# Una visión a los modelos actuales de soporte a transacciones distribuidas orientadas a objetos

*Camilo Villarreal Pinilla*[*]
*María Consuelo Franky*[**]

***Resumen:*** *Se presentan los cuatro modelos más conocidos para soportar manejo de transacciones distribuidas en ambientes orientados a objetos. Aunque del modelo FOAD se conoce en detalle su arquitectura, no existe, o por lo menos no es muy popular una versión implementada (i.e. API) de su metodología ni tampoco hay un estándar en cuanto a las interfaces que se proveen al programador para el desarrollo de aplicaciones que puedan manejar transacciones distribuidas. Por esto, únicamente se presentará una descripción de su teoría y la forma en que está diseñada su arquitectura. De OTS, JTS y MTS se presentará el diseño de sus arquitecturas, una breve descripción de los servicios que prestan sus interfaces y la forma en que actúan. La arquitectura de MTS no se conoce en detalle por razones comerciales, pero se ilustrará una aproximación basada en la documentación disponible. Es importante tener en cuenta que OTS y JTS son estándares (basados en la arquitectura de objetos distribuidos CORBA) propuestos para la implementación de productos tales como administradores de transacciones, mientras que MTS es un producto comercial.*

***Abstract:*** *We present the four most known models for supporting the management of distributed object-oriented transactions. Although FOAD model architecture is extensively known, there is no popular implementation of its methodology (i.e. API) neither a standard model that specifies interfaces provided to end user programmers for development of applications managing distributed transactions. That is why only a theory and design description will be presented. Conversely, we will present architecture design and a brief description of services offered by interfaces from OTS, JTS and MTS. The MTS architecture cannot be described in detail for commercial reasons but an approximated architecture design will be presented, based on available documentation. It is important to note that OTS and JTS are standards (based on the architecture of distributed objects model CORBA) proposed for the implementation of products such as transaction managers, while MTS is already a commercial product.*

* Ingeniero de Sistemas y Computación de la Universidad de los Andes, candidato al título de Magíster en Ingeniería de Sistemas y Computación de la misma universidad. Profesor-Investigador del Departamento de Ingeniería de Sistemas de la Pontificia Universidad Javeriana.

** Ingeniero de Sistemas y Computación, D.E.A. y Doctor de Tercer Ciclo en Informática de la Universidad de Lille I, Francia. Profesora - Investigadora del Departamento de Ingeniería de Sistemas y Computación de la Universidad de los Andes.

## 1. Introducción

El concepto de transacción, derivado del ambiente financiero, en el que un proceso requiere negociar la modificación de múltiples recursos con otros procesos manejadores de tales recursos verificando que todas las operaciones se realicen si todos los participantes acceden y manteniendo la integridad del sistema global, se ha extendido al universo de la programación distribuida orientada a objetos. Para que las aplicaciones no tengan que ser significativamente alteradas ni tengan que entenderse directamente con los componentes que intervienen en la ejecución de una transacción distribuida, se han desarrollado modelos, algunos ya implementados como un API de servicios, que permiten al usuario programador usar herramientas que facilitan la programación transaccional. De esta forma, el usuario se apoya en los servicios ofrecidos, que normalmente no dependen de la plataforma sobre la cual están distribuidos los componentes del sistema e implementa sus transacciones de forma transparente sin preocuparse del manejo a bajo nivel de los elementos necesarios para coordinar el desarrollo de la transacción.

El soporte transaccional que existía anteriormente, estaba pensado para controlar la atomicidad [2] de las operaciones y controlar el protocolo *two-phase commit* y no permitía trabajar con componentes heterogéneos o sobre ambientes distribuidos. Por ejemplo, *Java Data Base Connectivity* (JDBC) [6] provee un modelo para bloquear los recursos que van a ser usados en el contexto de una transacción y liberarlos en el momento en que se haga *commit*. Por defecto, JDBC inicialmente está en modo *autocommit* de tal manera que siempre que se ejecuta un procedimiento, se lleva a cabo un proceso *commit*. Sin embargo, JDBC no puede realizar un *commit* sobre conexiones distintas y por lo tanto, no permite desarrollar transacciones distribuidas.

