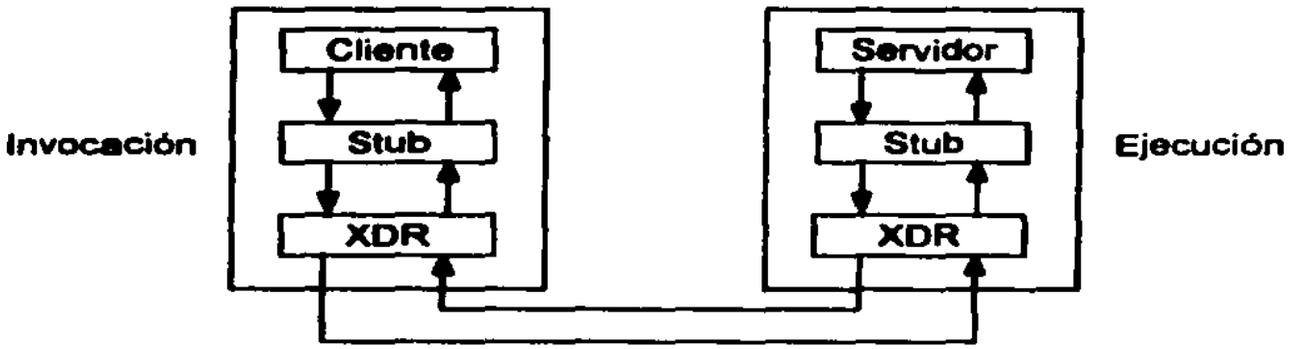

Figura 1.1.- Esquema de llamada a un procedimiento remoto

Típicamente, el programador emplea un lenguaje de descripción (RPCL en ONC RPC, IDL en DCE RPC) para especificar la firma (tipos de datos de los parámetros y resultado) de cada procedimiento exportado por el servidor. Como los compiladores de RPCL e IDL generan los *stubs* para los clientes y el servidor, los programadores sólo tienen que implementar los servicios e invocar los stubs para tener una aplicación distribuida. Estos lenguajes permiten también definir tipos de datos y soportan la noción de versiones de servicios, por lo que se pueden intercambiar datos con estructura compleja y se simplifica en cierta medida el problema de la actualización de programas.

Para que un cliente pueda ubicar un procedimiento remoto en tiempo de ejecución, debe conocer la identificación del servicio, la máquina que lo soporta, y el puerto de comunicación en que se encuentra. En ONC RPC, cada servidor tiene un programa llamado *portmapper*, en el que se registran los servicios soportados localmente; los clientes pueden contactar al *portmapper* para invocar indirectamente los servicios o preguntarle en qué puerto están para ejecutarlos por su cuenta. DCE RPC emplea un mecanismo más sofisticado y realmente distribuido.

1.1.2 Dynamic Data Exchange (DDE)

Windows de Microsoft contiene desde la versión 2.0 un protocolo orientado a la comunicación entre aplicaciones, llamado Intercambio Dinámico de Datos (DDE por sus siglas en inglés). Aunque en Windows 3.1 DDE sólo comunica aplicaciones dentro de una misma máquina, existen extensiones en otras versiones que ofrecen comunicación en red.

DDE [Microsoft, 1992] consiste de un subconjunto de mensajes de Windows reservado para la cooperación entre aplicaciones, y una serie de convenciones para manejarlos. Se basa en una arquitectura donde las aplicaciones buscan servidores que satisfagan sus requerimientos, estableciendo conversaciones punto a punto con ellos sobre un tema de su interés (correspondiente a una clasificación general del tipo de información que ofrece el servidor), e intercambian datos o artículos de manera asíncrona a lo largo de la conversación:

- Un programa de hojas de cálculo puede funcionar como servidor interpretando como temas los nombres de los archivos que crea, y como artículos las celdas mismas de cada hoja.
- Un programa de cotizaciones que accesa en tiempo real la información de las bolsas de valores puede manejar como temas los nombres de las bolsas y como artículos los nombres de las acciones cotizadas.

Una aplicación inicia una conversación mandando a todas las demás una solicitud de conexión (`WM_DDE_INITIATE`), que contiene el nombre del servicio que busca (típicamente el nombre de un programa) y el tópico o tema de su interés; como se permite usar comodines, el cliente puede contactar simultáneamente a varios servidores sobre diferentes temas. Las aplicaciones que reciben solicitudes de conexión determinan si soportan o no el servicio y tema especificados, y si aceptan la solicitud de conversación, confirman al cliente mediante un mensaje de asentimiento positivo `WM_DDE_ACK`. A partir de ese momento, ambos programas realizan intercambios de información llamados transacciones, que son respondidas mediante asentimientos positivos y negativos, y cuyo éxito o fallo no afecta al estado general de la conversación. La conversación termina hasta que ambas partes envían un `WM_DDE_TERMINATE`.

Las transacciones que el cliente puede realizar con un servidor son:

- Depósito de datos (WM_DDE_POKE)
- Ejecución de comandos (WM_DDE_EXECUTE)
- Solicitud de datos (WM_DDE_REQUEST), en cuyo caso el servidor entrega la información requerida a través de un mensaje WM_DDE_DATA. La figura 1.2 presenta una transacción WM_DDE_REQUEST.
- Establecimiento de ligas con el servidor, que permiten al cliente subscribirse a información de su interés y recibir notificaciones asíncronas cuando ésta se actualice. Estas ligas se inician y terminan con los mensajes WM_DDE_ADVISE y WM_DDE_UNADVISE, y son de dos tipos: calientes, donde el servidor entrega el nuevo valor del dato junto con la notificación; y tibias, en las que el cliente debe pedir explícitamente el nuevo valor del dato, si es que le interesa (ver figura 1.3).

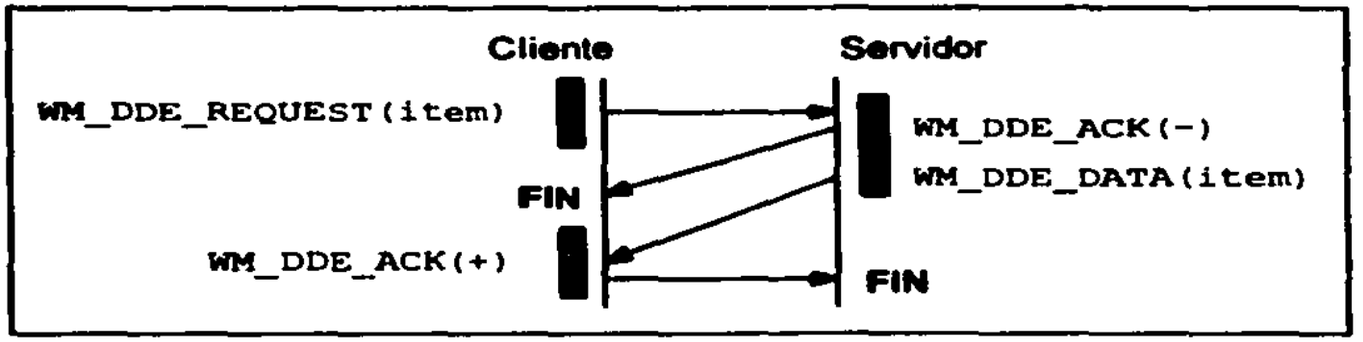

Figura 1.2.- Transacción WM_DDE_REQUEST

Como DDE resulta bastante complicado de programar al nivel de envío y recepción de mensajes, Microsoft definió en la versión 3.1 de Windows la *DDE Management Library* (DDEML), una biblioteca que ofrece una interfaz de programación de más alto nivel al formalizar las transacciones mencionadas, automatizando la aceptación y rechazo de conexiones, y permitiendo al usuario definir funciones *callbacks* en cada transacción, para que la biblioteca las llame cuando lleguen mensajes con datos [Microsoft, 1992].

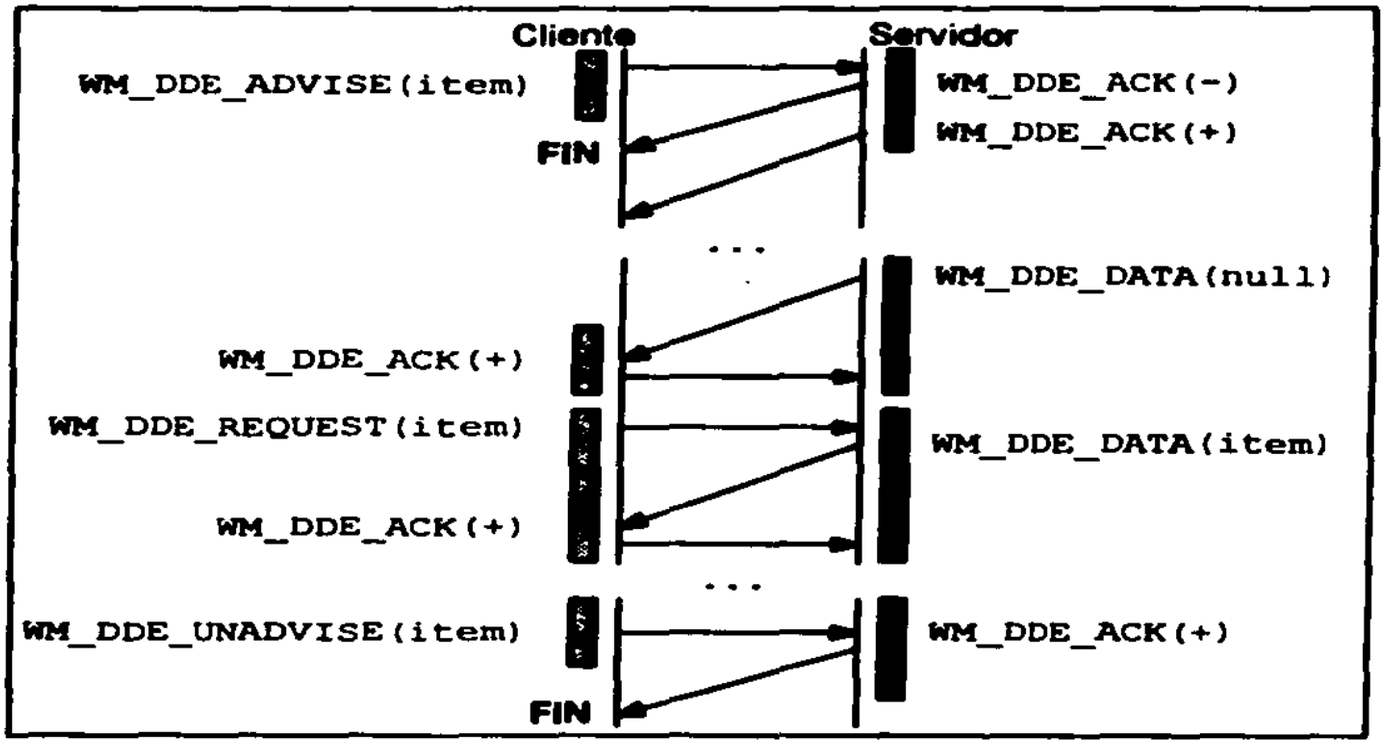

Figura 1.3.- Una liga tibia

1.1.3 Ambientes basados en objetos

Los ambientes de objetos distribuidos representan una evolución natural de los sistemas de procedimientos remotos, y suelen aplicarse también a la interacción entre aplicaciones porque soportan el desarrollo de sistemas dentro de un modelo de componentes. Algunos de estos posibles ambientes son DCOM [Box, 1996] y CORBA [OMG, 1996], que conceptualmente es el más probado y maduro de los dos.

En CORBA (*Common Object Request Broker Architecture*), cada objeto remoto está representado localmente por un objeto *stub*, que recibe las invocaciones de operaciones y emplea una capa de software llamada **Object Request Broker** (ORB) para codificarlas y transportarlas al lugar donde se encuentra el objeto remoto (ver figura 1.4). El tipo de datos de un objeto, representado por sus atributos y los métodos que pueden invocarse sobre él, se describe en interfaces escritas en IDL (lenguaje de definición de interfaces), y se compila para generar automáticamente un esqueleto de implementación del objeto remoto y el código de los objetos *stubs*. A diferencia de RPC, la información de tipos queda disponible en un repositorio de interfaces y en un repositorio de implementaciones.

por lo que cualquier cliente puede consultarla en tiempo de ejecución para invocar y ofrecer los servicios de objetos no conocidos al momento de compilación.

La implementación de un objeto típicamente usa el ORB a través de un **adaptador de objetos**. El propósito de este componente es encapsular los detalles de activación, desactivación y ejecución propios del objeto: si es o no persistente, su lugar de residencia (memoria, base de datos, u otra parte), si soporta *multithreading*, etc.

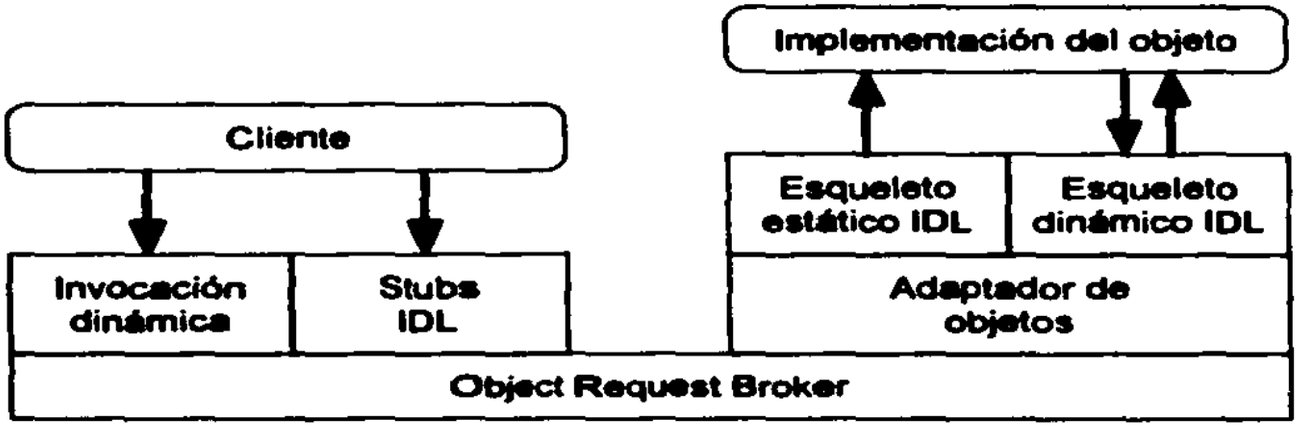

Figura 1.4.- Arquitectura de CORBA

Una arquitectura CORBA se complementa con una serie de objetos e interfaces estándares que proporcionan servicios tales como localización de objetos, establecimiento de protocolos (como propagación de eventos) y transacciones.

El estándar de CORBA deja muchos puntos libres para los implementadores, por lo que los ORB varían substancialmente y no son naturalmente compatibles entre sí. Para lograr la interoperabilidad entre ORBs y facilitar la construcción de sistemas de objetos más grandes, OMG definió el protocolo *General Inter-ORB Protocol* (GIOP).

1.2 Agentes y modelos mentales

La noción de agente se ha empleado para designar genéricamente a los programas autónomos que realizan alguna tarea en representación de algún usuario o sistema. Para fines de este trabajo, se entiende como agente a un ente que tiene además cualidades mentales como metas, intenciones, creencias y conocimientos propios, que existe dentro de una sociedad de semejantes con los que puede comunicarse en un lenguaje común, y que es capaz de observar y modificar el ambiente en el que se encuentra.

La cooperación puede verse como una forma especial de interacción, en la que los agentes comparten sus metas y realizan acciones para alcanzarlas durante un intervalo extenso de tiempo [Zachary y Robertson, 1989]. La ventaja de usar metas e intenciones es el nivel de abstracción obtenido, que es apropiado para diseñar estrategias cooperativas e interacciones inteligentes entre los sistemas, independientemente de si se trata de sistemas computacionales o de seres humanos representados por programas. La habilidad de los agentes para cooperar, junto con su propiedad de autonomía, permite que se les delegue parte de la solución a un problema complejo, abatiendo costos de desarrollo.

Actualmente se han desarrollado sistemas basados en agentes, entre los que destaca ARCHON, el sistema multiagente más grande de Europa. ARCHON ha sido empleado en problemas complejos, como la distribución y transmisión de electricidad, el control de aceleradores de partículas, y aplicaciones robóticas [Jennings, 1994].

Los trabajos desarrollados sobre agentes pueden dividirse en las siguientes categorías generales [Wooldridge y Jennings, 1994]:

- Modelos teóricos: Tratan sobre la definición de los agentes, las propiedades que deben tener, y cómo hay que representarlas y razonar sobre ellas. Son de especial importancia porque dan la consistencia del sistema de agentes y determinan formalmente sus capacidades y alcances.
- Arquitecturas de agentes: Cómo construir agentes que satisfagan las propiedades que se esperan de ellos. Esto comprende también las organizaciones de agentes: la conexión de los sistemas de agentes de acuerdo a protocolos constitutivos.
- Lenguajes de programación: Cómo implementar agentes, cuáles son las primitivas apropiadas para esta tarea, y cómo ejecutarlas.

En las siguientes secciones se presenta más a detalle el modelo de programación orientada a agentes, complementándolo con la descripción de los Sistemas Abiertos de Hewitt, que cubren el aspecto organizacional de una red multiagente.

1.2.1 Programación Orientada a Agentes

Shoham [1990] propone un modelo de programación basado en la descripción y manipulación de estados mentales al que denomina Programación Orientada a Agentes (POA), por considerarlo como especialización de la programación orientada a objetos (POO). POA ejemplifica las áreas de agentes mencionadas: no descuida la teoría, proporciona una arquitectura de agentes (aunque con énfasis en el individuo), y define un lenguaje de programación, llamado AGENT0.

La unidad básica de cómputo son los agentes, que al igual que los objetos en POO, envían y reciben mensajes que modifican su estado interno, descrito en base a cualidades

mentales. Los mensajes son actos de habla[2], por lo que incluyen una fuerza ilocutiva y un contenido proposicional, expresando así la intención del emisor. No se supone que los agentes son "racionales" más allá de considerar que sus aspectos mentales son interna y mutuamente consistentes.

Hay varias definiciones de lo que puede considerarse como estado mental, y unas son más apropiadas que otras para ciertas aplicaciones[3]. Shoham propone partir de un conjunto de suposiciones básicas: en cualquier instante, el futuro está determinado por la historia pasada (que determina el estado actual del agente) y las acciones que el agente realice (porque modifican el estado actual). Las acciones tomadas son a su vez determinadas por las decisiones o elecciones del agente; las decisiones se encuentran acotadas por las creencias que el agente tenga acerca del estado del mundo, de sus propias capacidades, y de las capacidades y estado mental de otros agentes. De aquí que los elementos básicos para este modelo de agentes son:

- **Tiempo**: los eventos y predicados en el sistema tienen una hora asociada. A las creencias, no creencias y compromisos se les supone persistentes por omisión: mientras no se indique lo contrario, siguen siendo válidos.
- **Creencia**: Es un operador modal, etiquetado con una proposición y un tiempo: el agente cree ciertas cosas en determinados momentos; posteriormente, sus creencias pueden o no cambiar. Su forma es (B, [t, mi_amigo(fulano)]).
- **Compromiso**: En un momento t, el agente se encuentra comprometido con otro respecto a alguna proposición f; esto es, (CMT, [t, otro-agente, f]). Por ejemplo, para levantar un brazo, un robot se comprometería a la proposición "el robot levanta su brazo". Los compromisos se consideran aún más básicos que las decisiones, ya que éstas últimas pueden modelarse como compromisos del agente consigo mismo: (CMT, [t, self, f]).
- **Potencialidad**: Expresa el hecho de que un agente puede hacer algo en un momento específico: (CAN, [t, a, f]). La **capacidad** es la versión inmediata de la potencialidad: (ABLE, [a, f])

Los agentes escritos en AGENT0 (el lenguaje de POA) tienen un ciclo durante el cual reciben mensajes, los aceptan de acuerdo a las reglas programadas, modifican su estado mental, y realizan acciones de acuerdo con este nuevo estado (ver figura 1.5).

---

[2] Los actos de habla se tratan con mayor detalle en el capítulo 2.

[3] De hecho, Shoham supone que diferentes aplicaciones de POA requerirán diferentes modelos de estado mental del agente.

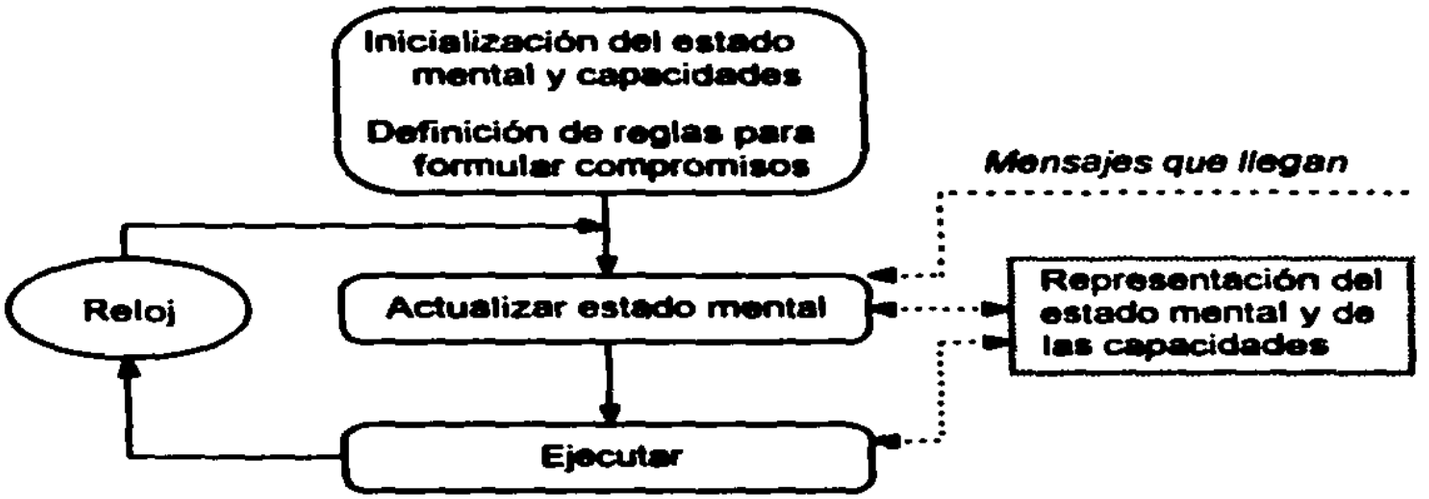

Figura 1.5.- Un intérprete genérico de agentes

Las acciones pueden clasificarse como privadas o comunicativas, y cada una de ellas puede a su vez ser condicional o incondicional.