## 2. La metodología FOAD

La extensión de la metodología FOAD (*Frame Object Analysis Diagrams*) [1] para el manejo de transacciones en ambientes distribuidos llamada *Transaction Object Management*, se puede considerar como precursora en cuanto al soporte de transacciones distribuidas, pues es el primer modelo ampliamente difundido que formuló una arquitectura en la que se pueden implementar aplicaciones distribuidas que soporten comportamiento transaccional en el paradigma de orientación a objetos y que sirve de base a nuevos servicios transaccionales como el esquema formulado en [7] y los modelos desarrollados por OMG, Java (Sun Microsystems) y Microsoft. La arquitectura FOAD es la más simple de las que se describen en éste artículo, pues sólo involucra tres tipos principales de objetos que se van a encargar del manejo transaccional y no hace referencia al manejo de componentes desarrollados sobre ambientes heterogéneos, aspecto que requiere que el servicio transaccional sea mucho más sofisticado y complejo, como se verá más adelante.

### 2.1. Descripción de la arquitectura FOAD

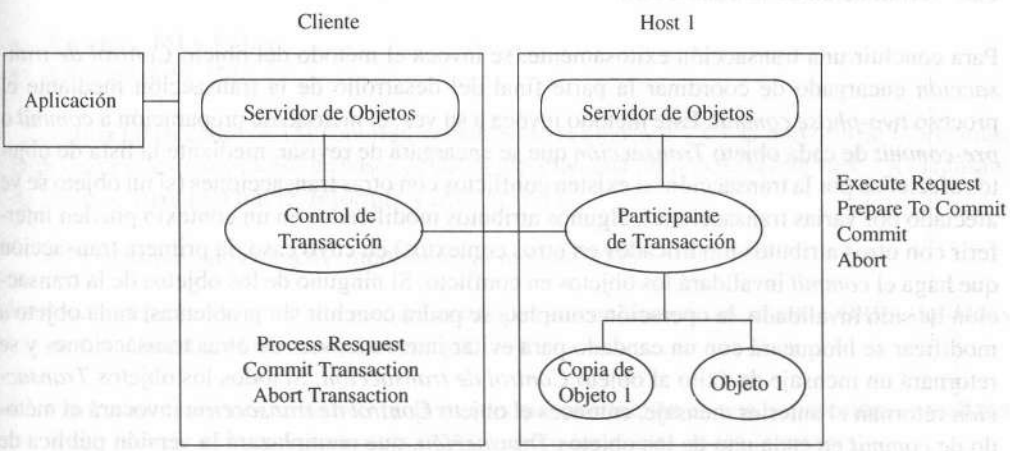
En FOAD se propone que *cada transacción sea vista como un objeto*, que incorpora el comportamiento necesario para se pueda realizar su ejecución en un esquema descentralizado (Figura 1). En cada sitio involucrado en la transacción, se tiene un *servidor de objetos* encarga-

do de recibir mensajes RPC y de manejar los objetos de su sitio, para que el usuario pueda manejar todos los recursos distribuidos en forma transparente. Además, se tienen dos tipos de objetos que son el objeto *Control de transacción* y el objeto *Transacción*; el primero se encarga de suministrar la interface a la aplicación cliente y de controlar el desarrollo de la transacción a nivel global y el segundo se encarga del procesamiento de requerimientos de la aplicación cliente que son parte de la transacción, además de la interacción con transacciones concurrentes. A través de la clase *Base*, se heredan los atributos y métodos que permiten que se puedan manejar los componentes de la transacción y que se mantenga la integridad sobre los recursos del sistema.

En los atributos del objeto *Control de transacción* se mantiene una lista de parejas *identificador de servidor de objetos/objeto Transacción* para todos los servidores de objetos que van a participar en la transacción, con el fin de poder referenciarlos durante el proceso. Además de los métodos de creación y destrucción, en este objeto se tiene un método para procesar los requerimientos cliente, uno para realizar el *commit* global y otro para abortar la transacción. -

Para evitar problemas de consistencia y concurrencia [2], se usan copias de los objetos que van a ser consultados y/o modificados durante el proceso y todas las operaciones que deban hacerse se harán sobre la copia del objeto para permitir que los cambios se hagan de forma aislada, de tal manera que cada objeto *Transacción* mantiene su propia copia del objeto en su respectivo sitio. En el momento en el que se realiza la operación *commit*, se remplaza el objeto original por la copia del objeto actualizado. En los atributos del objeto *Transacción* se encuentra la lista de las copias de los objetos involucrados en la transacción y en sus métodos, se tienen (aparte de los métodos estándar creación y destrucción, por supuesto) la operación que ejecuta los requerimientos cliente sobre las copias de los objetos y dos operaciones que implementan las etapas del protocolo *two-phase commit*, esto es, un método de *pre-commit* y el método que ejecuta el *commit* final. Adicionalmente, se tiene en este objeto un método para ejecutar localmente la operación de abortar la transacción, ya sea por conflictos con otras transacciones o por solicitud del usuario.

Figura 1. Esquema de la arquitectura FOAD.



El objeto *Control de transacción* representa un *transaction manager* o *manejador de transacciones* centralizado que permite ejecutar una transacción en dos fases. En la primera fase, se leen los objetos sobre los que se van a realizar operaciones y se anuncia la intención a los objetos encargados de manejar los objetos afectados. Para cada objeto que va a ser modificado, en el mismo sitio se crea una copia en donde se efectuarán los cambios a los que den lugar las operaciones de los requerimientos. Ya en la segunda fase, se reservan los objetos originales requeridos y se reemplazan por sus respectivas copias actualizadas que serán los nuevos objetos públicos. Si es necesario, se especifica un *timeout* o límite de tiempo máximo en el cual una transacción puede permanecer activa entre etapas, para permitir que otras transacciones puedan tener acceso a los objetos sobre los que se pretende realizar operaciones. Cuando se vence el tiempo límite se realiza una operación de *rollback* o reverso de la transacción que anula los efectos producidos.

### 2.2. Creación de la transacción

En el inicio de una transacción, el usuario genera un requerimiento RPC hacia el servidor de objetos local que entonces crea un objeto *Control de transacción* y retorna el identificador del nuevo objeto que recibirá los requerimientos posteriores (Figura 1). Un objeto *Transacción* es creado por el objeto *Control de transacción* en cada servidor que participará en la transacción y una lista de objetos *Transacción* será mantenida en el objeto *Control de transacción*, para poder referenciarlos en su tarea de coordinador. En el objeto *Control de transacción* se tiene un método de procesamiento de requerimientos cuyas funciones son: identificar el servidor de objetos del sitio en donde se encuentra el objeto especificado en el requerimiento, crear un objeto *Transacción* en ese sitio e invocar el método de ejecución en el objeto *Transacción* respectivo en el momento apropiado. Este último método de ejecución se encarga de crear una copia del objeto que va a ser afectado (si es la primera vez que el objeto es referenciado) a través de otro método heredado de la clase *Base*. Cada vez que se crea una copia de un objeto, se retorna un identificador que se anexa a la lista de copias que mantiene el objeto *Transacción* y se ejecuta la operación especificada en el requerimiento.

### 2.3. Terminación de la transacción

Para concluir una transacción exitosamente, se invoca el método del objeto *Control de transacción* encargado de coordinar la parte final del desarrollo de la transacción mediante el proceso *two-phase commit*. Este método invoca a su vez, el método de preparación a *commit* o *pre-commit* de cada objeto *Transacción* que se encargará de revisar, mediante la lista de objetos afectados por la transacción, si existen conflictos con otras transacciones (si un objeto se ve afectado por varias transacciones, algunos atributos modificados en un contexto pueden interferir con otros atributos modificados en otros contextos) en cuyo caso, la primera transacción que haga el *commit* invalidará los objetos en conflicto. Si ninguno de los objetos de la transacción ha sido invalidado, la operación completa se podrá concluir sin problemas, cada objeto a modificar se bloqueará con un candado para evitar intervenciones de otras transacciones y se retornará un mensaje de éxito al objeto *Control de transacción*. Si todos los objetos *Transacción* retornan el anterior mensaje, entonces el objeto *Control de transacción* invocará el método de *commit* en cada uno de los objetos *Transacción*, que reemplazará la versión pública de

cada uno de los objetos afectados por la copia en donde se realizaron las modificaciones y, una vez hecha esta operación, se retirarán los candados para permitir que nuevas actualizaciones se puedan hacer sobre los objetos. Por el contrario, si alguno de los objetos *Transacción* retornó un mensaje de error como resultado de la invocación del método de *pre-commit,* entonces el objeto *Control de transacción* invocará el método de abortar sobre todos los objetos *Transacción* para cancelar la operación global y destruir todas las copias de objetos involucrados. Este método también puede ser invocado directamente por el usuario al objeto *Control de transacción.*

## 3. La metodología OTS

La especificación OTS *(Object Transaction Service)* es un servicio transaccional de OMG que opera sobre el protocolo distribuido CORBA. [6] OTS brinda el soporte necesario para el procesamiento de transacciones distribuidas en las que el paradigma transaccional interviene sobre dos o más aplicaciones, procesos o *threads* que actúan sobre dos o más recursos independientes. Usa el concepto de *Transaction Manager* ó Manejador de Transacción, entidad externa que se encarga de proveer los medios necesarios para que una transacción pueda ser resuelta con más de una aplicación, proceso o máquina, manteniendo información en ejecución de los recursos afectados y coordinando el manejo mediante la comunicación con los recursos y el proceso que originó la transacción el cual ordena la acción definitiva: *commit* o *rollback.*