- Las acciones privadas dependen del agente que las contiene, y sus efectos pueden ser o no observados por otros agentes. Por tanto, son semejantes a la implementación de los *métodos* de un objeto tradicional. Tienen asociado el tiempo de su realización, y su sintaxis es DO(t, acción-privada).
  En contraste, las acciones comunicativas son uniformes y comunes a todos los agentes, y siempre involucran operaciones de entrada y salida. AGENT0 define tres tipos de acción comunicativa:
  INFORM(t, otro-agente, hecho)
  REQUEST(t, otro-agente, acción)
  UNREQUEST(t, otro-agente, acción)
  Las peticiones pueden anidarse arbitrariamente, por lo que es válido escribir
  REQUEST(1, a, REQUEST(5, b, INFORM(10, c, hecho))).
- Las acciones incondicionales se realizan siempre que se les encuentra. Son DO, INFORM, REQUEST, UNREQUEST, y REFRAIN, que impide que se realice una acción.
- Las acciones condicionales tienen asociada una condición mental, que se revisa antes de ejecutar la acción. Estas condiciones mentales son combinaciones lógicas de dos patrones mentales: creencias ((B, hecho)) y compromisos con otros agentes (((CMT, agente), acción)). La sintaxis de una acción condicional es
  IF condición-mental THEN acción

por lo que es posible escribir:

```
IF (B, [t, empleado(fulano, acme)]) THEN
        INFORM(u, h, [t, empleado(fulano, acme)])
```

que se lee como "si en el tiempo u se cree que en el tiempo t fulano es empleado de acme, hay que informarlo a h."

- Las reglas de compromiso se emplean para determinar cuándo un agente adquiere nuevos compromisos con otro. Contienen tanto condiciones mentales como patrones de mensajes, los cuales permiten clasificar mensajes de acuerdo a su origen, tipo de acción comunicativa o contenido. Su sintaxis es

  ```
  COMMIT(condición en mensaje, condición mental, agente, acción)
  ```

  Una regla simple de compromiso es:

  ```
  COMMIT(   (?agente, REQUEST, ?acción),
            (B, [?t, mi_amigo(?agente)]),
            ?agente,
            ?acción )
  ```

  Esto es: cuando se reciba una petición de un agente a quien se considere como amigo, hay que comprometerse con él a realizarla.

La programación en AGENT0 contiene todos los elementos básicos de la programación cooperativa basada en intenciones, pero es relativamente de bajo nivel: los estados de un agente tienen que describirse en base únicamente a las condiciones mentales; no hay una estructuración que aproveche la secuencialidad de varias de las actividades que se realizan[4], y los agentes no pueden "aprender" ni reaccionar a situaciones no previstas por el programador (no hay más reglas que las que se definen al principio del programa).

1.2.2 Sociedades de agentes

Hewitt define el concepto de **sociedad de información** como un sistema con componentes humanos y telecomputacionales, en el que la producción de artefactos (internos y externos) se realiza a través de especificaciones digitales [Hewitt e Inman, 1991]. Estas sociedades de información son un caso de los llamados Sistemas Abiertos de Información, que manejan altos volúmenes de información heterogénea y utilizan una concurrencia masiva. En estos sistemas, Hewitt e Inman notan que la mera inteligencia no es suficiente y que debe considerarse especialmente la robustez, administrabilidad y

---

[4] Por ejemplo, en un sistema de reservaciones de viajes, de "revisar ocupación" se pasa a "asignar asientos". Bajo una programación tradicional, esto se expresa directamente a través de la secuencialidad, pero en POA se tienen que emplear estados mentales diferentes, buscando que "asignar asientos" quede habilitado después de que se haya ejecutado lo correspondiente a "revisar ocupación".

escalabilidad del sistema. Como el trabajo en estos sistemas se realiza a gran escala, requiere organización: debe dividirse entre varios recursos, distribuidos en el tiempo y el espacio; de aquí también se sigue que la negociación es una actividad fundamental para estos sistemas.

Con ese fin, se introduce el concepto de **ORG** (Organización de Generalidad Restringida) como una extensión de la arquitectura de actores. Una ORG se basa en el principio de agregación: puede integrarse por actores y otras ORGs, y soporta la organización a través de las siguientes facilidades:

- Facilidad de operaciones, que provee recursos tales como procesadores, almacenamiento y comunicación a las tareas autorizadas.
- Facilidad de Reportes, que provee información a otras facilidades respecto a qué ha pasado, cuándo sucedió, y qué participantes se vieron involucrados.
- Facilidad de Membresía, que supervisa durante su ejecución a la población de la ORG, y crea y finaliza membresías conforme evoluciona.
- Facilidad de Relaciones, que controla las comunicaciones fuera de los límites de la ORG. Esta facilidad incluye también a Recepción, que se encarga de los mensajes que van dirigidos a la ORG como un todo.
- Facilidad de Administración, que controla la conducta de la ORG. Determina las políticas y procedimientos de la conducta de la ORG.

Estas facilidades se integran por uno o varios actores. Asumen así diferentes responsabilidades y se comunican de manera uniforme.

1.2.3 Facilitadores

Los **facilitadores** desarrollados en LANIA [Lemaître *et al*, 1993] son una propuesta de un sistema multiagente cooperativo y abierto, en el cual se asocia un ente experto en cooperación (el facilitador de la cooperación) a sistemas preexistentes o nuevos. La arquitectura se basa en una metáfora del trabajo en oficinas: un facilitador funciona como "secretaria" del sistema o usuario que tiene asociado, buscando con los demás facilitadores la información que su "jefe" necesite para realizar su trabajo, y pasándole las peticiones de servicios que hagan otros facilitadores. De esta manera se busca integrar sistemas preexistentes a nuevas aplicaciones (reutilizando su funcionalidad), y simplificar el diseño de nuevos sistemas multiagentes, modularizando el conocimiento social.

Un facilitador contiene principalmente:

- El control de la cooperación [Zamora, 1996]. El facilitador contiene una biblioteca de planes precompilados, que se representan mediante redes de transición aumentada (ATNs).
- El control de diálogos y la comunicación, que es el objetivo de este trabajo.
- Servicios de directorio, asociación y transferencia, que utilizan tanto los servicios del control de diálogos y otros protocolos disponibles para implantar el ambiente

normal de un sistema distribuido: obtención de las direcciones de los agentes a partir de sus nombres y capacidades, manejo de grupos de facilitadores, y otros servicios de transferencia de datos (correo electrónico, FTP para transferencia de archivos, etc.). Estos servicios forman la interfaz de red.
- La interfaz a su usuario, ya sea humano o sistema computacional.

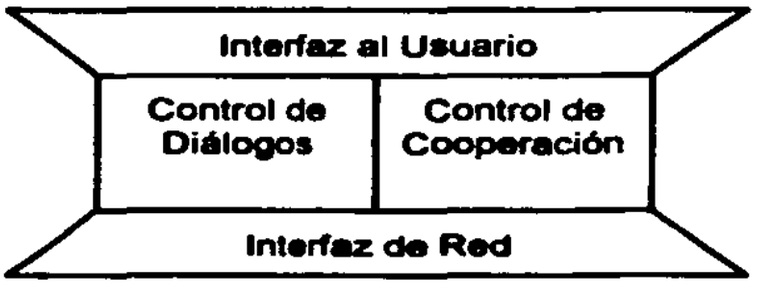

Figura 1.6.- Estructura de un facilitador

Con el énfasis en la comunicación, esos bloques se presentan de la siguiente forma: hay una **máquina de acciones**, que es la encargada de recibir los mensajes provenientes de la red y rutearlos hacia los componentes internos del facilitador, que corresponden principalmente a las categorías de interfaz al usuario, planeación o control del agente, y módulo de acciones privadas. La comunicación entre estos componentes se desarrolla a través del sistema de control de diálogos.

Los facilitadores pueden especializarse en dos tipos principales: ejecutivos y de staff [Lemaître y Excelente, 1997]. Los facilitadores de staff ofrecen servicios simples, encapsulando sistemas como bases de datos para su utilización dentro de la sociedad; corresponden a los servidores en los sistemas tradicionales. Los facilitadores ejecutivos tienen mayor conocimiento social.

Actualmente el trabajo sobre los facilitadores se ha dirigido a los campos de planeación y aprendizaje [Excelente, 1997].

1.3 Comentarios

¿Qué caracteriza a un ambiente de sistemas abiertos, como los considerados en este capítulo? Por lo visto en las secciones anteriores, destacan los siguientes puntos:

- Un medio ambiente compartido (dentro de una máquina o en una red) donde las aplicaciones coexisten y se comunican.
- La necesidad de contactar a las otras aplicaciones disponibles y comunicarse con ellas.

- Un gran dinamismo. Puede haber aplicaciones que se ejecutan todo el tiempo, mientras otras se activan y desactivan a intervalos irregulares. Puede haber varias versiones de una misma aplicación coexistiendo en cualquier momento.
- Diversidad de formatos de datos empleados por las aplicaciones.
- Heterogeneidad.

En base a esas características, puede verse de qué manera los modelos de comunicación y programación vistos se pueden aplicar en este ambiente.

RPC permite la invocación de servicios entre aplicaciones, aún cuando éstas se encuentren distribuidas en una red heterogénea. Su modelo de programación es muy familiar, permite describir a detalle la estructura de los datos a intercambiar, y si se hacen disponibles las descripciones en RPCL, pueden desarrollarse nuevos clientes fácilmente. Proporciona un servicio básico de localización de servicios, pero no existe información disponible en tiempo de ejecución que permita descubrir o utilizar fácilmente servicios nuevos (no hay posibilidad de averiguar ni el tipo del resultado ni el tipo de parámetros requeridos).

DDE ejemplifica las ventajas que se pueden obtener al trabajar con protocolos establecidos; por ejemplo, las ligas calientes y tibias estandarizan la adquisición interactiva de datos entre aplicaciones. Pero presenta varios inconvenientes y limitantes que han hecho que se prefiera a OLE2 como estándar para la comunicación de aplicaciones en Windows. Por ejemplo:

- Si alguno de los comandos invocados en el servidor genera uno o varios datos como resultado, WM_DDE_EXECUTE (que sólo informa del éxito o fallo de la ejecución) no es apropiado: se requiere programar un protocolo especial entre cliente y servidor, por ejemplo a través de WM_DDE_REQUEST.
- Los clientes requieren mucho conocimiento previo respecto a los servidores que contactan: necesitan saber qué temas y artículos manejan, cómo pueden aprovecharlos, y si además quieren darles comandos a ejecutar, deben conocer su sintaxis. En este respecto, DDE sugiere que los servidores soporten un "tema del sistema" (SZDDESYS_TOPIC) donde se describan las características del servidor, pero además de que no todas las aplicaciones lo incluyen, su contenido resulta ser insuficiente[5]. No hay una estandarización que permita el

[5] SZDDESYS_TOPIC debe contener al menos tres artículos: SysItems, Topics y Formats, que enumeran respectivamente: todos los artículos del tema del sistema, los temas conocidos, y los formatos de datos aceptados por el servidor.

descubrimiento y aprovechamiento de las capacidades de otras aplicaciones en tiempo de ejecución: en la práctica, los clientes se programan para comunicarse directamente con un servidor específico, o el usuario del programa cliente indica qué acciones tomar.

- Al momento de intercambiar datos, el protocolo permite a un cliente indicar su preferencia por un formato específico. Pero eso no garantiza que el servidor lo soporte, y al no haber una manera general de averiguar los formatos que soporta cada servidor, el problema debe resolverse mediante una negociación caso por caso, para lo cual no hay ningún soporte preestablecido.
- DDEML administra una parte importante de la complejidad de la comunicación entre las aplicaciones, pero el peso de la programación cae sobre las funciones *callback*, que no proveen una visión estructurada ni jerárquica del tipo de comunicaciones que se realizan.

Los sistemas de objetos distribuidos, por su parte, ofrecen servicios útiles y proporcionan una plataforma muy adecuada para el desarrollo de sistemas abiertos. El almacenamiento de información sobre los tipos de objetos presentes en el sistema permite el descubrimiento e invocación de servicios nuevos (como en *scripting* y *automation* de OLE2).

En el fondo, la programación en los sistemas de objetos distribuidos se basa en el descubrimiento y utilización de las interfaces que implementan los objetos, por lo que las interacciones entre componentes quedan determinadas por las interfaces que conocen e implementan. Por ejemplo, en OLE2 los objetos contenedores y contenidos deben implementar las interfaces descritas en la figura 1.7; para trabajar conjuntamente, sólo necesitan saber que el otro tipo de objetos soporta las interfaces correspondientes. Por tanto, las interacciones entre objetos arbitrarios quedan restringidas a las interfaces establecidas por los arquitectos de OLE.

En contraposición, el modelo de agentes ofrece una comunicación no restringida y de alto nivel, aunque puede impactar en la heterogeneidad del ambiente, porque requiere algún tipo de soporte para el manejo de los estados mentales y para el procesamiento de los diferentes contenidos de mensajes que podrían necesitarse. Shoham indica que el modelo sólo supone que los aspectos mentales a considerar son mutuamente consistentes, pero para el ambiente heterogéneo es necesario establecer distintos niveles de participación, en base a los recursos disponibles a cada agente. Un mecanismo de comunicación que facilite la estructuración y el procesamiento de los mensajes puede ayudar en ese sentido, por lo cual se revisará más de cerca las bases del modelo de comunicación que emplean los agentes de Shoham.

Tipo	Interfaces	Propósito
Contenedor	IOleClientSite	Ofrecer servicios e informar sobre el área de despliegue que proporciona el contenedor
	IAdviseSink	Recibir notificaciones de los objetos contenidos sobre cambios en despliegue y datos.
Objeto contenido (servidor)	IOleObject	Informar sobre el objeto contenido e invocar operaciones.
	IDataObject	Transferir datos, notificar de cambios, e informar sobre los formatos y medios de intercambio
	IPersistStorage	Guardar y recuperar al objeto contenido, a través del medio de almacenamiento que da el contenedor
	IViewObject	Alternativa a IDataObject para que el objeto se despliegue, y establecer una conexión opcional para notificar de cambios en la presentación del objeto.

Figura 1.7.- Interfaces OLE2 para interacción entre objetos contenedores y contenidos

Capítulo 2

Actos de habla y procesamiento del discurso

...Su espíritu es el de los baratijeros Mosche y Daniel que en mitad de la gran llanura de Rusia se saludaron.

– ¿A dónde vas, Daniel? –dijo el uno.
– A Sebastopol –dijo el otro.

Entonces, Mosche lo miró fijo y dictaminó:

– Mientes, Daniel. Me respondes que vas a Sebastopol para que yo piense que vas a Nijni-Novgórod, pero lo cierto es que vas realmente a Sebastopol. ¡Mientes, Daniel!

J.L. Borges. *El Truco*

La cooperación, como es entendida en este trabajo, requiere para realizarse de una comunicación que haga evidentes las intenciones de los hablantes (ya que en el ambiente de red heterogénea que suponemos no es factible que cada agente pueda estar pendiente de las acciones de los demás para deducir sus intenciones, como en [Genesereth *et al*, 1988]). La comunicación más expresiva posible es la que se da entre seres humanos, así que dentro del proceso de análisis del problema se presentan algunos trabajos desarrollados en las áreas de filosofía del lenguaje y procesamiento del lenguaje natural.

¿Qué hay detrás del fenómeno de la comunicación? ¿Cómo empleamos el lenguaje, y cómo lo reconocemos e interpretamos? ¿Cómo es posible que cuando alguien emite[1] bajo las circunstancias apropiadas una serie de sonidos como "mañana paso por tí", esté en realidad haciendo una promesa, y no otra cosa? ¿Cómo reconoce el escucha de esa frase que esa emisión realmente equivale a hacer una promesa, y no a otra acción como la petición de un favor?

Estas preguntas y muchas otras relacionadas con el significado de las oraciones han dado amplio material de estudio a los filósofos de todas las épocas. Sus primeras respuestas buscaron asociar a las oraciones un valor en base a la verdad o falsedad de su

---

[1] Cabe señalar que en el caso general, los símbolos pueden ser tanto sonidos como palabras escritas. Emisión, pronunciación, y escritura son equivalentes entre sí; recepción, escucha y lectura también. El proceso de reconocimiento es análogo.

contenido; una frase como "el granizo es frío" sería cierta o falsa dependiendo de si el granizo es realmente frío o no. Si bien este tipo de enfoque concuerda con el ideal aristotélico de la filosofía como búsqueda de la verdad, resulta demasiado restrictivo para el estudio de una gran variedad de oraciones que usamos cotidianamente: ¿cómo decidir si emisiones como "compra el periódico" son verdaderas o no? Lo más que puede decirse en esos casos es si son o no sinceras.

Ante esos inconvenientes, se empezaron a buscar alternativas pragmáticas: ¿cómo es que usamos el lenguaje, y qué cosas son básicas para la comunicación exitosa? ¿Cómo se desarrolla la comunicación? La respuesta a estas preguntas incluye las nociones de lenguaje-acción, conocimiento compartido, actos de habla, y estructura del discurso, cuya exposición en este capítulo se complementa con la de ACL (*Agent Communication Language*), un lenguaje para la comunicación entre agentes inteligentes basado en actos de habla.

2.1 Conocimiento compartido y discurso

Según Webster, comunicación es "un proceso mediante el cual se intercambia información... a través de un sistema común de símbolos y signos". Dicho intercambio se realiza a través de mensajes, los cuales constan de contenido y contexto. El contenido son los datos codificados en los mensajes, y contexto es aquello que transforma los datos en información: como tal, es suministrado directamente por el remitente del mensaje (antecedido por una frase aclaratoria como "esto es una petición"), por el receptor (como función del estado en que éste se encuentre), o por una mezcla de ambos.

Podemos empezar nuestro estudio a partir de los resultados obtenidos por el lingüista Noam Chomsky y el filósofo John Searle. Chomsky afirma que el lenguaje es parte de nuestra propia naturaleza biológica: la capacidad de utilizarlo y comprenderlo no la aprendemos, sino que es parte de nuestro patrimonio genético; dicho de otra manera, los niños no aprenden a hablar porque ya saben hacerlo. Correspondientemente, John Searle adopta la llamada perspectiva del **lenguaje-acción**: considera que hablar un lenguaje es hacer cosas, tomando parte en una conducta altamente compleja pero gobernada por reglas comunes a un grupo social, así que hablar un lenguaje es exhibir el dominio que se tiene de dichas reglas.

De las observaciones anteriores se sigue que, para comunicarnos, recurrimos a elementos que consideramos presentes en nuestros interlocutores, principalmente conocimiento de diversos tipos. Esto puede corroborarse al identificar algunas propiedades claves del lenguaje [Rich y Knight, 1991]:

- A través de un conjunto de símbolos, finito y por tanto aprehensible, podemos hablar acerca de un mundo infinito. A cambio de esta posibilidad, hay que aceptar que una frase pueda adquirir significados distintos cuando se le usa en contextos

diferentes. Para distinguir entre los significados posibles, los participantes requieren compartir conocimientos.

- Las oraciones son descripciones incompletas de la información que transmiten, ya que al hablar el emisor puede ser tan preciso o vago como desee, y puede suponer que sus oyentes o receptores saben cosas en relación al tema tratado.
- Un lenguaje es sumamente dinámico, pues se le pueden incorporar palabras nuevas; evoluciona conforme a nuestras experiencias, y es relativamente fácil extenderlo para cubrirlas, pues los participantes en una conversación poseen la capacidad de reconocer y explicar el uso de los nuevos términos a través de ciertas pistas. Consideremos la palabra "tesobono" en ámbitos financieros y económicos; aunque nunca la hubiéramos escuchado, en una frase como "el pago al vencimiento de los tesobonos", su posición sintáctica como sustantivo y su empleo junto con otros elementos comunes al ámbito (como "pago al vencimiento"), nos permiten hacer una identificación suficiente para comprender el sentido de la oración y obtener información sobre la palabra.
- Hay muchas maneras de decir la misma cosa; por ejemplo, decir "cumplo 32 años el próximo 2 de agosto" equivale a afirmar "nací un día 2 de agosto". Un participante en una conversación es capaz de rescatar un conocimiento a partir de otro cuando lo requiere, ya que sabe muchas cosas.

Como se ve, el conocimiento común abarca tanto al tema que tratan los interlocutores como a las reglas del lenguaje que utilizan, y es vital para la realización de la comunicación[2]. Queda por ver cómo se le usa: ¿qué tiene que hacer un escucha para reconocer los mensajes que recibe, y cuáles son los procesos que lleva a cabo? En el área de procesamiento de lenguaje natural se distinguen [Allen, 1987]:

- Análisis morfológico. Análisis de los componentes de las palabras individuales, y distinción de elementos tales como símbolos de puntuación y diferencias en tonos de pronunciación y énfasis.
- Análisis sintáctico. Reconocimiento de las estructuras que muestran cómo se relacionan las palabras entre sí.