El estándar OTS se basa en el modelo definido por X/Open (consorcio compuesto por varias compañías que trabajan en el área, encargado de la definición de estándares de portabilidad para ambientes Unix) para el procesamiento de transacciones distribuidas llamado el DTP Reference Model [3] en el que se asegura la propiedad de atomicidad de sistemas transaccionales. En el modelo se identifican tres componentes principales en el ambiente de procesamiento transaccional distribuido:

- Application o aplicación (AP),
- Resource Manager o manejador de recursos (RM) y
- Transaction Manager o manejador de transacción (TM).

También se definieron las respectivas interfaces entre ellos:

- XA entre TM y RM, y
- TX entre AP y TM.

Varios fabricantes de TM soportan actualmente la interface TX y la mayoría de fabricantes importantes de manejadores de bases de datos soportan una versión de la interface XA (Transarc de Encina, Tuxedo, Oracle, Informix y SQL Server).

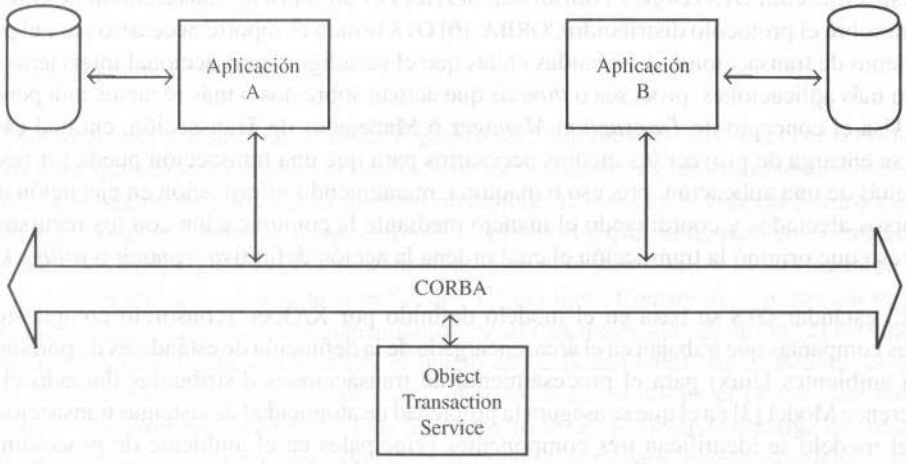
OTS incorpora dos tópicos importantes al modelo DTP:

- Las interfaces XA y TX son reemplazadas por un conjunto de interfaces CORBA definidas en IDL (Interface Definition Language).
- Las comunicaciones entre los componentes que intervienen en una transacción, se realizan como invocaciones a métodos vía CORBA sobre instancias de las interfaces.

### 3.1. Cómo funciona OTS

Para ilustrar la forma en que OTS interviene en la coordinación de una transacción distribuida, se presenta el caso de dos aplicaciones encargadas cada una de manejar su propio recurso —una base de datos—, que están distribuidas sobre CORBA (Figura 2). La aplicación A se encarga de originar la transacción y debe modificar su base de datos e invocar a la aplicación B, quien a su vez debe realizar una modificación sobre su base de datos y OTS se encarga de que todas las operaciones se realicen atómicamente.

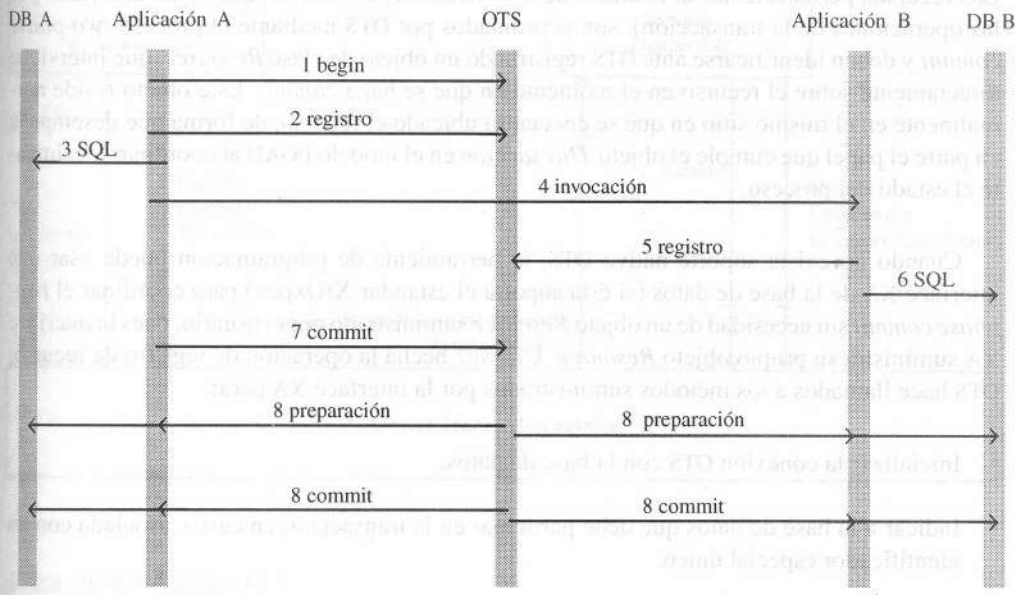
Figura 2. Interacción de dos aplicaciones sobre OTS.