---

[2] Esta dependencia del conocimiento común puede dar lugar al fenómeno conocido como transposición del texto: dos lectores o receptores, con experiencias y conocimientos diferentes, realizan interpretaciones distintas del mismo texto. J.L. Borges aprovecha esta posibilidad en *Pierre Mènard: Autor del Quijote*, donde el personaje del título, a pesar de reescribir el Quijote con las mismas oraciones, le da un sentido diferente pues sus lectores son de conocimientos, experiencias, y siglos distintos a los de Cervantes.

- Análisis semántico. Mapeo entre las estructuras sintácticas y los objetos presentes en el mundo del que se habla.
- Integración o estructura del discurso. Establece el significado de una oración en términos de las oraciones que la preceden, y reconoce cómo puede afectar a la interpretación de las que la siguen. Por ejemplo, en una oración como "Juan lo necesita", "lo" hace referencia a algo mencionado anteriormente, mientras que el uso del pronombre "él" en una oración posterior puede referirse a "Juan".
- Análisis pragmático. Reinterpreta la estructura semántica para identificar qué es lo que realmente se quería decir. Por ejemplo, dependiendo del contexto en que se emplee, una frase como "se está haciendo tarde" puede ser más una forma de apurar a alguien que una mera observación sobre la hora.

Cada uno de estos procesos aporta información que conduce al reconocimiento o rechazo de las emisiones que aparecen en el discurso, y se realizan simultáneamente porque la información generada por uno es aprovechable por los demás. La mayor parte de las características de alto nivel del lenguaje se encuentran en el análisis semántico, análisis pragmático e integración del discurso. Como el análisis semántico es un proceso que depende claramente del contexto de aplicación (el tema del que se habla), para el modelado de los diálogos nos ocuparemos solamente de los actos de habla y la estructura del discurso.

2.2 Actos de Habla

Austin [1975] desarrolló la Teoría de Actos de Habla como un enfoque pragmático al problema de la semántica, dando atención especial al papel de las intenciones en la comunicación. Al hablar en términos de condiciones mentales que provocan acciones, se tiene un marco apropiado para contener la perspectiva del lenguaje-acción (hablar es participar en una conducta gobernada por reglas).

Cuando un ente A envía un mensaje M expresado en un lenguaje L a otro ente B, Austin distingue los siguientes actos realizados al hablar (o actos de habla):

- **Actos locutivos.** El conjunto de vocablos de L que utiliza A al emitir M.
- **Actos ilocutivos.** Consisten en la intención que tiene A al comunicar M a B: informar, convencer, preguntar, solicitar, etc. Dicha intención recibe el nombre de *fuerza ilocutiva* de M. Propiamente, éstos son los verdaderos actos de habla.
- **Actos perlocutivos.** Son los efectos que A causa en B tras que éste reconoce el acto ilocutivo asociado a M. Por ejemplo, el efecto perlocutivo de un acto ilocutivo de convencimiento produce creencia.

Esencialmente, con esta clasificación Austin plantea distinguir entre "decir algo", "hacer cosas diciendo algo", y los efectos que esto causa en el escucha. Bajo este

esquema, la comunicación entre hablante y escucha es exitosa si el destinatario del mensaje reconoce la fuerza ilocutiva del mensaje y si los efectos perlocutivos corresponden al acto ilocutivo del mensaje. Si *A* envía un mensaje a *B* para convencerlo de algo, *B* puede reconocer la fuerza ilocutiva de convencimiento pero no aceptar el contenido del mensaje porque choca con sus creencias, porque duda de la calidad moral de *A*, o por otros motivos. Si no se alcanza el efecto buscado por *A*, pueden requerirse más intercambios de mensajes, hasta que *A* consiga su objetivo o desista.

2.2.1 Clasificación de Searle

John Searle [1969] no concuerda completamente con la clasificación de Austin: aunque los actos perlocucionarios siguen siendo importantes en cuanto a que expresan lo que ocurre con el escucha, él no encuentra una distinción importante entre actos locucionarios e ilocucionarios, pues considera que los primeros caen dentro del acto de habla general "decir"; así pues, la unidad básica y fundamental del lenguaje no es la palabra, ni la oración: es el acto de habla, lo que se busca al emitir una serie de palabras. Searle considera entonces que al hablar se hacen tres actos fundamentales[3]:

- Emitir palabras (morfemas, oraciones). Esto es, realizar actos de emisión.
- Referir sujetos y objetos, identificándolos, y predicar cosas acerca de ellos. O sea, realizar actos proposicionales.
- Enunciar, preguntar, mandar, prometer, etcétera. En otras palabras, actos ilocucionarios.

Estos actos no existen independientemente unos de otros, sino que se relacionan. En particular, los actos de emisión y proposicionales no son sólo medios para el fin "realizar actos ilocucionarios", sino que más bien son como reglas constitutivas: además de la fuerza ilocutiva, un acto ilocucionario es emisión, prédica y referenciación, pero es provechoso distinguirlos para su estudio.

De cualquier forma, la enunciación exitosa depende del acatamiento de las llamadas reglas sociales, que se expresan en términos de condiciones mentales.

2.2.2 Actos de habla como operadores

Para valorar los efectos de la comunicación, un agente requiere un modelado mental de sus escuchas. Formalmente, hay varias formas de realizar este modelado [Wooldridge y Jennings, 1994], pero Cohen y Perrault [1988] proponen una especialmente atractiva: ver

---

[3] Esta clasificación proporciona algunos argumentos para la distinción entre lenguaje del mensaje y lenguaje del contenido del mensaje.

a las acciones comunicativas como operadores dentro de un plan, lo que permite tratarlas junto con las acciones normales del agente.

Con ese fin, se da una semántica a los actos de habla en términos de las precondiciones y efectos de su ejecución, mostrando cómo afectan las creencias de los agentes que se comunican. El modelo de creencias que proponen Cohen y Perrault parte de los predicados **KNOW**, **BELIEVE** y **WANT**, empleándolos para distinguir entre conocimiento, creencia y metas; por definición, esos predicados satisfacen una serie de axiomas (ver figura 2.1).

Dados dos agentes A1 y A2, A1 debe manejar:
1. *Creencias* (**BELIEVE**). A1 debe distinguir entre las siguientes creencias:
   - A2 **BELIEVE** (*el camión sale del andén 65*)
   - A2 **BELIEVE** (*el camión tiene un andén de salida*)
   - A2 **KNOW** (*de qué andén sale el camión*)
2. *Deseos/Metas* (**WANT**). A1 debe diferenciar entre
   - Las creencias de A2
   - Sus propias creencias y metas
   - Modelo de creencias y metas de A2 para otros agentes
   - Distintos cuantificadores (*A2 quiere tomar un tren* contra *Hay un tren que A2 quiere tomar*)

Figura 2.1.- Creencias y metas

Una vez definidos esos predicados se pueden expresar diversas situaciones. Por ejemplo, A1 **WANT** (A2 **BELIEVE** P) ("A1 quiere que A2 crea P") es la razón que induce a A1 a comunicar P a A2; y A1 **BELIEVE** (A1 **WANT** P) expresa a P como meta de A1.

Para definir los actos de habla en base a estos predicados se reconocen dos tipos de precondiciones: **CAN_DO** para señalar las condiciones necesarias para efectuar el acto de habla, y **WANT** como condición de *sinceridad* (el agente que realiza el acto de habla realmente quiere obtener el acto perlocucionario asociado; o sea, no está jugando). Se puede ejemplificar esto a través de la definición de un operador tipo **REQUEST** (ver figura 2.2). Si alguna de las precondiciones no se cumple al momento de ejecutar el operador, un sistema ejecutor de planes puede introducirla como su siguiente meta a realizar.

En realidad, no basta con que el escucha crea que el hablante quiere que se realice lo que está pidiendo. Debe también obtenerse el efecto perlocucionario (el escucha debe estar convencido), lo que se puede modelar a través del operador **CAUSE_TO_WANT**, que busca obtenerlo (ver figura 2.3).

REQUEST(Hablante, Escucha, X)
Precondición CAN_DO:
El hablante debe creer que el escucha puede realizar X:
Hablante **BELIEVE** (Escucha **CAN_DO** X)
El hablante debe creer que el escucha sabe que puede hacer X:
Hablante **BELIEVE** (Escucha **BELIEVE** Escucha **CAN_DO** X)
Precondición WANT:
La condición de sinceridad por parte del hablante:
Hablante **BELIEVE** Hablante **WANT** instancia-**REQUEST**
Efecto:
El escucha cree que el hablante quiere que X se realice:
Escucha **BELIEVE** (Hablante **BELIEVE** Hablante **WANT** X)

Figura 2.2.- Acto de habla REQUEST

CAUSE_TO_WANT(Agente1, Agente2, X)
Precondición CAN_DO:
Agente2 debe estar enterado de lo que quiere Agente1:
Agente2 **BELIEVE** (Agente1 **BELIEVE** Agente1 **WANT** X)

Efecto:
Agente2 toma a X dentro de sus metas:
Agente2 **BELIEVE** (Agente2 **WANT** X)

Figura 2.3.- Operador CAUSE_TO_WANT

Este mismo tipo de razonamientos y definiciones puede aplicarse a los demás tipos de actos de habla. Por ejemplo, un **INFORM** puede expresarse como se muestra en la figura 2.4. En este caso, la precondición **CAN_DO** puede resultar demasiado restrictiva: el agente que va a comunicar una proposición debe estar convencido de su verdad. Esto es aparente cuando se tiene un agente *A* que quiere que un agente *K* funcione como intermediario y comunique *P* a otro agente *B*; con este esquema, *K* también debe ser convencido de la verdad de *P*. Nuevamente, la definición de estos operadores depende del tipo de cooperación y comunicación deseado entre los agentes.

```
INFORM( Hablante, Escucha, P )
   Precondición CAN_DO:
      Hablante BELIEVE P
   Precondición WANT:
      Hablante BELIEVE Hab WANT instancia-INFORM
   Efecto:
      Escucha BELIEVE (Hablante BELIEVE P)
```

Figura 2.4.- Acto de habla INFORM

2.2.3 Agent Communication Language

Dentro de la Agencia de Proyectos de Investigación Avanzada de la Defensa (DARPA) de EE.UU. se ha realizado el proyecto *Knowledge Sharing Effort* (KSE), con el objetivo de desarrollar estándares para el almacenamiento y compartición de conocimiento entre sistemas de diversa índole. Con este fin se diseñaron dos lenguajes independientes: *Knowledge Interchange Format* (KIF), para la descripción del conocimiento; y *Knowledge Query and Manipulation Language* (KQML), que describe un formato de mensajes y protocolos para la comunicación entre sistemas computacionales no necesariamente basados en conocimiento. KQML y KIF pueden complementarse formando el *Agent Communication Language* [Genesereth, 1994].

Aunque la separación entre lenguaje de contenido y lenguaje de mensajes puede introducir problemas[4], el diseño resultante facilita la estructuración de un agente por capas y permite la heterogeneidad entre los agentes comunicantes: no se necesita que un agente contenga un intérprete de KIF para que realice trabajo útil en la red, pues KQML le proporciona información suficiente como para hacer algo útil con el mensaje.

Los mensajes de KQML son cadenas ASCII con una sintaxis estandarizada, en la que se distinguen la fuerza ilocutiva (llamada *realizativa*) y una serie de parámetros que identifican características tales como el destinatario, remitente, contenido del mensaje y el lenguaje en que se expresa el contenido (ver figura 2.5).

---

[4] Por lo menos hay problemas de redundancia, si los parámetros del mensaje se expresan en el lenguaje del contenido. Otro problema, señalado en [Cohen; Levesque], es que si el lenguaje de contenido contiene operadores de actitud (BELIEVE, GOAL, INTEND, etc.), pueden generarse mensajes totalmente contradictorios.

```
(evaluate
   :language KIF
   :ontology motores
   :content (val (torque motor1) (tiempo-simulación 5))
   :receiver simulador
   :sender interfaz-gráfica
   :reply-with idquery32 )
```

Figura 2.5.- Mensaje KQML

El estándar reserva algunos parámetros y realizativas para que su interpretación sea la misma en todas las implementaciones[5]; dependiendo de la realizativa, algunos parámetros son obligatorios y otros opcionales. Algunos de los parámetros más comunes son:

- `:sender`, `:receiver` y `:content`, que proporcionan la información mínima que se espera ver en un mensaje. El `:content` del mensaje puede ser a su vez otro mensaje KQML, lo que se usa para expresar funciones tales como ruteo de mensajes, disponibilidad a responder a los mensajes que correspondan al contenido, etc.
- `:reply-with`, que da un identificador que el receptor del mensaje debería emplear en un mensaje de respuesta dentro de un parámetro `:in-reply-to`. Esto permite identificar secuencias de conversación.
- `:language`, para especificar el lenguaje (KIF, KQML, Prolog, etc.) en el que está descrito el contenido proposicional del mensaje.
- `:ontology`, para indicar el dominio sobre el que debe realizarse el mapeo semántico del contenido proposicional (motores, béisbol de la liga mexicana, etc.)

Las realizativas reservadas por el estándar se refieren tanto a la comunicación punto a punto (como `tell`, `reply`, `ask_if`, y `ask_one`), como a otras interacciones dirigidas a los facilitadores KQML, que son agentes que hacen ruteo del contenido de mensajes, intermediación de información, reclutamiento de proveedores, y servicios parecidos (ver figuras 2.6, 2.7 y 2.8).

Por tratarse de una especificación que deja a los programadores decidir libremente cuáles realizativas van a utilizar y cuáles van a añadir, no existe un software KQML sino implementaciones, varias de las cuales sólo manejan el nivel comunicativo y no llegan al nivel de mantenimiento de estados mentales. KATS (desarrollada por Loral y la University

---

[5] Un programador puede definir nuevas realizativas y parámetros, pero como KQML no contempla mecanismos que permitan a un agente reconocer el tratamiento que debe dar a las nuevas realizativas (como una jerarquía o clasificación de las mismas), la consistencia de la interpretación es responsabilidad del programador.

of Maryland Baltimore County) es una de ellas, que impone un modelo de procesamiento de mensajes mediante *callbacks* [Finin *et al*, 1994]. KAPI es otra, desarrollada por Lockheed, Enterprise Integration Technologies, y el SIMA (*Stanford Integrated Manufacturing Association*) y CDR (*Center for Design Research*) de la Universidad de Stanford [Weber y Kuokka, 1995].

KAPI (*KQML Application Programmer's Interface*) es un software desarrollado para el proyecto SHARE (también financiado por DARPA). La meta final de SHARE es desarrollar metodologías y ambientes para el desarrollo colaborativo de productos, ya que ésta es una actividad que involucra cada vez a más equipos de personas de varias organizaciones, que trabajan juntas a través de las redes de computadoras con el soporte de los servicios de información y cómputo. SHARE pretende proveer la tecnología posibilitadora que apoyará a los ingenieros de diseño al permitirles el acceso a información útil a través de la red, lo que normalmente no está disponible a un ambiente de un solo usuario.

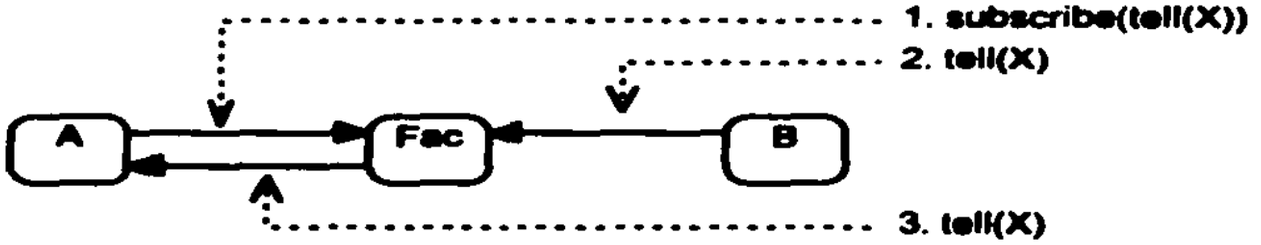

Figura 2.6.- Ruteo por contenido: protocolo *subscribe*

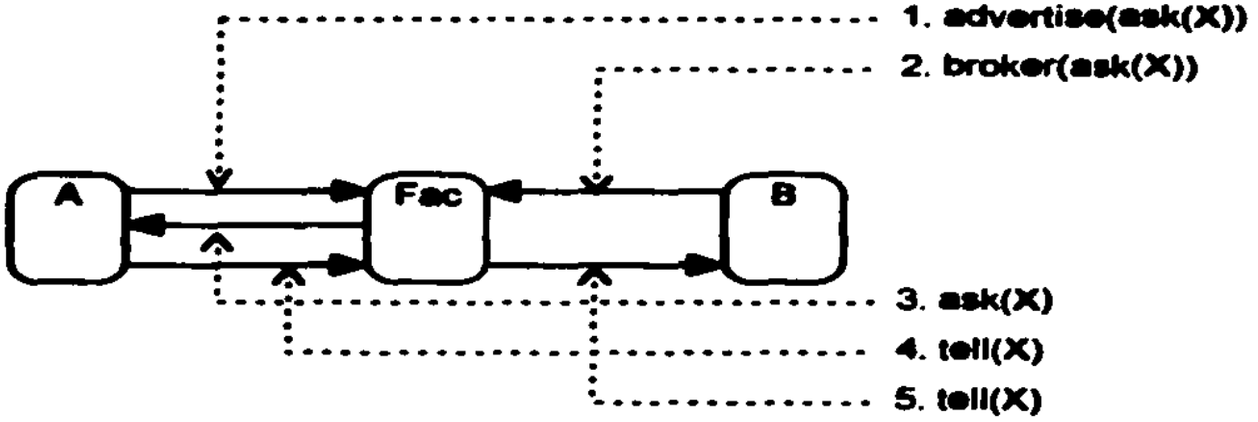

Figura 2.7.- Intermediación: protocolo *advertise*

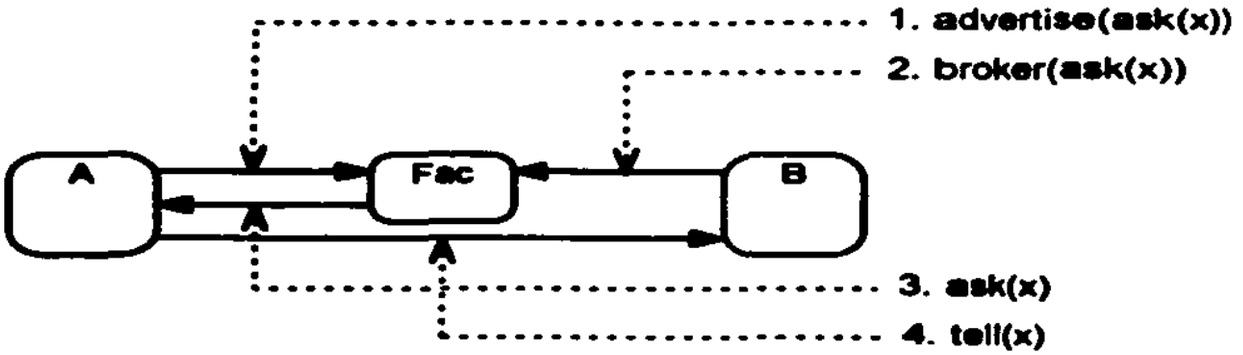

Figura 2.8.- Reclutamiento: protocolo *advertise*

KAPI está programado en C y su característica más importante es el soporte para el envío y recepción de mensajes a través de diferentes protocolos (TCP, HTTP, correo electrónico, y MBUS para comunicación *multicast*) de una manera bastante transparente para el programador, quien sólo necesita registrar las direcciones donde el agente espera recibir y enviar mensajes. Las direcciones, formadas con el protocolo de comunicación empleado (TCP, HTTP, etc.) y una serie de parámetros específicos, se expresan a través de los mismos *Uniform Resource Locators* (URLs) del *World Wide Web* y se emplean en los parámetros `:receiver` y `:sender` del mensaje. Para asociar nombres lógicos a las direcciones se emplea un tipo especial de agentes dedicados llamados *Agent Name Servers* (ANS), que proporcionan un servicio de directorio[6].

2.3 Estructura del discurso

Un discurso, como secuencia de intercambios de actos comunicativos, tiene asociado un grupo de participantes en el que se distingue a aquel que inicia el discurso llamándolo participante iniciador (PI). Cada discurso tiene una meta llamada propósito del discurso (PD), que explica por qué se realiza el intercambio de emisiones (en vez de efectuar cualquier otra acción no comunicativa), y porqué el contenido del discurso es el actual y no otro.

Durante un discurso se habla de diferentes cosas, y se tienen que concretar varias submetas antes de alcanzar el PD, como en el caso de un instructor que enseña a un aprendiz a realizar cierta tarea: el discurso de la explicación tiene como submetas el

---

[6] **La implementación actual del directorio de KAPI sólo funciona sobre HTTP, y no usa los protocolos de KQML definidos para este propósito.**

enseñar cada uno de los pasos que llevan a la culminación de la tarea, aclarando en cada uno de ellos las dudas que pudieran surgir.

El problema de integración del discurso (la determinación de qué se dijo y qué significa) radica en identificar la **estructura del discurso**, en la cual Grosz y Sidner [1986] distinguen tres partes fundamentales: qué etapas lo forman (**estructura lingüística**), cuáles son sus objetivos (**estructura de intenciones**), y cuáles son los objetos de los que se habla en cada una de ellas (**estructura de atención**). La información obtenida en el reconocimiento de una parte puede emplearse para precisar otras, ya que se encuentran muy relacionadas (por ejemplo, en las estructuras lingüística y de intenciones, un cambio en una se refleja en la otra). Pero ya que ninguna estructura puede conocerse de antemano al desarrollo del discurso, es necesario reconocerlas dinámicamente.

2.3.1 Estructura lingüística

En el nivel más elemental, las frases generadas por los participantes en el discurso se estructuran en **segmentos de discurso**, que intuitivamente son conjuntos de frases relacionadas por una materia común. Los segmentos se presentan de manera secuencial, pero pueden contener a su vez otros segmentos y actos de habla. Cada segmento tiene asociada una meta llamada **propósito del segmento de discurso** (PSD), que colabora de alguna manera a la satisfacción del PSD del segmento en que está contenido. La **estructura de segmentos** refleja estas relaciones (ver figura 2.9).

Retomando el ejemplo del maestro y su aprendiz, cada paso de instrucción correspondería a un segmento; si en un momento dado el aprendiz expresa una duda u objeta alguna instrucción, se está introduciendo un subsegmento anidado en el segmento correspondiente a la instrucción.

En el área de procesamiento de lenguaje natural, el proceso de **segmentación**, que consiste de la identificación de los límites de segmentos, se realiza principalmente buscando ciertas frases clave ("para empezar...", "de lo dicho anteriormente se sigue que...", "a continuación...", "profundizando en este tema, quiero añadir...", etc.) que suelen marcar los cambios de segmento. Conforme los hablantes hagan más explícitos los límites de segmento se requerirá menos procesamiento.

2.3.2 Estructura de intenciones

Un PI puede tener varias razones privadas para realizar un discurso, pero el PD es aquel que pretende que reconozcan los demás participantes; como tal, resulta fundamental para la comunicación: sin él no puede alcanzarse el efecto perlocutivo buscado al realizar el discurso. No existe una categorización de los PD posibles, pero algunos ejemplos son los siguientes:

- **Pretender que algún agente crea algún hecho.**
- **Pretender que algún agente crea que un hecho apoya a otro.**
- **Pretender que algún agente pretenda realizar alguna tarea física (esto es, que la tome como meta).**
- **Pretender que algún agente conozca alguna propiedad de algún objeto.**

Se ha mencionado que los PSD apoyan la obtención del PD; en general, los propósitos que aparecen durante el discurso mantienen relaciones que definen órdenes parciales entre ellos. Estas relaciones forman la estructura de intenciones, que sigue muy de cerca a la estructura de segmentos (ver figura 2.9). Dichas relaciones son:

- **Dominio:** Se dice que el propósito de un segmento *SegA* domina al de otro *SegB* (denotado como *SegA.PSD DOM SegB.PSD*) si *SegB.PSD* provee parte de la satisfacción de *SegA.PSD*. Esta relación refleja la contención en la estructura de segmentos, donde un subsegmento explica o apoya al segmento en que está contenido.

  De hecho, el propósito del discurso (*PD*) domina al de cualquier segmento.

- **Precedencia de satisfacción:** Se da cuando se requiere satisfacer el propósito de un *SegA* antes de satisfacer al de otro *SegB* (i.e. *SegA.PSD PS SegB.PSD*). Esta relación se presenta típicamente en los discursos orientados a tareas[7] ya que corresponde a las relaciones secuenciales entre los segmentos.

  Esta relación ayuda durante la planeación de un discurso a determinar cuándo un segmento puede desarrollarse en paralelo con otros.

En procesamiento de lenguaje natural, la identificación de la estructura de intenciones se realiza buscando un PSD por cada segmento, y determinando las relaciones entre los objetos encontrados; si se llegan a encontrar varios propósitos dentro de un segmento tentativo, la estructura de segmentos puede modificarse introduciendo nuevos segmentos. En general, se requerirá menos procesamiento conforme los participantes hagan más claras las relaciones entre los distintos PSD, lo que pueden lograr haciendo explícitos los límites de segmentos.

---

[7] Un discurso orientado a una tarea consiste de una secuencia de instrucciones o pasos, en los que el orden de ejecución de las acciones es importante: un paso debe realizarse o satisfacerse antes de empezar el siguiente.

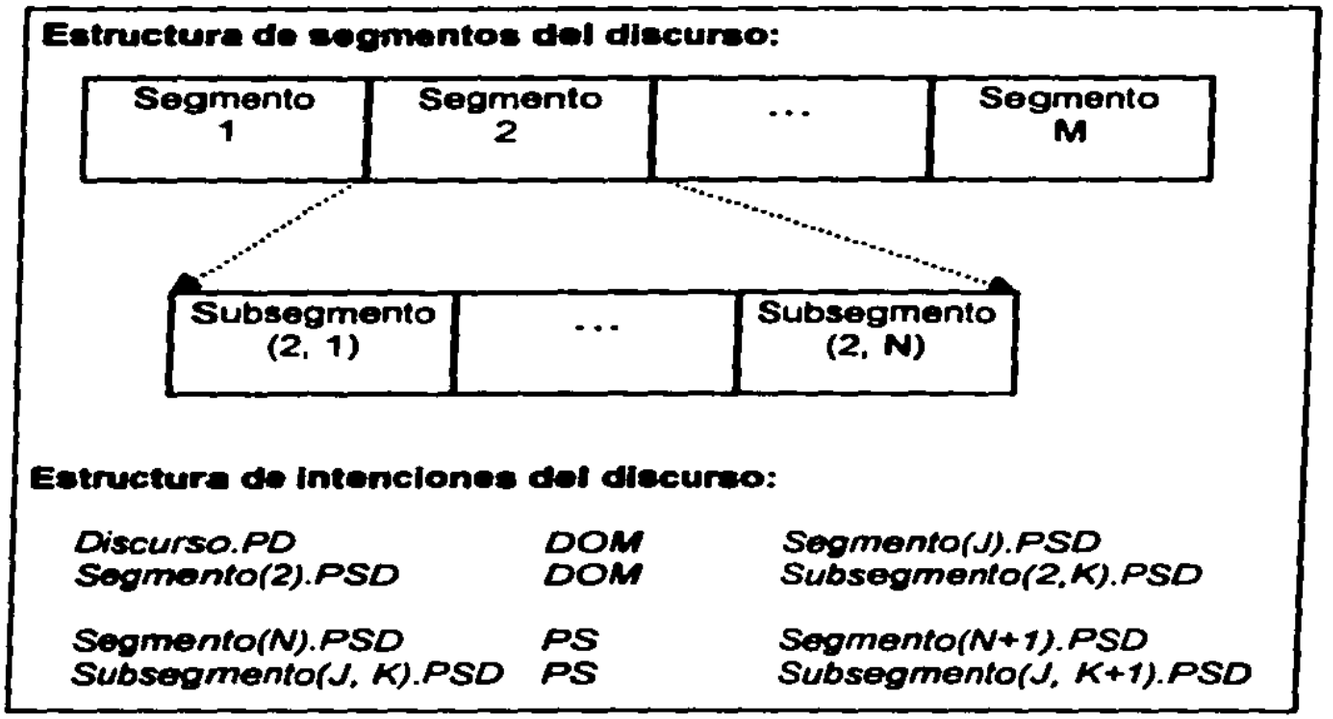

Figura 2.9.- Estructuras de segmentos e intenciones

2.3.3 Estructura de enfoque

El tercer componente del discurso es el **estado de atención**, que es una abstracción del foco de atención que mantienen los participantes conforme se desarrolla el discurso. El estado de atención contiene información respecto a los objetos, propiedades y relaciones que resultan más sobresalientes en cada momento, por lo que se trata de una propiedad del discurso en sí y no de sus participantes.

Se le modela a través de un conjunto de **espacios de foco** y una **estructura de enfoque**; cada segmento tiene asociado un espacio de foco para guardar su PSD y sus entidades más sobresalientes. Conforme aparecen segmentos y ocurren cambios en el estado de atención durante el procesamiento del discurso, se agregan y eliminan espacios de foco de la estructura de enfoque de manera análoga a un stack: en el tope se encuentra el espacio de foco del segmento actual, que es eliminado (*pop*) al hallar un fin de segmento; cuando aparece un subsegmento, se mete (*push*) su espacio de foco; si un PSD tiene una relación de satisfacción con el siguiente, su espacio de foco es reemplazado por

el nuevo. De esta forma, los objetos más recientes están disponibles cerca del tope del stack[2].

La estructura de enfoque no reemplaza a la de intenciones, porque sólo refleja una parte de esta última: los PSDs correspondientes a los segmentos anidados en el momento actual. Se trata principalmente de un auxiliar para el procesamiento, y su manipulación depende de la segmentación y la estructura de intenciones.

2.3.4 Ejemplo de reconocimiento de discurso

Como ejemplo de reconocimiento de estructura de discurso, se analiza un protocolo posible para llamadas a procedimientos remotos. Suponiendo que el cliente ya ha establecido contacto con el servidor, la comunicación puede desarrollarse como en la figura 2.10.

E1	Cliente:	¿Está soportada la función XYZ?
E2	Servidor:	Sí.
E3	Cliente:	Aquí está el parámetro 1. Es un real de precisión sencilla.
E4	Cliente:	11011101100100110011001l...
E5	Servidor:	(Acepta si es el tipo del parámetro 1 de XYZ)
E6	Cliente:	Aquí está el parámetro 2. Es un arreglo de 25 enteros largos.
E7	Cliente:	0101000100100111101001....
E8	Servidor:	(Acepta si es el tipo del parámetro 2 de XYZ)
E9	Servidor:	(implícitamente, al terminar de recibir los parámetros, ejecuta)
E10	Servidor:	Aquí está el resultado. Es un real de precisión sencilla.
E11	Cliente:	(Acepta si concuerda con el resultado que esperaba)

Figura 2.10.- Diálogo para llamada RPC

---

[2] Hay algunos casos en los que la estructura de enfoque no puede manejarse exactamente como un stack: por ejemplo, en las disgresiones, cuando uno de los participantes en el discurso quiere regresar al ambiente de un segmento anterior (en una frase como "respecto a X, se me olvidaba decirles que...").

Se encuentran tres fases dentro de este discurso: determinación de disponibilidad del servicio en el servidor, envio de parámetros por parte del cliente (y revisión de los mismos por parte del servidor), y la ejecución del servicio con la entrega de resultados. Cada fase tiene un propósito específico y corresponde a un segmento; aunque podria hacerse una descomposición más fina, este nivel es suficiente para el ejemplo.

Las estructuras lingüísticas y de intenciones presentes en este ejemplo se muestran en la Figura 2.11. El estado de atención va cambiando durante el procesamiento, de acuerdo con los esquemas que se ilustran en la figura 2.12.

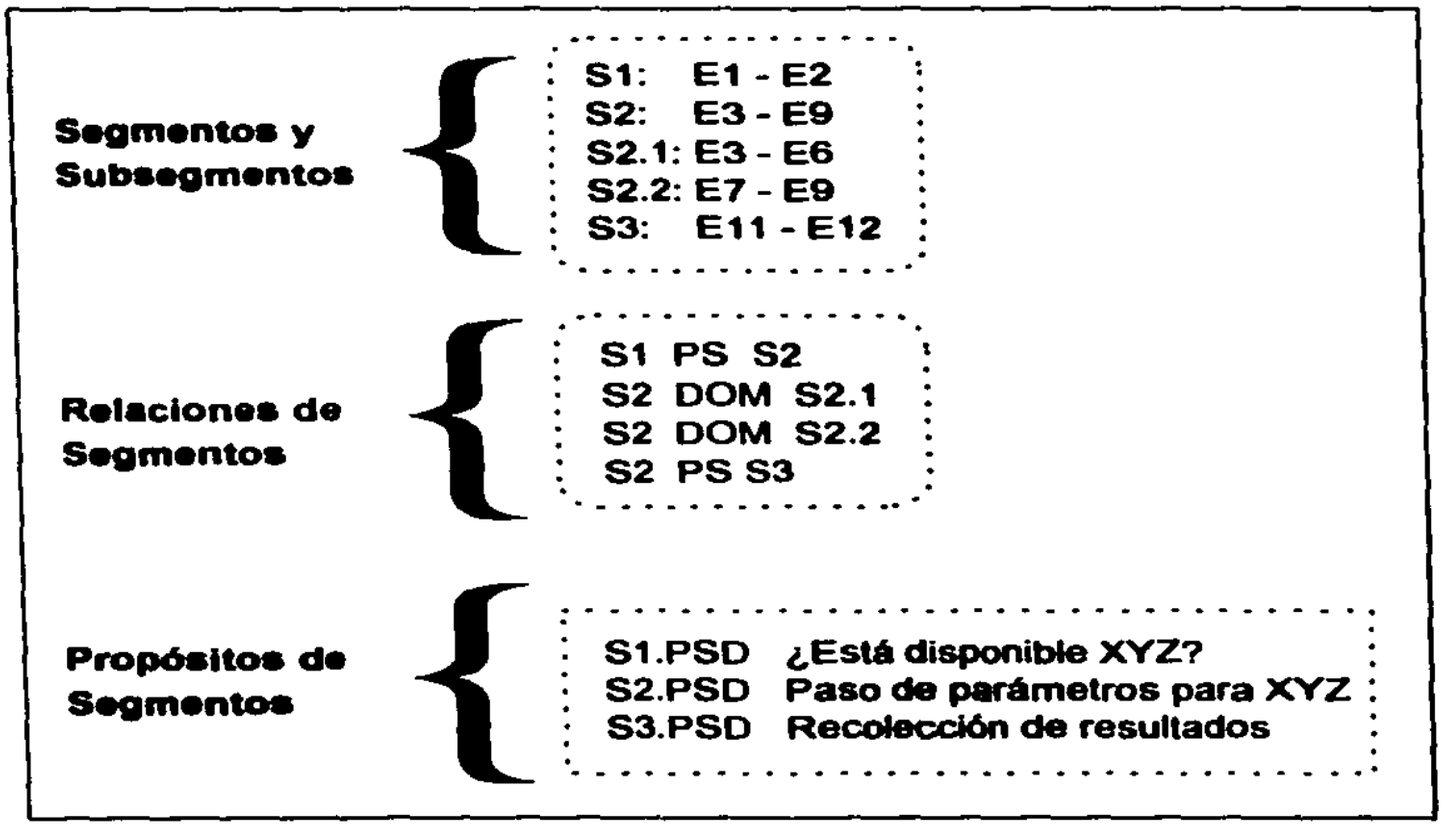

Figura 2.11.- Estructuras de Segmentos e Intenciones en ejemplo RPC

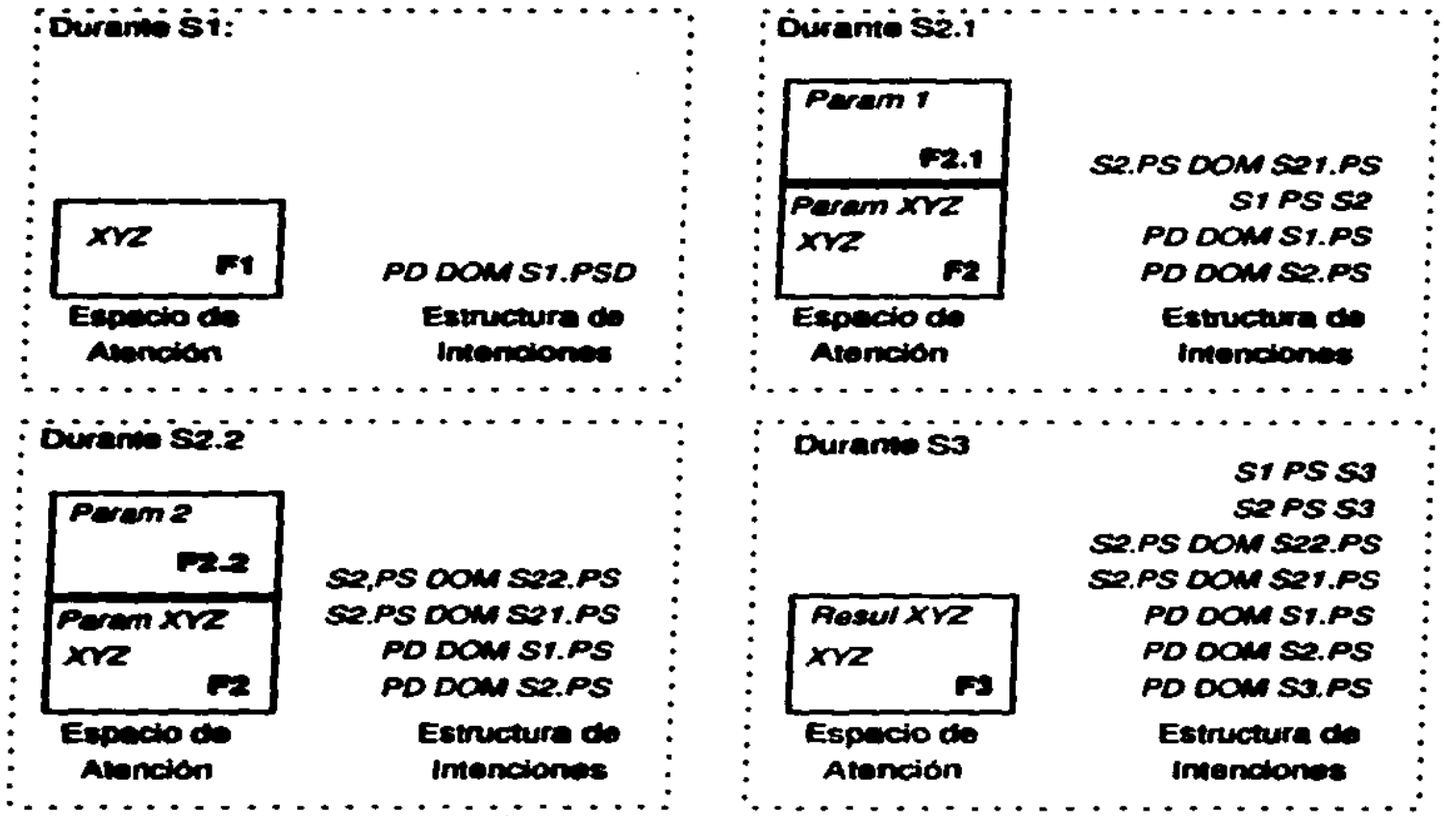

Figura 2.12.- Procesamiento del discurso RPC.

2.4 Comentarios

La comunicación basada en actos de habla permite expresar las intenciones de los hablantes y su reconocimiento posterior por los escuchas. Es un modelo general, y ampliamente empleado por los sistemas basados en agentes.

Un agente mantiene diversas actitudes respecto a la información que maneja: puede preguntarla, afirmarla, creerla, requerirla, comprometerse a realizarla, interesarse en ella, ofrecerla, etc. Al participar en un grupo social, se hace necesario que cada agente distinga las actitudes que los demás mantienen respecto a la información que él maneja: por ejemplo, si un agente sabe que otro agente requiere un conocimiento que él posee, puede decidir comunicárselo y esta estrategia posibilita la cooperación entre agentes. La distinción que hace Searle de los componentes de actos de habla (dato transferido o contenido proposicional, y una actitud o fuerza ilocutiva) indica que se puede hacer algún trabajo en la comunicación si se reconoce o actúa frente a uno solo de estos componentes.

lo que tiene implicaciones positivas para que un ambiente heterogéneo. Esta es una justificación para realizar la distinción entre el lenguaje de contenido del mensaje y el lenguaje de descripción del mensaje mismo.

Por su parte, las bases que Cohen y Perrault dan para integrar los actos de habla como operadores de un plan, enlazándolos mediante precondiciones y efectos de ejecución, puede emplearse en tiempo de ejecución como mecanismo de descubrimiento, aprovechamiento y comprensión del uso de recursos. Sin embargo, la definición de los operadores de Cohen y Perrault, y las cuestiones de semántica y modelos de los demás no se incorporan al sistema de control de diálogos que se presenta en este trabajo, aunque son necesarias para modelar correctamente el entendimiento y condiciones de formulación de actos de comunicación. Como se recordará, estas definiciones dependen del tipo de interacción deseado para los agentes, así que se deja el problema de la interpretación para los niveles superiores del facilitador.

Respecto al procesamiento e interpretación del discurso, el modelo de Grosz y Sidner proporciona bases para entender la estructura de un diálogo o conversación, destacando el papel que juegan las intenciones e identificando componentes centrales a considerar para el soporte de comunicación: el segmento de discurso, el PSD, y la estructura de foco.

KQML debe pensarse como una base de reutilización de diseño para desarrollar sistemas basados en agentes; salvo para los agentes más simples, es difícil ver cómo un agente que hable KQML podría comunicarse fácilmente con otro agente KQML desarrollado independientemente. Buena parte del problema se debe a que la especificación original [Finin *et al*, 1993] es demasiado informal y ambigua, aunque recientemente Yannis Labrou [1996] ha realizado un trabajo que pretende solventar estos inconvenientes al proporcionar una definición más clara y formal.

Capítulo 3

Diseño del sistema

La capacidad comunicativa es el factor común en los participantes de un ambiente heterogéneo como el de los facilitadores, así que resulta útil emplear un modelo de comunicación expresivo que esté al alcance de todos ellos. Con ese fin, se emplean los mensajes basados en actos de habla y se distinguen dos niveles comunicativos, en base a las habilidades requeridas para tratar el conocimiento compartido:

- El sistema de control de diálogos, que está presente en todos los facilitadores y da servicios para crear, manejar, enviar y recibir mensajes con un formato común, y para definir contextos empleados en la interpretación de los mismos.
- Componentes de nivel superior, que manejan las ontologías, reconocen las fuerzas ilocutivas e interpretan los mensajes. La sofisticación de estos componentes depende de la granularidad del facilitador correspondiente.

Además de proveer mecanismos para posibilitar la comunicación de alto nivel, que se considera fundamental para la actividad cooperativa, el sistema de control de diálogos debe permitir la experimentación de nuevas ideas, ya que hay muchas propuestas de arquitecturas y protocolos de coordinación [Barbuceanu y Fox, 1995; Sidner, 1994; Chang y Woo, 1991], y se han planteado objeciones razonadas a los estándares mismos [Cohen y Levesque, 1995]. No se implantan estrategias específicas de negociación, coordinación u otra actividad cooperativa, ni se supone un método particular de programación de los agentes (reactivo, declarativo, procedural, etc.); se pretende que estas decisiones se tomen en el momento de la implementación del agente.

3.1 Arquitectura

El sistema de control de diálogos es un componente de cada facilitador, y maneja los siguientes elementos principales (ver figura 3.1):

- **Mensaje**. Este módulo permite crear y manipular mensajes basados en actos de habla, formados por una fuerza ilocutiva, un contenido, y una serie opcional de parámetros valuados que sirve como mecanismo de extensión. Cada capa del sistema de control de diálogos tiene que ver de alguna manera con los mensajes.
- **Transporte**. Los mensajes pueden transmitirse a través de diferentes canales o medios de transporte de una manera transparente para los niveles superiores del facilitador. Por la heterogeneidad de los facilitadores, se busca un diseño portable que pueda emplearse en diferentes ambientes de programación y ejecución,

empleando como denominador común los protocolos de Internet (TCP, HTTP, etc.) La capa de transporte convierte los mensajes a una representación externa que pueda ser transmitida por la red; actualmente se emplean cadenas ASCII que siguen la sintaxis de KQML.

- **Diálogo**. Contiene elementos básicos de manejo de discursos, para ayudar a mantener el contexto de comunicación durante el procesamiento de los mensajes, y para que se usen racionalmente los recursos de la capa de transporte al acotar los tiempos y espacios en que se les emplea. Es la base para definir protocolos de coordinación.

Los demás componentes del facilitador relacionados con el sistema comunicativo se agrupan bajo el concepto de **capa de acción**. Estos componentes determinan cuándo pueden efectuarse las operaciones de entrada y salida, interpretan los mensajes recibidos, manipulan los diálogos, y manejan las ontologías del facilitador:

- **Acción**. El diseño e implementación de esta capa es dependiente del ambiente operativo, lenguaje y facilidades de programación empleadas en el facilitador. Se requiere que la capa de diálogos pueda emplearse desde diferentes tipos de capas de acción.

Como parte de la validación del sistema de control de diálogos, se diseñó e implementó una capa **Acción** para programar facilitadores que funciona como un sistema multitarea con despacho basado en la recepción de mensajes. De manera experimental, también se diseñó e implementó un servicio a nivel de **Acción** que permite la definición y ejecución de protocolos de coordinación independientes de ontologías.

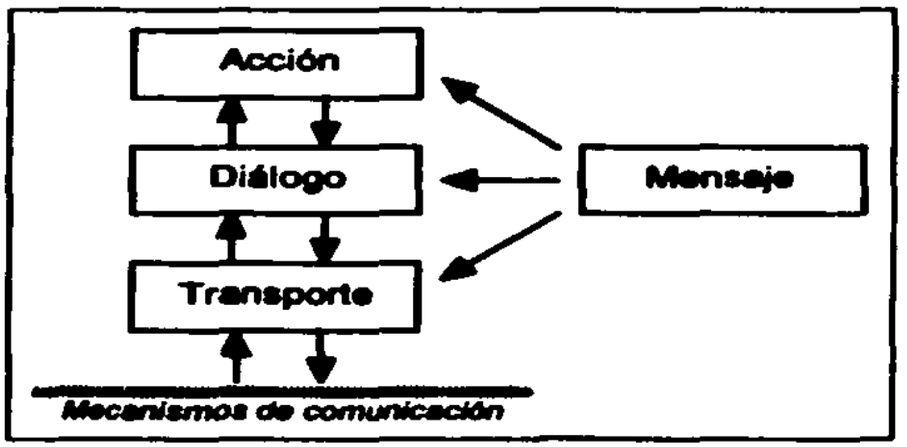

Figura 3.1.- Esquema de capas

3.2 Mensajes

Las funciones de este nivel (ver figura 3.2) permiten crear y destruir mensajes, y definir y leer sus parámetros, encapsulando los detalles del lenguaje de representación de mensajes (como la exigencia de KQML de responder a los mensajes que tengan un parámetro `:reply-with`, se usa su valor para crear un parámetro `:in-reply-to` en el mensaje de respuesta). No se considera necesario que un facilitador soporte varios lenguajes de descripción del contenido[1].

Los intercambios de mensajes entre facilitadores siempre se realizan dentro de un diálogo. Un objeto **Mensaje** (ver figura 3.3) tiene como atributos mínimos una fuerza ilocutiva, un contenido, un conjunto de pares nombre/valor (en el estilo de KQML y CORBA), y los identificadores de emisor, receptor, y de los diálogos asociados dentro de cada facilitador. Los mensajes pueden obtenerse de tres maneras: creando un mensaje vacío y asignando valores a sus atributos; por recepción a través de la red (en cuyo caso hay un proceso de reconocimiento sintáctico y conversión); o como respuesta a otro **Mensaje**.

MENSAJES	
d_mensaje_crear(Realizativa, Cont)	Nuevo mensaje
d_mensaje_aridad(Msj)	Aridad del mensaje
d_mensaje(Msj, Propiedad)	Valor de propiedad
d_mensaje(Msj, Propiedad, Valor)	Modificar valor de propiedad
d_mensaje_accesar(Msj, N)	N-ésimo par (Propiedad, Valor)
d_diálogo(Msj)	Diálogo asociado al mensaje

Figura 3.2.- Funciones para manejo de mensajes

Cuando se crea una respuesta a un mensaje, se manipulan los atributos de emisor, receptor, e identificadores de diálogo interno y externo, para obtener continuaciones y conclusiones. Las primeras contienen el identificador del diálogo de contexto, permitiendo al receptor del mensaje dirigir su respuesta a él e hilar la conversación, mientras que una conclusión no espera una réplica futura.

---

[1] La traducción entre diferentes lenguajes de contenido no es un servicio básico de comunicación. Puede tenerse un agente que conozca varios lenguajes de contenido, y pedirle a él que haga las traducciones.

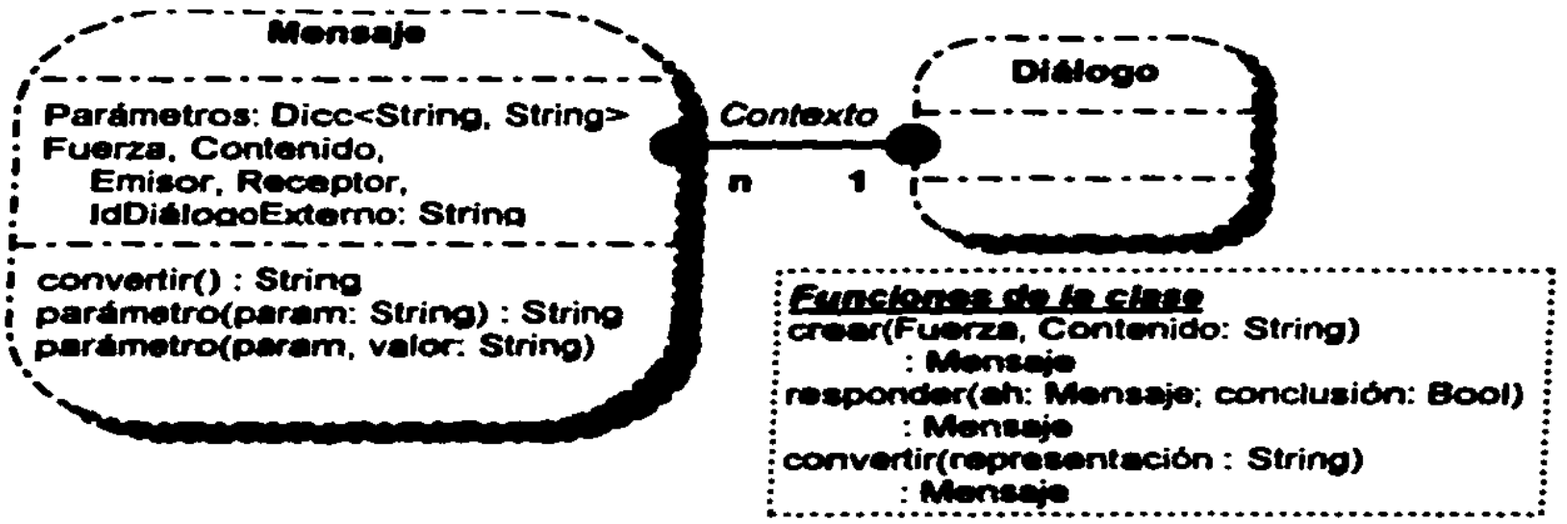

Figura 3.3.- Mensaje

3.3 Transporte de mensajes

Las funciones de este nivel permiten enviar y recibir **Mensajes** a través de diferentes protocolos de comunicación que están representados por la noción de **Canal**. No son directamente accesibles a los programadores de facilitadores, ya que siempre se les usa a través de un diálogo.

3.3.1 Requerimientos

Para el mantenimiento de los diálogos se requiere, al igual que para KQML, una capa de transporte de mensajes (**Transporte**) que asemeje el intercambio de mensajes punto a punto sobre canales de comunicación que cumplan con las siguientes características:

- Los mensajes se transmiten confiablemente, sin fragmentación y respetando el orden de emisión.
- Se puede identificar el canal de entrada sobre el que llega un mensaje particular.
- Se puede especificar del canal de salida por donde debe enviarse un mensaje.

Además de estos requerimientos, **Transporte** debe:

- Permitir el uso de varios protocolos de comunicación, dando mecanismos para añadir otros cuando sea necesario.
- Considerar las cuestiones de heterogeneidad. O sea, consumir el mínimo de recursos tales como procesos o *threads*, e integrarse con las facilidades de entrada y salida que ofrezca el ambiente de programación. Concretamente, esto implica que **Transporte** no debe crear procesos adicionales, y que no va a tener

el control sobre los momentos en que se pueden leer, escribir o esperar mensajes, sino que será notificado cuando pueda hacerlo.

El asunto de la heterogeneidad distingue a **Transporte** de otros sistemas semejantes. Por ejemplo, KAPI [Weber y Kuokka, 1995] está muy ligado a Unix y es difícil portarlo a otros ambientes; cuando el programa corre bajo X11, KAPI impide que las ventanas del programa se refresquen mientras espera la recepción de un mensaje[2]; además, KAPI asume el modelo de entrada y salida con bloqueo, que degrada el desempeño de sistemas tales como Windows 3.1 y Mac.

3.3.2 Diseño

Un objeto **Canal** comunica a una dirección física (*host* y puerto) a través de un protocolo específico; **Transporte** crea un **Canal** a partir de una cadena de caracteres que representa el URL del destino. Los **Canales** pueden ser tanto puntos de recepción como de transmisión de mensajes, y su implementación requiere de objetos de más bajo nivel que permitan identificar la ocurrencia de eventos de red (lectura, escritura o excepciones) sobre cualquiera de los **Canales** abiertos. Para el caso del sistema actual, donde se emplean sólo los protocolos de Internet, los sockets del sistema operativo son suficientes.

Se dice que un **Canal** está activo cuando es posible leer o escribir datos en él inmediatamente, sin bloquear la ejecución del programa. Para evitar los problemas de bloqueo, **Transporte** sólo trata con canales activos; la detección de estos la realiza **Acción**, que conoce los detalles del ambiente de programación y llama a **Transporte** cuando detecta un canal activo[3].

Los métodos principales de un **Canal** son **eventos**, **atender** y **enviar**. El primero de estos indica qué eventos interesan en ese instante al **Canal** (lectura, escritura, excepciones, o alguna mezcla de los tres). Cuando haya ocurrido uno de estos eventos, **Acción** invocará a **atender**, que reacciona al evento leyendo datos hasta formar un

---

[2] El problema se debe a que la comunicación del proceso con el servidor X se realiza a través de un socket no disponible a KAPI: cuando éste hace una espera sobre los sockets que tiene abiertos, ya no se atienden los eventos de X. Hay dos soluciones: crear un proceso adicional donde se ejecute KAPI, o modificarlo para que use entrada y salida no bloqueante y haga accesibles sus sockets a X11, quien seguiría atendiendo sus eventos y avisaría a KAPI cuando alguno de sus sockets esté activo.

[3] Esta detección depende tanto del sistema operativo como del modelo de programación del ambiente o lenguaje en que se programe el facilitador. Un proceso en Unix controla su ejecución hasta que ejecuta la función `select`, que entrega sus sockets activos. En Windows, el medio óptimo es recibir mensajes asíncronos en un procedimiento de ventana, cuyos parámetros indican qué socket está activo.

mensaje o transmitiendo partes de un mensaje depositado previamente mediante enviar (ver figura 3.4). Opcionalmente, un Canal puede tener asociado un objeto Notif, al cual se envían notificaciones de sucesos relacionados al canal (como atención de eventos o destrucción del objeto por fallas en la comunicación), permitiendo que otras partes del sistema reaccionen a los cambios que le ocurran al canal.

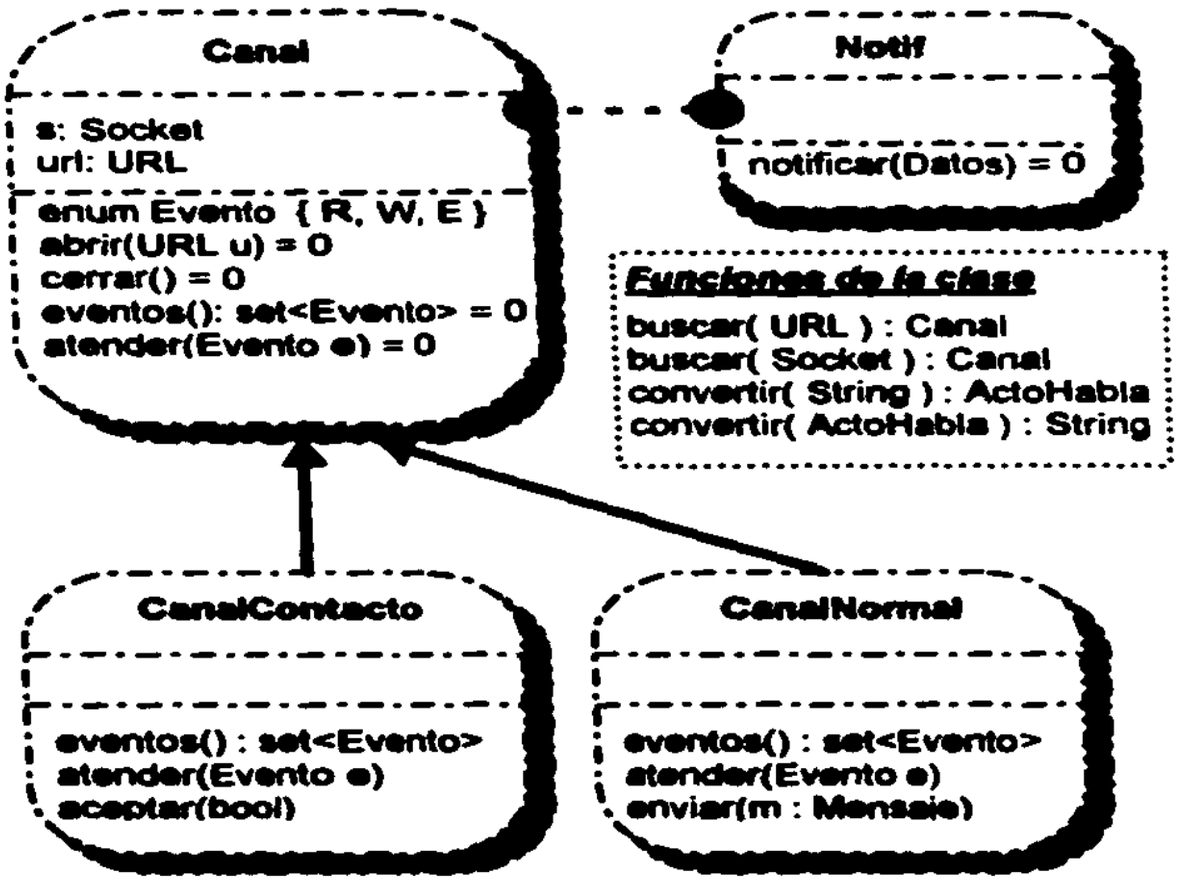

Figura 3.4.- Clase Canal

Transporte recibe las direcciones físicas en forma de cadenas de caracteres. Para crear objetos a partir de esas cadenas, se emplea un esquema de fábricas de objetos: cada clase X derivada de Canal crea un objeto derivado de FábricaCanal que sabe cómo crear instancias de X a partir de las cadenas. Al recibir un URL, la clase FábricaCanal busca entre sus instancias un objeto que lo reconozca, dependiendo de qué se desea hacer con el nuevo canal: enviar o escuchar solicitudes de transmisión de datos (ver figura 3.5).

Figura 3.5.- Fábrica de canales

Las figuras 3.6 y 3.7 muestran cómo usar las clases Canal y FábricaCanal para proporcionar el soporte al protocolo TCP; las clases que derivan de ellas redefinen los métodos abstractos para proporcionar los nuevos servicios. En la figura 3.8 se muestra el escenario de transmisión de un mensaje.

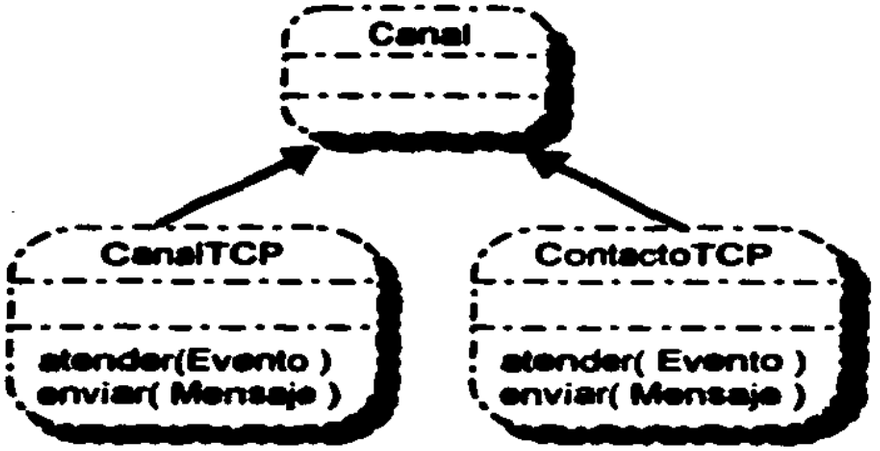

Figura 3.6.- Clase CanalTCP

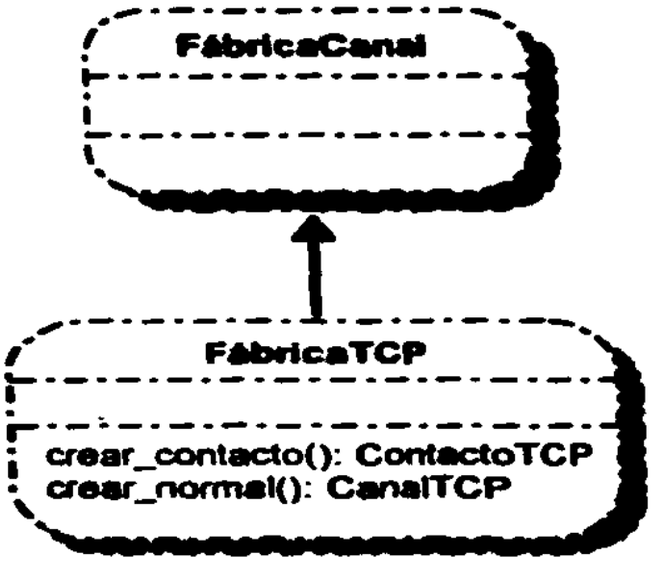

Figura 3.7.- Clase FábricaTCP

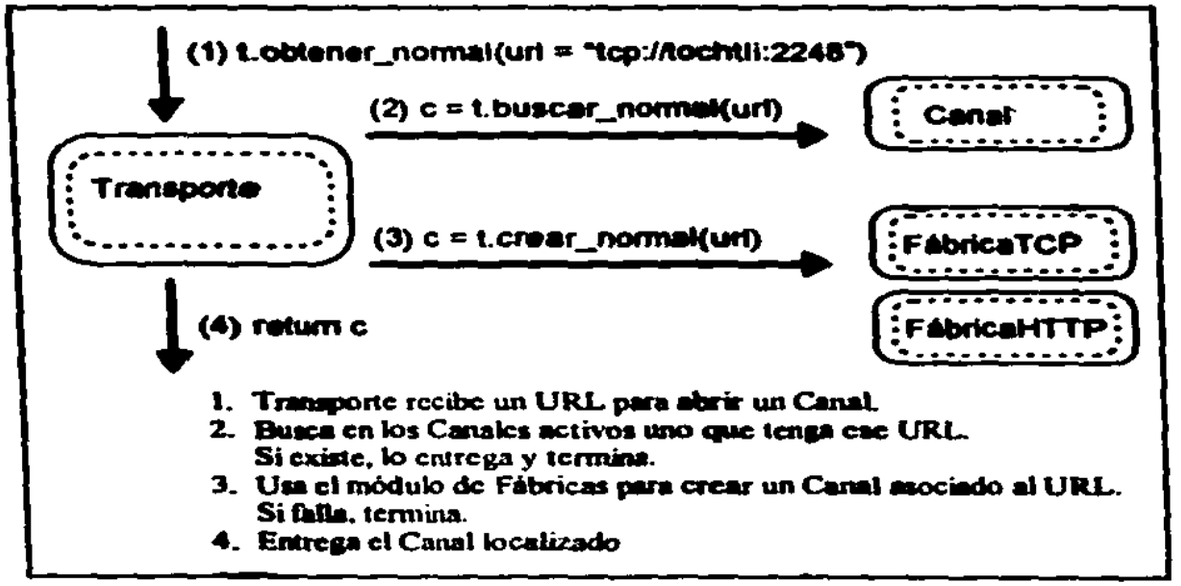

Figura 3.8.- Envio de mensaje

3.4 Diálogos

La capa Diálogo provee servicios asociados a la estructura de atención del discurso, no a la estructura completa de éste. Se llama contexto de diálogo, o diálogo para simplificar, al equivalente del espacio de foco de Grosz y Sidner [1986]; es un objeto que almacena información acerca de un segmento del discurso. El momento de su creación y terminación son determinados por cada participante del discurso; no hay marcas explícitas de fin de segmento, porque la determinación de éstas corresponde al nivel de coordinación, dentro de la capa Acción.

Los mensajes sólo se pueden transmitir y recibir a través de un diálogo, y llevan asociados los identificadores de los diálogos generador y destino. Para uniformizar, los errores de comunicación y *time-outs* se reportan en el diálogo correspondiente mediante mensajes generados por el sistema.

Un facilitador puede mantener varios diálogos simultáneamente. La inicialización del sistema comunicativo crea siempre un diálogo raíz, que recibe los mensajes dirigidos al agente que no tienen asociado un diálogo destino, y a partir del cual se pueden crear los demás diálogos.

Definimos tres operaciones (SUB, SEQ, FORK) que, dado un diálogo, crean otro que mantiene respecto a él una de las tres relaciones que se muestran en la figura 3.9. Estas relaciones entre diálogos crean una estructura de cactus, donde las ramas son stacks[4] (ver figura 3.10). Salvo FORK, que permite el soporte de diálogos múltiples, dichas relaciones corresponden a las definidas entre segmentos de discurso en el capítulo 2.

Cada diálogo tiene asociado un contexto local formado por variables, definidas por el usuario del diálogo para almacenar los atributos del segmento de diálogo (grupo de participantes, participante iniciador, PSD, tema, etc.). Las variables sólo existen durante la vida del diálogo al que pertenecen, pero los subdiálogos pueden leer y modificar las variables de su diálogo generador, de acuerdo a las relaciones de la figura 3.10. La búsqueda de valores de variables y destinos en el diálogo se realiza primero en el ámbito local y procede hacia el diálogo generador, quien repite el proceso.

La figura 3.11 muestra las funciones de manejo de diálogos.

---

[4] El empleo de una estructura tipo stack facilita la implementación de la capa **Diálogo**, aunque limita las habilidades de procesamiento del discurso en situaciones tales como las disgresiones (ver [Grosz y Sidner, 1986]). El modelo resultante es, sin embargo, suficientemente general para nuestros propósitos.

RELACIÓN	OPERACIÓN	DESCRIPCIÓN
Dominio	SUB	El diálogo nuevo es un subdiálogo del original, inhibe la recepción de mensajes de este último, pero tiene acceso a las variables que estén definidas.
Precedencia	SEQ	El diálogo nuevo reemplaza al original. Desde el punto de vista dinámico (mantenimiento de la estructura de focos), equivale a definir operaciones de apertura y cierre de diálogos. Tiene más utilidad desde el punto de vista de las acciones que interpretan los mensajes del diálogo.
Bifurcación	FORK	El nuevo diálogo es independiente del original, pero puede emplear las variables que estén definidas.

Figura 3.9.- Operaciones para crear diálogos

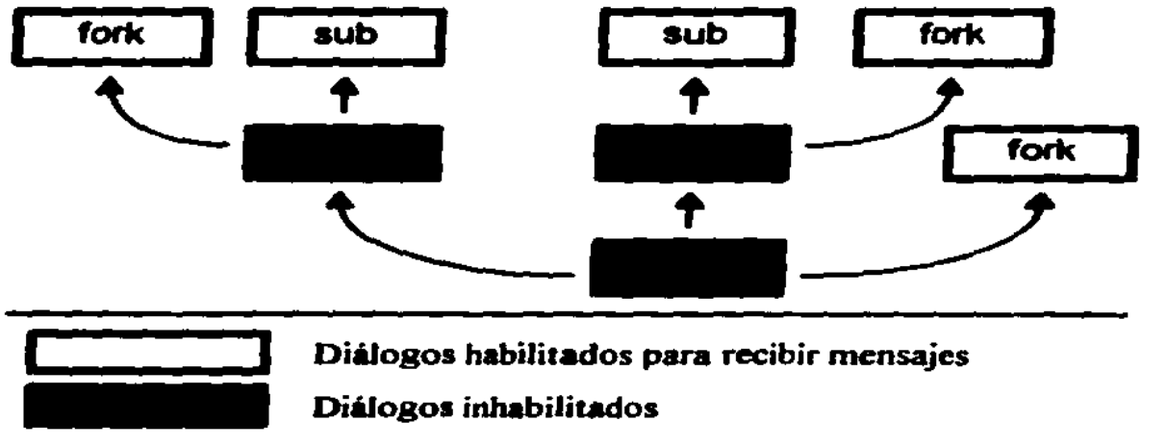

Figura 3.10.- Relaciones entre diálogos en el cactus

DIÁLOGOS Y SUS RELACIONES	
d_fork(Diag)	Nuevo diálogo (bifurcación)
d_call(Diag)	Nuevo diálogo (subdiálogo)
d_close(Diag)	Cerrar diálogo
ACCESO A VARIABLES Y MODIFICACIÓN DE SU AMBIENTE	
d_variable(Diag, Var)	Obtener valor de variable
d_variable(Diag, Var, Valor)	Modificar variable
d_nombre(Diag, Nom)	Obtener dirección a partir de nombre
d_nombre(Diag, Nom, Dirección)	Asociar dirección a nombre
d_default(Diag)	Leer URL destino por default
d_default(Diag, Url)	Asociar URL destino por default
ENVÍO Y RECEPCIÓN DE MENSAJES	
d_escuchar(Diag, Url)	Crear un canal de contacto
d_enviar(Diag, Msj)	Enviar mensaje
d_notificar(Diag, Msj)	Notificar mensaje

Figura 3.11.- Funciones para manejo de diálogos

Los objetos de la clase Diálogo (ver figura 3.12) tienen asociados un identificador, su diálogo generador, su subdiálogo inmediato (si existe), conjuntos locales de variables y nombres de destinos, un destino por default para los mensajes, y una acción a ejecutar cuando reciban un mensaje. Los métodos sub y fork construyen diálogos nuevos que se encuentran en relación de subdiálogo o bifurcación; en ambos casos, el diálogo sobre el que se aplica el método pasa a ser generador del nuevo.

La figura 3.13 ilustra el escenario de transmisión de un mensaje a través de un diálogo. La operación contraria, la recepción de mensajes, pertenece a la capa Acción porque implica una decisión respecto al control de flujo del programa y requiere soporte por parte del ambiente externo al sistema de control de diálogos.

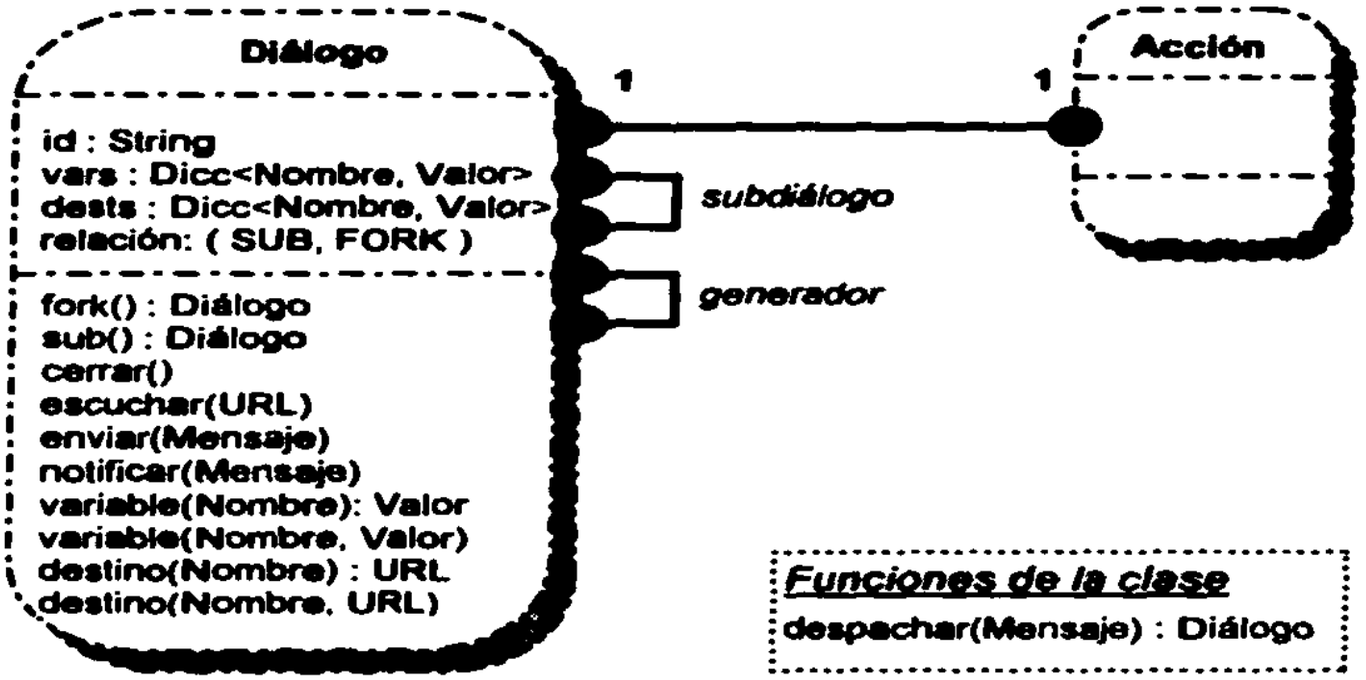

Figura 3.12.- Clase Diálogo

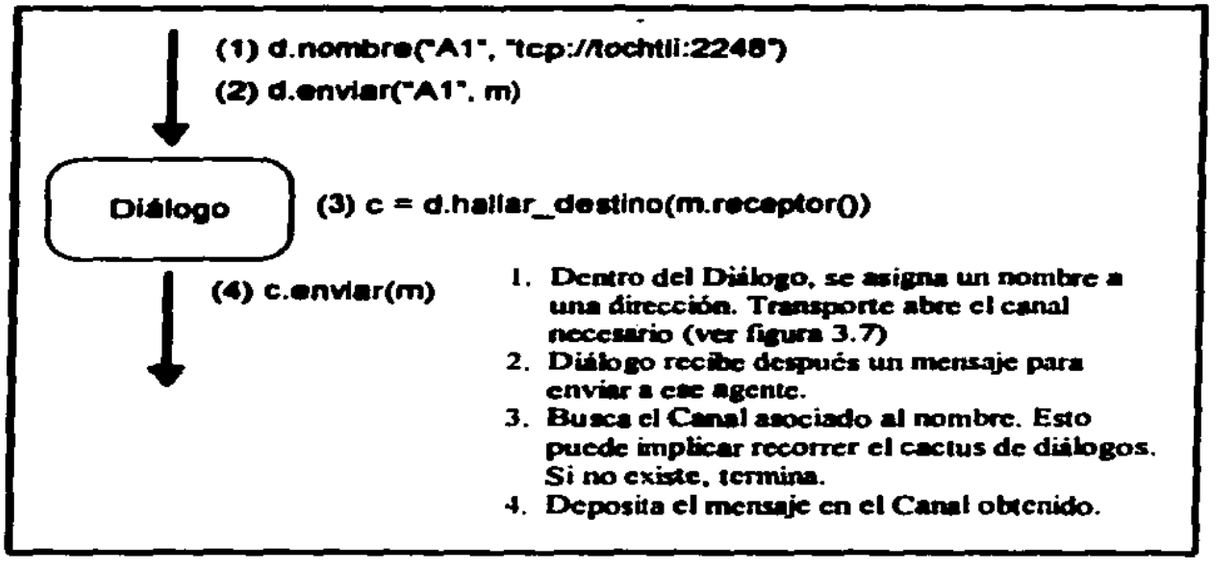

Figura 3.13.- Transmisión de un mensaje

3.5 Acción

Este nivel hace abstracción de las capas superiores del agente; **Acción** es quien lleva el control de la ejecución de **Diálogo** y **Transporte**, decidiendo cómo y en qué momento se atienden los canales, y cuándo y de qué manera se pueden interpretar los mensajes recibidos. Por tanto, su forma exacta está determinada por la arquitectura interna elegida para el facilitador, el lenguaje en que está programado, y el sistema operativo en que corre.

La interfaz entre los diálogos y la implementación de la capa de **Acción** puede seguir diferentes modelos, que presentan analogías con los modelos empleados para el diseño de interfaces al usuario (ver figura 3.14). Las analogías sugieren situaciones en las que un modelo es preferible a los demás.

MODELO	DESCRIPCIÓN
Entrada y salida en archivos	Los diálogos se tratan como archivos, leyendo o escribiendo en ellos y deteniendo la ejecución hasta que terminan, con éxito o error. Su implementación requiere algún tipo de soporte multitarea y mecanismos de sincronización, para detener una operación sin afectar la realización de otras acciones en el sistema.
Reactivo	La llegada de un evento (recepción de mensaje o error) provoca la ejecución de una acción asociada. Su implementación no necesariamente requiere soporte multitarea, porque los intérpretes de acciones pueden verse como procesos que sólo reciben tiempo para ejecución cuando llega un mensaje dirigido a su diálogo. Es apropiado para interacciones sumamente dinámicas, como lo demuestra su uso extendido en programación de interfaces gráficas al usuario.
Mixto	El programador decide en qué momentos aplicar alguno de los modelos anteriores. Estableciendo una correspondencia entre un diálogo comunicativo y una interfaz gráfica al usuario, este modelo equivale a focalizar temporalmente la interacción del usuario con el programa, inhabilitando los demás elementos de la interfaz (como hacen los diálogos modales de Windows).

Figura 3.14.- Modelos de acción

3.5.1 Máquina de acciones

El componente principal de la capa **Acción** es la máquina de acciones, que usa los servicios de **Transporte** para detectar los mensajes recibidos y luego despacharlos a los diálogos hacia donde van dirigidos, siguiendo las reglas implicadas por el cactus de diálogos (ver figura 3.15).

Figura 3.15.- Control de Acción

Cada diálogo tiene asociada una **Acción**, un ente que hace algo ante eventos tales como actos de habla recibidos o errores detectados y notificados por **Diálogo** (como problemas con la capa de red, *time-outs*, etc.). Al reaccionar, las acciones proporcionan la interpretación de los eventos.

Los objetos **Acción** constituyen una abstracción respecto a los modelos de capa **Acción** de la figura 3.14. Por ejemplo, en el caso de la metáfora de archivos, el objeto **Acción** depositaría el mensaje recibido en un buffer protegido por un monitor o estructura similar de sincronización de procesos.

3.5.2 Diseño de una capa Acción

La máquina de acciones que se presenta con este trabajo usa el modelo reactivo. Cada diálogo tiene un objeto **Interpretación** asociado, que es el que se invoca cuando el diálogo recibe un mensaje. Bajo este esquema el control de **Acción** funciona como un despachador de procesos, siguiendo las reglas de diálogos:

- La **Interpretación** asociada a un diálogo no puede procesar mensajes si se encuentra activo el subdiálogo. Haciendo una comparación con los sistemas de interfaz gráfica al usuario, esta situación es equivalente a la presentación de una ventana de diálogo que inhibe la respuesta de otras partes de la aplicación, como

suele serlo una caja de selección de archivos. En términos de Windows, es un diálogo modal.

- Las interpretaciones asociadas a diálogos en ramas diferentes del cactus pueden despacharse para ejecución de manera independiente. Regresando a la comparación con las interfaces gráficas, esto equivale a presentar ventanas diferentes dentro de la misma aplicación, o ventanas de diálogo que no inhiben la respuesta de otras partes de la aplicación aunque aparezcan al frente de las otras ventanas. Nuevamente en términos de Windows, se trata de aplicaciones con interfaces de documentos múltiples (MDI) y diálogos no modales (*modeless*).

El control de los subdiálogos tiene dos consecuencias importantes bajo este esquema de diálogos simultáneos y despacho en base a eventos:

- Si un diálogo tiene canales abiertos y posteriormente crea un subdiálogo, es posible que lleguen mensajes dirigidos a él mientras el subdiálogo está activo. Estos mensajes no pueden pasarse inmediatamente a interpretación, por lo que deben meterse en una cola global para ser entregados en cuanto terminen los subdiálogos correspondientes.
- Como el control de flujo lo decide la recepción de mensajes, se requieren mecanismos de sincronización entre las interpretaciones para que se puedan aprovechar los resultados obtenidos por un subdiálogo, o entre diálogos paralelos. Para mantener la homogeneidad en el sistema, se emplean mensajes especiales que se intercambian directamente entre los diálogos involucrados, sin intervención de la capa de transporte.

La figura 3.16 muestra las funciones que requieren soporte explícito de la capa **Acción**.

RELACIÓN CON DIÁLOGOS	
d_interpretar(Diag, Acción)	Indicar quién interpreta al diálogo
d_recibir(Diag)	Esperar llegada de mensaje al diálogo (*)
d_esperar(Diag)	Esperar a que terminen diálogos derivados
d_cerrar(Diag)	Cerrar un diálogo (requiere esperar a que terminen los diálogos derivados)
CONTROL GENERAL DEL FACILITADOR	
d_inicializar(URL)	Diálogo raiz
d_debug(Flag)	Habilitar seguimiento de diálogos
d_run()	Empezar ciclo de despacho de mensajes

(*) Sólo en caso de modelo de archivos o mixto.

Figura 3.16.- Funciones del nivel de Acción

3.5.3 Ejecución de protocolos dinámicos

Con fines experimentales, se diseñó una capa Acción especial que soporta la definición de protocolos, a través de autómatas que se pueden cargar y ejecutar dinámicamente a través de un motor de seguimiento. Durante la ejecución de un protocolo, se revisa que los mensajes generados y recibidos por el agente sean válidos en el estado actual, y se invocan acciones y condiciones mentales para transitar entre los estados, como en la Programación Orientada a Agentes. El objetivo es ayudar a depurar el comportamiento comunicativo de un agente y establecer interacciones independientes del dominio de aplicación, que estén disponibles a los participantes en el sistema.

La figura 3.17 muestra las funciones correspondientes al manejo de protocolos.

SOPORTE DE PROTOCOLOS	
d_cargar(Archivo)	Carga y entrega identificador de protocolo
d_ejecutar(Diag, Prot)	Ejecutar protocolo sobre un diálogo
d_transitar(Prot, Estado)	Validar y elegir un nuevo estado

Figura 3.17.- Funciones para manejo de protocolos

La descripción de un protocolo de interacción contiene transiciones que hacen referencia a estados origen y destino, condiciones mentales, acciones, y la fuerza ilocutiva de los mensajes; el lenguaje empleado se ejemplifica en la figura 3.18. Para implementar un protocolo, el programador asocia predicados o procedimientos a cada identificador de acción y condición mental.

En tiempo de ejecución, el motor de seguimiento carga e interpreta la descripción de los protocolos y los incorpora a una biblioteca que mantiene. Cuando el facilitador decide ejecutar alguno de esos protocolos, se crea un contexto de diálogo para su ambiente comunicativo y se realiza el ciclo de seguimiento: dado un estado actual y un evento de generación o recepción de mensaje, el motor de seguimiento busca las transiciones aplicables y ejecuta los procedimientos asociados a las acciones y condiciones mentales involucradas; si tiene éxito, el estado destino se convierte en el estado actual del autómata. Este mecanismo se complementa con la función "d_transitar", que usan las acciones para designar el siguiente estado cuando el protocolo permita no determinismo.

La clase que ofrece el servicio de ejecución de protocolos dinámicos es ProtocoloDin, que crea y administra internamente un contexto de diálogo para el intercambio de mensajes (ver figura 3.19).

```
PROTOCOL "Negociación"
STATES s1
ACTION inicializar, revisar_contraoferta

# Inicialización del protocolo
INITIALIZE TO s1
  ACTION inicializar
  OUT proponer
END

FROM s1 TO END
  WHEN   rechazar, aceptar
  IF     seguir_negociando
  ACTION revisar_contraoferta
  OUT    confirmar, cancelar
END

FROM s1 TO s1
  WHEN   contraproponer
  IF     seguir_negociando
  ACTION revisar_contraoferta
  OUT    modificar
END

END
```

Figura 3.18.- Descripción de protocolos

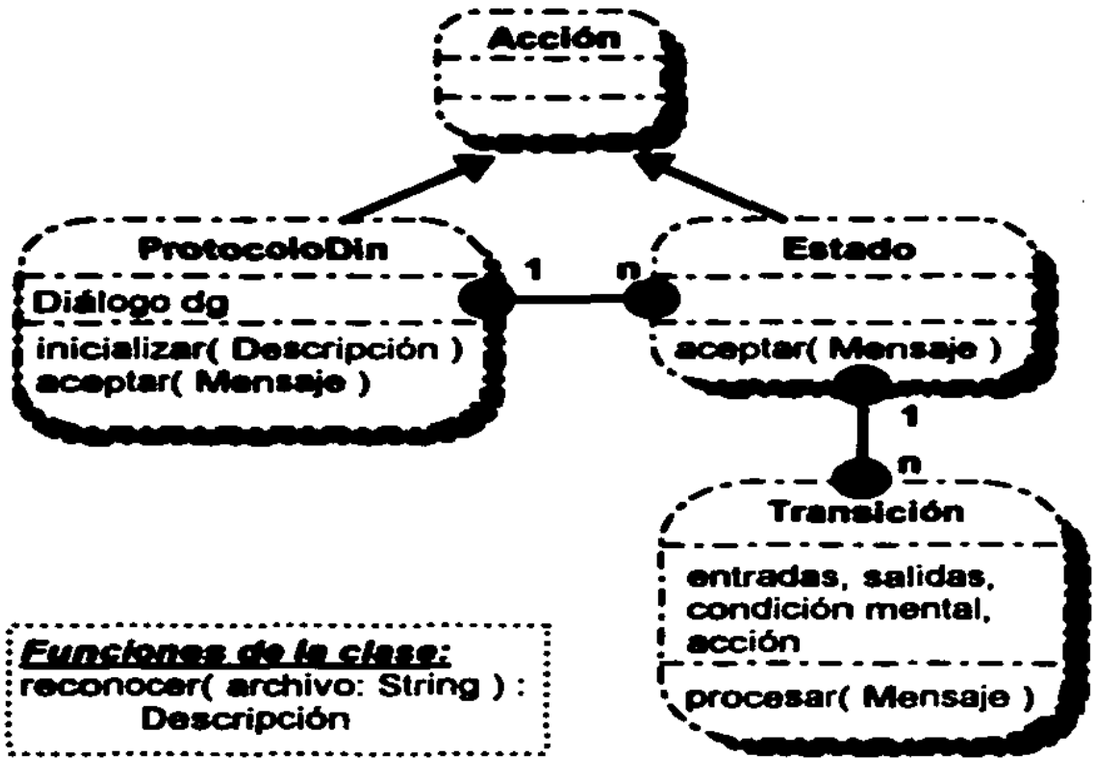

Figura 3.19.- Soporte de protocolos dinámicos

Capítulo 4

Resultados y conclusiones

En este trabajo se ha hecho una comparación de los mensajes basados en actos de habla con otros mecanismos de comunicación, y se ha presentado el diseño de un sistema de control de diálogos entre facilitadores basado en actos de habla. Con el fin de evaluar el diseño, se le implementó en un ambiente heterogéneo y se le utilizó para desarrollar una aplicación que requiriera cooperación.

Los resultados y conclusiones obtenidas se refieren tanto a los límites de la implementación del sistema de control de diálogos, como a la generalización de la experiencia con la comunicación basada en actos de habla en un ambiente heterogéneo.

4.1 Implementación

La implementación del sistema de control de diálogos se realizó en C++, y con el fin de evaluar la portabilidad del diseño se emplearon varias plataformas. En ambientes PC se utilizaron Windows 3.1, Windows 95 y Windows NT; en UNIX se usaron SunOS y Linux, y además se desarrollaron interfaces a los lenguajes SWI-Prolog y Tcl/Tk. Estas plataformas son representativas de las disponibles actualmente; por ejemplo, para adaptar el sistema de diálogos a la Mac, se puede empezar con la versión de Windows y usar las rutinas de MacTCP.

Los compiladores empleados (GNU C++ 2.6.2 en Unix, Borland C++ 4.0 para Windows 3.1, y Microsoft Visual C++ 4.0 para Windows 95 y NT) difieren bastante en su grado de implementación del estándar propuesto de C++, por lo que en el código se evitan características del lenguaje que no están bien soportadas (como el manejo de excepciones, las últimas innovaciones de plantillas, y la información de tipos en tiempo de ejecución). Por estas razones, será necesario revisar el código en el futuro.

El uso de C++ para la programación permitió detectar puntos genéricos, que pudieron expresarse mediante plantillas parametrizadas (*templates*). Sobre estos puntos de genericidad se comenta en detalle más adelante.

4.1.1 Transporte

El objetivo de esta capa es ocultar los detalles de comunicación propios del sistema operativo empleado, empleando una interfaz común de bajo nivel: los sockets TCP/IP. En Unix, **Transporte** usa los sockets definidos por el sistema operativo a través del API

estándar BSD, y detecta los sockets activos mediante la llamada al sistema `select`. En Windows, **Transporte** usa la biblioteca dinámica Winsock 1.1 y detecta los sockets activos mediante los servicios de notificación que ésta ofrece, evitando la emulación de `select` porque deteriora el desempeño de las demás aplicaciones en Windows 3.1.

Ambas implementaciones de **Transporte** tienen en común funciones que encuentran el objeto **Canal** asociado a un socket, lo que permite usar el sistema de diálogos con software que haga su propia detección de actividad.

4.1.2 Diálogo

La capa **Diálogo** proporciona funciones para definir y manejar los contextos para la interpretación de los mensajes. Maneja ámbitos de variables, nombres de participantes y las relaciones entre diferentes diálogos.

El código fuente de la capa **Diálogo** es el mismo en todas las plataformas soportadas. Las que cambian son las capas **Transporte** y **Acción**, que son la interfaz a los niveles superior e inferior de los facilitadores.

4.1.3 Acción

La forma y comportamiento de la capa **Acción** dependen del tipo de facilitador que se esté programando, y para desarrollarla la biblioteca de diálogos exporta un conjunto de clases y funciones primitivas que sirven como interfaz con **Transporte** y **Diálogo**. Una vez que el programador tiene una capa **Acción**, puede crear acciones normales o protocolos dinámicos y registrarlos con la máquina de acciones correspondiente.

Como las capas **Acción** de un mismo tipo son muy semejantes una vez inicializadas, se hizo un ejercicio de clasificación de las interacciones posibles entre un ente que lleve el control y el resto del sistema de diálogos. Como resultado, se obtuvo un catálogo de plantillas parametrizadas de C++ que implementan algunas de las funciones de interfaz identificadas en el diseño, y máquinas de acción para el modelo mixto de capa de **Acción** (ver figura 3.13). Para usar esas plantillas, los programadores deben definir clases que provean los servicios primitivos para la interfaz, e instanciarlas para obtener funciones de la capa **Acción**. La figura 4.1 ejemplifica este mecanismo.

Las interfaces a otros lenguajes de programación se implementaron como capas **Acción** adicionales:

- En el caso de SWI-Prolog 2.1.0 [Wielemaker, 1995], un Prolog desarrollado en la Universidad de Amsterdam que acepta invocaciones a predicados desde C, la capa Acción usa los servicios de Transporte para detectar los sockets activos. El control del facilitador lo lleva inicialmente Prolog pero lo cede a la máquina de acciones, quien despacha los mensajes. Al recibir un mensaje, Los diálogos

```
// PLANTILLA: API DE LA CAPA ACCIÓN
// Acc debe ser una clase definida por el programador que implemente
// los servicios básicos de la nueva capa Acción (como la conversión
// entre la representación de diálogos y los apuntadores a Diálogo
// de C++)
template<class Acc>
class ApiAcciones {
  ...
  static
    Acc::Dialogo do_fork(Acc::Dialogo dg);
  ...
};

// Implementación del API parametrizado
template<class Acc>
Acc::Dialogo
ApiAcciones<Acc>::do_fork(Acc::Dialogo dg)
{
  // Manejar conversiones necesarias para llamar a "fork" de Diálogos:
  Dialogo *pd = Acc::convertir(A, dg);
  return (Acc::Dialogo)(pd==0? 0 : (ApiDialogos::fork(pd)));
}

// Clase que define los servicios básicos para la interfaz a Tcl/Tk:
class InterfazTcl {
  ...
};

// Implementación de la capa Acción para Tcl/Tk
ApiAcciones<InterfazTcl> capaAccion;
```

Figura 4.1.- Plantilla de interacción

invocan a un manejador interpretaciones de las acciones (normales y protocolos) escritas en Prolog.

- En la interfaz a Tcl 7.3 y Tk 3.6 [Ousterhout, 1994; Welch, 1995], un lenguaje de script usado para crear interfaces en X11, el intérprete detecta por sí mismo los sockets activos y mantiene el control todo el tiempo. En este caso, la capa Acción proporciona los sockets activos a la máquina de acciones, quien sólo busca los canales correspondientes para despachar los mensajes a los diálogos, e invocar las acciones escritas en Tcl.

4.2 Aplicación: negociación de reuniones

Para evaluar el sistema de control de diálogos se desarrolló con él una aplicación típicamente cooperativa: la negociación de reuniones entre facilitadores asociados a usuarios humanos. La aplicación desarrollada y el protocolo de negociación utilizado se describen a detalle en [Palacios].

Cada usuario humano tiene asociado un facilitador ejecutivo (que sabe negociar reuniones) y un facilitador de staff (que maneja la agenda del usuario). El facilitador ejecutivo mantiene un modelo mental de su usuario, que usa junto con el tipo de reunión, la posición jerárquica de su usuario en la oficina, y la disponibilidad de tiempo en la agenda para decidir la aceptación de una invitación.

El protocolo de negociación empleado se resume en la figura 4.2. La negociación de una reunión empieza cuando un usuario (el promotor de la reunión) encarga a su facilitador ejecutivo que lo represente para invitar a un grupo de usuarios en una fecha y hora determinadas. El facilitador envia la invitación a los facilitadores de los invitados y espera sus respuestas, que pueden ser de aceptación, rechazo, o contrapropuesta; en este último caso, el promotor puede realizar modificaciones a la propuesta si se trata de un invitado importante. Eventualmente, el promotor notifica la confirmación o cancelación de la reunión a todos los invitados.

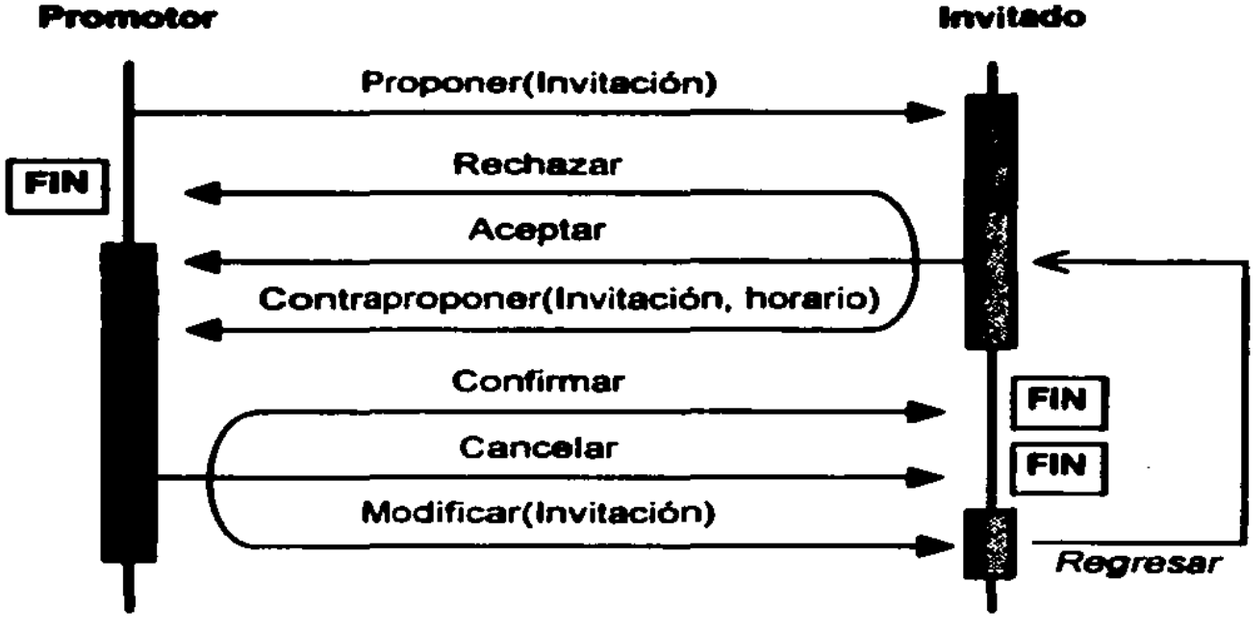

Figura 4.2.- Protocolo de negociación de reunión

Los facilitadores se programaron en SWI-Prolog, en dos versiones: una que emplea sólo los servicios básicos del sistema de control de diálogos, y otra que emplea el módulo de protocolos dinámicos con los autómatas que se muestran en las figuras 4.3 y 4.4. El sistema de control de diálogos permite a los facilitadores mantener varias negociaciones simultáneas, empleando los diálogos para distinguir cada contexto.

```
PROTOCOL  InicioNegociación
STATES    esperar_opinión
ACTIONS   formular_oferta, revisar_contraoferta

INITIALIZE TO esperar_opinión
  ACTION formular_oferta
  OUT    proponer
END

FROM esperar_opinión TO END
  WHEN   rechazar, aceptar
  ACTION revisar_contraoferta
  OUT    confirmar, cancelar
END

FROM esperar_opinión TO esperar_opinión
  WHEN   contraproponer
  ACTION revisar_contraoferta
  OUT    modificar
END
END
```

Figura 4.3.- Protocolo de negociación: papel del promotor/emisor

```
PROTOCOL RecepciónNegociación
STATES   esperar_confirmación
ACTIONS  evaluar_oferta, notificar_estado

FROM start TO esperar_confirmacion
  WHEN   proponer
  ACTION evaluar_oferta
# AGUAS!!! 'rechazar' implica el fin del protocolo
  OUT    aceptar, rechazar, contraproponer
END

FROM esperar_confirmación TO END
  WHEN   cancelar, confirmar
  ACTION notificar_estado
END

FROM esperar_confirmación TO esperar_confirmación
  WHEN   modificar
  ACTION evaluar_oferta
# Mismo comentario de antes:
  OUT    aceptar, rechazar, contraproponer
END
END
```

Figura 4.4.- Protocolo de negociación: papel del invitado/receptor

4.3 Resultados de la implementación

Los resultados obtenidos durante la implementación del sistema comprenden tanto la evaluación de su desempeño en uso como las líneas de desarrollo futuro que fueron identificadas a partir de la evaluación. Dichas líneas pueden verse posteriormente en un marco general, como se hace en la siguiente sección.

4.3.1 Situación actual

La implementación debe evaluarse desde los puntos de vista de sistemas (el estado alcanzado de desarrollo), y de validación de conceptos que lo sustentan.

Desde el punto de vista de sistemas, las rutinas de control de diálogos resultaron ser suficientemente generales para comunicar facilitadores en distintos ambientes y lenguajes de programación, debido a su diseño y a que su código fuente es transportable. Además, el diseño de objetos empleado permitió definir un buen mecanismo de extensibilidad para incorporar nuevos protocolos de transporte.

Como validación de las ideas del modelo comunicativo del sistema, la experiencia de la implementación del sistema de control de diálogos, sus pruebas, y el desarrollo de la aplicación de negociación de reuniones, permitieron obtener las siguientes conclusiones:

- Los diálogos son un buen mecanismo para estructurar la comunicación, pero como siempre es necesario interpretar los mensajes, se requiere una relación cercana entre **Diálogo** y **Acción**.
- Dado que los facilitadores pueden tener distintos estilos de programación y uso de concurrencia, por portabilidad está bien aislar en la capa **Acción** los detalles relativos al control del facilitador. Pero dadas la necesidad de implementar una capa de acción para cada tipo de facilitador y la relación cercana entre diálogos y acciones, también es necesario contar con una arquitectura o mecanismo que sirva de interfaz entre ellos.
- Los protocolos dinámicos pueden ser útiles para definir e implementar interacciones estándares en el sistema de agentes. Su descripción puede emplearse para revisar el comportamiento comunicativo del agente, pero tiene la desventaja de ser poco expresiva. No hay forma de asegurar que dos implementaciones de la misma acción en diferentes facilitadores tengan la misma semántica, y frecuentemente uno quiere describir más cosas sobre el protocolo que está programando: declarar un grupo de participantes, indicar qué mensaje se envía a cada quién en qué momento, y cuál es el significado o intención de ese mensaje en términos de pre- y postcondiciones.

4.3.2 Trabajo futuro

La continuación natural del trabajo sería refinar el código actual y portarlo a otros ambientes (como pueden ser los controles OLE en Windows), para seguir probando su capacidad de heterogeneidad, validando los conceptos que lo sustentan, e identificando otros mecanismos y conceptos que deban considerarse en la arquitectura. Además, destacan las siguientes posibilidades:

- Dedicar atención a la parte de **Acción**, no tratándola ya independientemente de la aspecto comunicativo, y definir una interfaz nueva entre esa capa y **Diálogo**.
  En este sentido, puede seguirse trabajando con el catálogo de plantillas parametrizadas o adoptar un concepto semejante al de adaptador de objetos en CORBA (ver sección 1.3 y [OMG, 1996]).
- Continuar desarrollando el catálogo de plantillas parametrizadas, no sólo para expresar aspectos genéricos en la interfaz de capas, sino también para descubrir similitudes y correspondencias entre los patrones de interacción de las partes estáticas de un sistema (módulos y procedimientos) y las interacciones de las partes dinámicas, lo que llevaría a desarrollar un enfoque cooperativo de la programación.
  Además, cuando las nuevas características de plantillas que propone el estándar de C++ (parámetros por default, métodos parametrizados, etc.) estén mejor soportadas, pueden aprovecharse en el catálogo, lo que lo haría más fácil de usar.
- Mejorar las descripciones de protocolos para aprovechar mejor su disponibilidad en tiempo de ejecución. Si las descripciones permitieran saber qué es lo que debe hacer un procedimiento o predicado asociado a un identificador, almacenando información respecto al significado de las condiciones mentales y las acciones, podría elaborarse un mecanismo de asociación dinámica entre identificadores y acciones para que los agentes que sepan emplear esa información puedan aprovecharla.
  El trabajo en este sentido permitiría no sólo programar *scripts* de facilitadores, sino en algunos casos, tener agentes que aprendan a utilizar protocolos nuevos.

4.4 Reflexiones

Partiendo de las conclusiones de la implementación, se pueden encontrar líneas más generales para la continuación del trabajo en el futuro. En este sentido, es importante revisar y aprovechar otros desarrollos que puedan ser útiles.

4.4.1 Relevancia de la interacción

Visto el trabajo en un sentido general, durante su realización destacó la relevancia de las cuestiones de interacción de los componentes de un sistema, en todos los niveles: desde subprogramas y objetos en el nivel local, hasta objetos y agentes en el medio distribuido. Se identificaron analogías y patrones recurrentes de interacción, desde el

análisis de protocolos entre participantes en DDE y KQML, hasta la clasificación de interacciones entre **Acción** y las demás capas que se empleó para el catálogo de plantillas de máquinas de acción.

Los aspectos de la interacción en medios estáticos y dinámicos, que se hicieron más evidentes por las condiciones de heterogeneidad supuestas para el ambiente donde se explotaría el sistema, son secundarias al objetivo del trabajo (la comunicación en el ambiente heterogéneo), pero definitivamente importantes dada la relación estrecha entre comunicación y acción. El sistema de diálogos actual necesita más estructuración y trabajo para proveer una mejor integración entre **Diálogo** y **Acción**, y puesto que ya hay desarrollos que proveen servicios en ese sentido (los sistemas de objetos distribuidos), lo más natural es emplearlos.

Para tal efecto, lo que se requiere es definir primero una equivalencia o mapeo entre el modelo de objetos y los mensajes basados en actos de habla, para así tener un marco más completo para la programación de sistemas cooperativos comunicativos heterogéneos.

4.4.2 Líneas a seguir

Un sistema de objetos distribuidos favorece un modelo de trabajo en el que los programadores (y a veces los usuarios finales) toman los componentes que quieren o pueden emplear para crear documentos y componentes más elaborados; el énfasis está en el resultado final, no tanto en los componentes empleados. Los componentes se utilizan desde una aplicación contenedora (como Word o Excel en Windows) o un lenguaje de programación desde el cual se invocan sus servicios. En ambientes de este estilo aplican las siguientes observaciones:

- La programación se basa en el descubrimiento y utilización de las interfaces que implementan los objetos; por tanto, las interacciones posibles entre los componentes quedan determinadas por las interfaces que implementan y las que saben usar.
- Como consecuencia, las interacciones entre un contenedor y otros componentes quedan restringidas a las situaciones e interfaces previstas por sus diseñadores.
- Si se incorporan componentes con interfaces nuevas o incompatibles al ambiente, lo más que se puede hacer es tratar de extender las aplicaciones, los contenedores mediante lenguajes de programación (si los proveen), o desarrollar nuevas aplicaciones, quizá con un lenguaje de *scripts* que aprovechen las facilidades de invocación dinámica del sistema para emplear más rápidamente los nuevos objetos.

Esencialmente, mediante las interfaces se ha establecido un nivel de coordinación de funcionalidad y trabajo de los componentes, que es necesario mantener para que el sistema dé resultados. Desafortunadamente, el proceso de integración que implica se hace actualmente de manera centralizada: un humano (programador o usuario) decide qué

componentes emplea y dispone de ellos a su voluntad, dentro de las normas establecidas de cooperación.

Pero es posible y deseable hacer algo más. Como se señala en el modelo de programación por contratos [Meyer, 1988], un objeto que expone un método o interfaz de uso se compromete a que el resultado de una invocación cumpla determinadas condiciones, si los parámetros de entrada cumplen con otras; la comunicación entre objetos también está sujeta a reglas.

Desde el punto de vista de los modelos de comunicación vistos en el capítulo 2, puede afirmarse que el conocimiento de las interfaces (métodos, significado, parámetros y orden de invocación) representa el conocimiento compartido, las reglas que hacen posible la comunicación entre los objetos. Un objeto que expone una interfaz de uso está esencialmente ofreciendo mantener un diálogo. Averiguar si un objeto soporta determinada interfaz es averiguar si va a cumplir las convenciones asociadas a la interfaz.

Continuando con ese punto de vista, cuando se envía un mensaje a un objeto, la intención y ontología están implícitas en el significado asociado al método que se invoca, y al alterarse el estado del objeto como resultado de la ejecución del método, cambia su "estado mental". Para facilitar una expresión más amplia de la cooperación, es necesario hacer explícitas las intenciones y las acciones asociadas a ellas. Si se clasificaran las intenciones posibles en la comunicación, separándolas del contenido de los mensajes, se podrían distinguir estados mentales y protocolos de interacción; se podría distinguir entre la forma de uso de la interfaz de un objeto (cómo se debe interactuar con ella) y el objetivo concreto de la interfaz. Almacenar esta información ayudaría a establecer patrones de interacción como los que se reconocen en ingeniería de software [Coad, 1995], y a aprovechar ese conocimiento para repetir ese tipo de interacción con otros objetos, probablemente en tiempo de ejecución.

Conviene aclarar que la asociación de cualidades mentales a los sistemas no implica que éstos tengan que programarse declarativamente o con un sistema basado en conocimiento: el modelo de los interlocutores del agente puede mantenerse mediante una representación de autómatas, o por el mismo programador, que es lo que en realidad hacemos al programar clientes y servidores tradicionales[1]. Realizando mayor trabajo en la parte del diseño, puede establecerse un esquema de comunicación basada en intenciones en un ambiente de objetos distribuidos, sin requerir que todos los participantes mantengan explícitamente modelos mentales. Puede procederse de la siguiente manera:

---

[1] El problema en este último caso es el mantenimiento de la coherencia entre las interpretaciones de estados mentales que hagan los sistemas participantes

1. Partir de una clasificación de las intenciones e interacciones asociadas, que sea suficientemente amplia para permitir la participación de los componentes interesantes (algunos sistemas de comunicación basados en actos de habla, como [Cohen *et al*, 1994], dan mayor peso al lenguaje del contenido y emplean un conjunto reducido de intenciones, por lo que los participantes requieren recursos mayores de cómputo para procesar, entender y reaccionar a los mensajes).
   Esta clasificación puede empezar con el trabajo desarrollado en el área de patrones [Coad, 1995].
2. Una vez realizada esa clasificación, las interfaces de los objetos se pueden diseñar en términos de intenciones agrupándolas, por ejemplo, como métodos en interfaces.
3. Pueden hacerse entonces componentes que empleen esos métodos para coordinar el trabajo de los demás objetos. Este es el esquema empleado en [OMG, 1997] y [Microsoft, 1997] para proporcionar el soporte de transacciones en CORBA y DCOM respectivamente, y en [Andreoli, 1996] para proporcionar soporte de coordinación y negociación en CORBA.
   Estos componentes pueden ser de mayor peso computacional, y mantener modelos mentales explícitos de los demás. Serían los que trabajaran en la coordinación del sistema.

En este momento es interesante retomar las experiencias del sistema de diálogos, para crear un marco en el que se pueda probar la interacción en un medio cooperativo heterogéneo; por ejemplo, se puede experimentar con:

1. Representaciones de la forma de uso de las interfaces (como el orden de invocación de sus métodos), para emplear y verificar su uso correcto de una interfaz.
2. La información necesaria para aprender a usar una interfaz o método en tiempo de ejecución, lo que ayudaría a diseñar agentes que busquen formas alternativas de realizar tareas, y a encontrar nuevos patrones de interacción de más alto nivel.

Este nuevo marco podría probarse en un ambiente como Windows. Las interfaces COM para acceso a datos que Microsoft ha definido recientemente (Universal Data Access, Ole DB, Active Data Objects) pueden proporcionar un buen material para detectar patrones de interacciones, probar representaciones de forma de uso, y coordinar trabajos.

4.4.3 Sobre el uso de intenciones y modelos mentales

Reconociendo explícitamente la intención, los sistemas toman una dimensión totalmente diferente: pueden ofrecer sus servicios a quien lo requiere (ya sea usuario humano o computacional), pues tienen mejor conocimiento respecto a lo que pretenden los demás y pueden explotar de mejor manera la información disponible dinámicamente. Pero hay que tomar en cuenta que faltaría argumentar sobre la validez del uso de las intenciones y estados mentales para un sistema abierto heterogéneo, donde los recursos de

cómputo disponibles, granularidades y medios de implementación de los participantes pueden ser muy distintos.

Cuando hablamos acerca de las cosas, y especialmente cuando no nos son familiares, es muy común asociarles cualidades mentales tales como creencias e intenciones; entendemos claramente cuando nos dan una explicación como "ahora la computadora quiere que le digas dónde debe guardar tu archivo", aún cuando la frase no sea válida en un sentido estricto. Este enfoque animista manifiesta en realidad nuestras capacidades de abstracción, es otra de las formas mediante las cuales manejamos la complejidad, y podemos aplicarla a cualquier objeto con el que interactuamos, como lo muestra el ejemplo del interruptor de [Shoham, 1990]: es posible tratar a un interruptor de luz como un agente cooperativo que tiene la capacidad de transmitir corriente a voluntad, cosa que hace sólo cuando cree que queremos que sea transmitida; mover el apagador es sólo nuestra manera de comunicarle nuestros deseos.

Precisamente por su generalidad, la perspectiva animista no siempre es deseable: en varios casos (como el ejemplo del interruptor), entendemos suficientemente los objetos como para tener descripciones más simples de su funcionamiento. Entonces ¿cuándo hay que asociar estados mentales a los objetos? McCarthy (citado en [Shoham, 1990]) dice:

> Atribuir ciertas creencias, libre albedrío, intenciones, conciencia, habilidades o deseos a una máquina o programa de computadora es **legítimo** cuando tal atribución expresa la misma información acerca de la máquina que la que expresa acerca de una persona. Es **útil** cuando la atribución nos ayuda a entender la estructura de la máquina, su conducta pasada o futura, o cómo repararla o mejorarla. Quizá nunca es **lógicamente requerido**, incluso para las personas... La atribución de cualidades mentales es **más sencilla** para las máquinas de estructura conocida como los termostatos y sistemas operativos de computadoras, pero es **más útil** cuando se aplica a entidades cuya estructura es casi completamente desconocida.

Bajo este punto de vista, cuando tenemos un ambiente abierto, distribuido y heterogéneo, en el cual los sistemas participantes pueden ser arbitrariamente simples o complejos, la atribución de estados mentales es especialmente útil: un sistema en la red no conoce gran cosa acerca de sus posibles interlocutores, así que le puede resultar muy redituable haber sido diseñado para pensar en ellos (y en sí mismo) en términos de modelos mentales.

Apéndice A:

Notación de objetos

La notación simplificada de objetos que se empleó en este trabajo se presenta en la figura A.5. Se complementa con la siguiente nomenclatura para estructuras de datos:

set<T>	Conjunto de objetos del tipo T.
dequeue<T>	Cola de objetos T, que permite acceso a todos los elementos.
dicc<T,U>	Diccionario: a un objeto T corresponde un objeto tipo U.

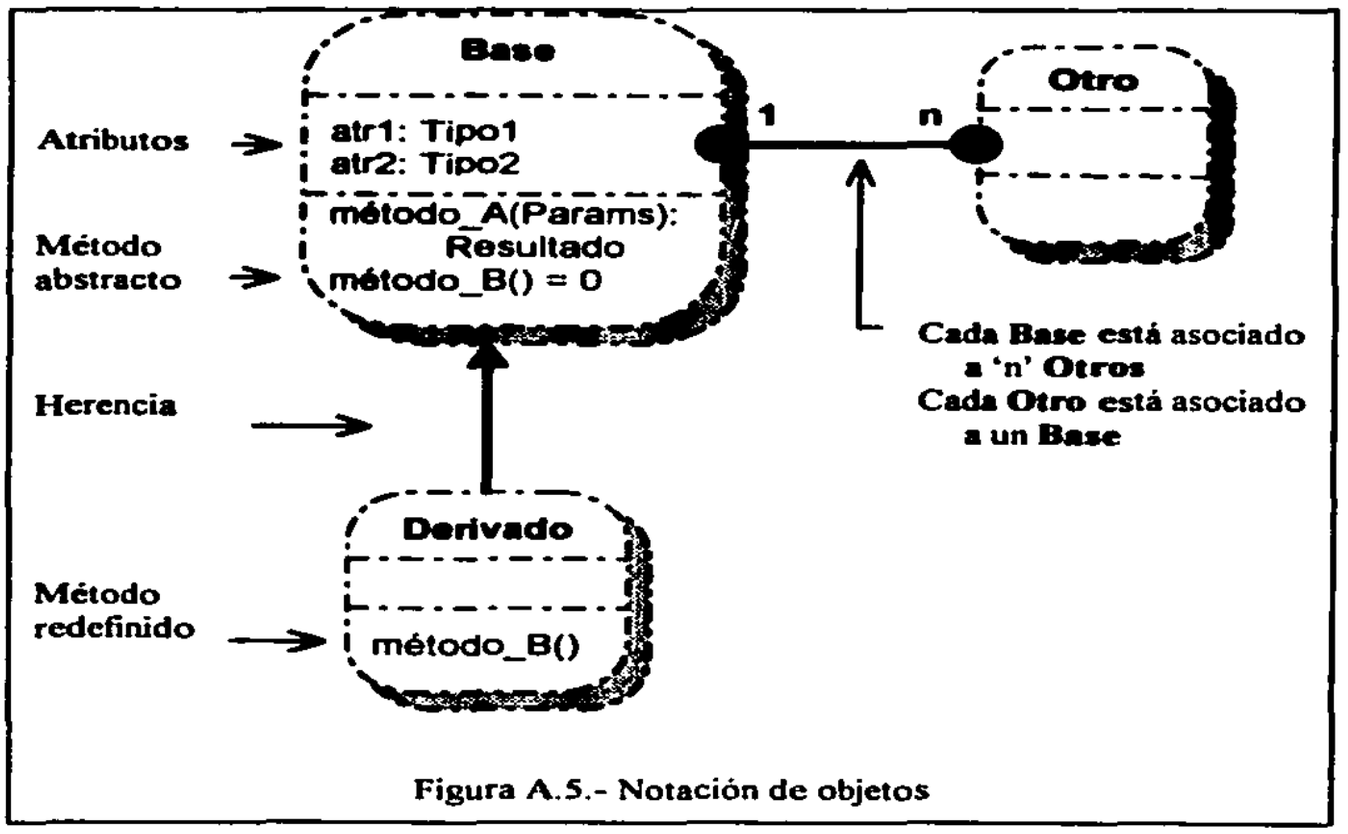

Figura A.5.- Notación de objetos

Bibliografía

[Allen, 1987]
James Allen
Natural Language Processing
Benjamin/Cummings. Menlo Park, CA, USA. 1987

[Andreoli *et al*, 1996]
Jean-Marc Andreoli; Steeve Freeman; Remo Pareschi
The Coordination Language Facility: coordination of distributed objects
Theory and Practice of Object Systems V2(2) 1996

[Austin, 1975]
James Austin
How to do things with words
2da. edición.
Harvard University Press. Cambridge, MA, USA. 1975.

[Barbuceanu y Fox, 1995]
Mihai Barbuceanu; Mark S. Fox
COOL: A Language for Describing Coordination in Multi Agent Systems.
Proceedings of the International Conference on Multi-Agent Systems
AAAI Press, San Francisco, CA, USA. Junio de 1995.
Disponible en:
`http://www.cs.umbc.edu/kqml/papers/kool.ps`

[Bond y Gasser, 1988]
Alan H. Bond; Les Gasser (eds.)
Readings in Distributed Artificial Intelligence
Morgan Kaufmann. San Mateo, CA, USA. 1988.

[Box, 1996]
Don Box
Introducing Distributed COM and the new OLE features in Windows NT 4.0
Microsoft Systems Journal V11(5). Mayo de 1996.

[Brockschmidt, 1994]
Kraig Brockschmidt
Inside OLE 2
Microsoft Press. Redmond, WA, USA. 1994.

[Carvajal *et al*, 1993]
Horacio Carvajal; Fabiola López; María G. López; Luis Zamora; Luis A. González
Sistemas Cooperativos: Un Marco de Referencia
Memorias de la X Reunión Nacional de Inteligencia Artificial. México, D.F. 1993.

[Chang y Woo, 1991]
Man Kit Chang; Carson C. Woo
SANP: A Communication level protocol for negotiations
En E. Werner, Y. Demazeau (Eds.). *Decentralized Artificial Intelligence 3.*
Proceedings of the 3rd. European Workshop on Modelling Autonomous Agents in a Multiagent World
Elsevier Science Publishers. Netherlands. 1991.

[Coad, 1995]
Peter Coad, con David North y Mark Mayfield
Object Models: Strategies, Patterns and Applications
Prentice-Hall, 1995
Catálogo de patrones disponible en
`http://www.oi.com/`

[Cohen *et al*, 1994]
Philip R. Cohen; Adam Cheyer; Michelle Wang; Soon Cheol Baeg
An Open Agent Architecture
Proceedings of the AAAI Spring Symposium Series on Software Agents
AAAI Press, San Francisco, CA, USA. Marzo 1994
Disponible en:
`ftp://ftp.ai.sri.com/pub/papers/oaa-aaai94.ps.gz`

[Cohen y Levesque, 1995]
Philip R. Cohen; Hector J. Levesque
Communicative Actions for Artificial Agents
Proceedings of the International Conference on Multi-Agent Systems
AAAI Press, San Francisco, CA, USA. Junio 1995.
Disponible en:
`ftp://cse.ogi.edu/pub/chcc/pcohen/our-kqml-unix.ps`

[Cohen y Perrault, 1988]
Philip R. Cohen; C. Raymond Perrault
Elements of a Plan-Based Theory of Speech Acts
pp. 169-186 en **[Bond]**

[Corbin, 1991]
John R. Corbin
The Art of Distributed Applications
En ser. Sun Technical Reference Library
Springer-Verlag. New York, NY, USA. 1991.

[Davies, 1994]
Winton Davies
AGENT/K: An Integration of AOP and KQML
Proceedings of the Third International Conference on Information and Knowledge Management (CIKM-1994)
Disponible en:
http://www.csd.abdn.ac.uk/wdavies/Publications/CIKM94/agentk.html

[Excelente, 1997]
Cora B. Excelente T.
Módulo de Planificación para Multiagentes
Tesis para obtener el grado de Maestra en Inteligencia Artificial
Universidad Veracruzana. Febrero de 1997

[Finin *et al*, 1993]
Tim Finin; Jay Weber; Gio Wiederhold; M. Genesereth; et al
Draft Specification of the KQML Agent-Communication Language
The ARPA Knowledge Sharing Initiative: External Interfaces Working Group.
Documento de trabajo de 1993.
Disponible en:
http://www.cs.umbc.edu/kqml/papers/kqml-spec.ps

[Finin *et al*, 1994]
Tim Finin; Richard Fritzson; Don McKay; Robin McEntire
KQML as an Agent Communication Language
The Proceedings of the Third International Conference on Information and Knowledge Management (CIKM'94), ACM Press. Noviembre de 1994.
Disponible en:
http://www.cs.umbc.edu/kqml/papers/kqml-acl.ps

[Genesereth *et al*, 1988]
Michael R. Genesereth; Matthew L. Ginsberg; Jeffrey S. Rosenschein
Cooperation without Communication
pp. 220-226 en **[Bond]**

[Genesereth y Ketchpel, 1994]
Michael R. Genesereth; Steven P. Ketchpel
Software Agents
Communications of the ACM V37(7). Julio de 1994.

[Grosz y Sidner, 1986]
Barbara Grosz; Candace Sidner
Attention, Intention, and the Structure of Discourse
Computational Linguistics, Vol. 12(3). 1986
También en Nagao (Ed.). Natural Language

[Grosz y Kraus, 1993]
Barbara Grosz; Sarit Kraus
Collaborative Plans for Group Activities
13th. International Joint Conference on Artificial Intelligence (IJCAI-93). 1993.

[Hewitt e Inman, 1991]
Carl Hewitt; Jeff Inman
DAI Betwixt and Between: From "Intelligent Agents" to Open Systems Science
IEEE Transactions on Systems, Man and Cybernetics. V21(6) Nov./Dic. 1991

[Jennings, 1994]
N. R. Jennings
The ARCHON system and its applications
2nd International Working Conference on Cooperating Knowledge Based Systems
Keele, UK. 1994
Disponible en:
`ftp://ftp.elec.qmw.ac.uk/pub/isag/distributed-ai/publications/CKBS94.ps.Z`

[Labrou, 1996]
Yannis Labrou
Semantics for an Agent Communication Language
Doctoral Dissertation
Computer Science and Electrical Engineering Department - University of Maryland Graduate School.
Baltimore, MA, USA. 1996
Disponible en:
`http://www.cs.umbc.edu/~jklabrou/thesis.html`

[Lemaître *et al*, 1993]
Christian Lemaître; Víctor G. Sánchez; Cora B. Excelente; Luis Zamora
Red de Cooperación para Sistemas Expertos
Memorias de la X Reunión Nacional de Inteligencia Artificial. México, D.F. 1993.

[Lemaître y Excelente, 1997]
Christian Lemaître; Cora B. Excelente
Multi-Agent Network for Cooperative Work
Por publicar en Expert Systems with Applications Vol. 14(1-2). Enero de 1998.

[Referencias de SHARE]
Mayor información disponible en:
`http://gummo.stanford.edu/html/SHARE/share.html`
`http://www.eit.com/creations/research/share/share/share/share-home.html`

[Mayfield *et al*, 1995]
James Mayfield; Yannis Labrou; Tim Finin
Desiderata for Agent Communication Languages
Proceedings of the AAAI Symposium on Information Gathering from Heterogeneous, Distributed Environments
AAAI-95 Spring Symposium, Stanford University, Stanford, CA.
Disponible en:
`http://www.cs.umbc.edu/kqml/papers/desiderata-acl.ps`

[Meyer, 1988]
Bertrand Meyer
Object-Oriented Software Construction
Prentice-Hall. New York, NY, USA. 1988

[Microsoft, 1992]
Microsoft Corporation
Microsoft Windows 3.1 Programmer's Reference Guide. Vol. 2: Functions; Vol. 3: Messages, Structures and Macros
Microsoft Press. Redmond, WA, USA. 1992.

[Microsoft, 1996]
Microsoft Corporation
Microsoft Transaction Server
Información disponible en:
`http://www.microsoft.com/msdn/trans.html`

[OMG, 1996]
Object Management Group
The Common Object Request Broker: Architecture and Specification
Revisión 2.0 (julio de 1995). Actualizado en julio de 1996
Disponible en:
`ftp://ftp.omg.org/pub/docs/ptc/96-08-04.pdf`
Mayores referencias en:
`http://www.omg.org`

[OMG, 1997]
Object Management Group
CORBA services: Common Object Services Specification - Transactions Service
Edición revisada (marzo de 1995). Actualizada en julio de 1997
Disponible en:
`ftp://ftp.omg.org/pub/docs/formal/97-07-19.pdf`

[Ousterhout, 1994]
John Ousterhout
Tcl and the Tk Toolkit
Addison-Wesley. 1994.
Mayores referencias en:
`http://sunscript.sun.com`
`http://www.sunlabs.com/~ouster/`
`http://www.sunlabs.com/research/tcl/`

[Palacios]
Amalia Palacios R.
Tesis para obtener el grado de Maestra en Ciencias de la Computación
UACP y P del CCH - UNAM. Por presentar.

[Reed *et al*, 1997]
Dave Reed; Tracey Trewin; Mai-lan Tomsen
Microsoft Transaction Server helps you write scaleable, distributed Internet apps
Microsoft Systems Journal V12(8). Agosto de 1997.

[Rich, 1991]
Elaine Rich; Kevin Knight.
Artificial Intelligence
2da. edición. McGraw-Hill. 1991

[Ritchie, 1984]
Dennis M. Ritchie
A Stream Input-Output System
AT&T Bell Laboratories Technical Journal V63(8) Oct.1984 pp. 1897-1910
Disponible en;
`http://plan9.bell-labs.com/cm/cs/who/dmr/index.html`

[Rodríguez, 1985]
Emir Rodríguez Monegal (ed.)
Jorge Luis Borges: Ficcionario. Una antología de sus textos
Col. Tierra Firme. Fondo de Cultura Económica. México, D.F. 1985

[Sánchez, 1995]
Víctor G. Sánchez
Programación cooperativa ¿Un nuevo paradigma?
Soluciones Avanzadas 4(25). Septiembre de 1995.

[Searle, 1969]
John Searle
Actos de Habla Ensayo de filosofía del lenguaje
Planeta-Agostini. Madrid, España. 1994

[Shoham, 1990]
Yoav Shoham
Agent-Oriented Programming
Stanford University Technical Report CS-1335-90

[Sidner, 1994]
Candace Sidner
An Artificial Discourse Language for Collaborative Negotiation
Proceedings of the 12th. National Conference on Artificial Intelligence (AAAI-94).

[Weber y Kuokka, 1995]
Jay Weber; Dan Kuokka
KQML Application Programmer's Interface (KAPI)
Anteriormente disponible en:
`ftp://hitchhiker.space.lockheed.com/pub/aic/shade`
`/software/KAPI/`

[Welch, 1995]
Brent Welch
Practical Programming with Tcl and Tk
Prentice-Hall. 1995.

[Wielemaker, 1995]
Jan Wielemaker
SWI-Prolog 2.1.0
Mayores referencias en:
`http://swi.psy.uva.nl/projects/xpce/SWI-Prolog.html`

[Wooldridge y Jennings, 1994]
Michael J. Wooldridge; Nicholas R. Jennings
Agent Theories, Architectures, and Languages: A Survey
Proceedings ECAI-Workshop on Agent Theories, Architectures and Languages (eds. M.J. Wooldridge, N.R. Jennings)
Amsterdam, The Netherlands. 1994
Disponible en:
`ftp://ftp.elec.qmw.ac.uk/pub/isag/distributed-ai/publications/ECAI94-WS.ps.Z`

[Zachary y Robertson, 1989]
Wayne W. Zachary; Scott P. Robertson
Introduction
en Wayne W. Zachary; Scott P. Robertson; John P. Black (eds.)
Cognition, Computing and Cooperation
Ablex, Norwood, NJ, USA. 1989

[Zamora, 1996]
Luis Zamora C.
Un Sistema para el Control de la Cooperación entre Facilitadores
Tesis para obtener el grado de Maestro en Ciencias de la Computación
UACP y P del CCH - UNAM. Mayo de 1996.