En la Figura 3, se ilustran en orden los pasos que hay que llevar a cabo; las líneas verticales representan los componentes que intervienen en la transacción y las líneas horizontales representan las acciones realizadas entre componentes. La descripción del proceso global es la siguiente:

Inicialmente, la aplicación A hace un llamado CORBA a OTS y de ésta manera, la aplicación A entra en el *contexto de transacción* (paso 1). Enseguida, registra el recurso con el OTS (paso 2), informando que tiene una base de datos que debe ser modificada en el *contexto de transacción* y procede a actualizar su base de datos (paso 3), pero la actualización no se realiza inmediatamente sino que permanece en una lista de intenciones, pues OTS se encargará de indicar el momento en que se haga *commit*. La aplicación A debe ahora comunicarse con la aplicación B a través del ORB haciendo un llamado sobre un *objeto transaccional* (paso 4) en donde se informa que la transacción ya ha comenzado a realizarse sobre A. Cuando la aplicación B recibe el llamado, registra su recurso (paso 5) indicando que su base de datos debe ser modificada en el momento de completar la transacción y fabrica su lista de intenciones (paso 6) para que OTS haga efectiva la ejecución de la actualización en el momento del *commit*. El control pasa de nuevo a la aplicación A, origen de la transacción, que solicita a OTS que se complete el proceso invocando el método *commit* (paso 7). OTS se encarga de hacer commit directamente sobre las bases de datos usando el protocolo *two phase commit* (paso 8).

Figura 3. Evolución paso a paso de una transacción que involucra dos aplicaciones.




## 3.2. Algunos aspectos importantes de OTS

En el esquema OTS se han especificado una gran cantidad de interfaces que hacen que el proceso transaccional sea más complejo y más funcional. Para comprender la forma en que se desarrolla el proceso es necesario conocer el papel que juegan algunas de estas interfaces, porque los componentes del sistema pueden actuar de diferentes maneras dependiendo del uso que se haga de la funcionalidad de ellas. A nivel global, probablemente las dos operaciones más importantes del proceso transaccional son la *propagación del contexto y el registro de recursos*.

### *3.2.1. Propagación de la transacción*

Cuando la aplicación A realiza la invocación sobre la aplicación B, en realidad se hace una comunicación entre objetos, en donde A es el cliente y la aplicación B se visualiza como un objeto *servidor transaccional* y la invocación misma también se refiere a la información que A transfiere a B sobre el *contexto de la transacción*. Para realizar esta transferencia de *contexto de transacción*, en OTS existen dos métodos llamados explícito e implícito. En el método explícito es necesario incluir un parámetro especial en los métodos que van a efectuar modificaciones en la interface IDL del objeto servidor remoto (alternativa no deseable cuando hay gran cantidad de métodos) mientras que en el método implícito, el *contexto de la transacción* es transferido de forma casi transparente al objeto servidor mediante una simple operación: permitir que el objeto servidor herede de la clase de objeto transaccional.

### *3.2.2. Registro de recursos e interoperabilidad XA*

Los recursos pertenecientes al *contexto de transacción* (i.e. recursos que serán afectados por las operaciones de la transacción), son coordinados por OTS mediante el proceso *two-phase commit* y deben identificarse ante OTS registrando un objeto de clase *Resource*, que interviene directamente sobre el recurso en el momento en que se haga *commit*. Este objeto reside normalmente en el mismo sitio en que se encuentra ubicado el recurso, de forma que desempeña en parte el papel que cumple el objeto *Transacción* en el modelo FOAD al coordinar localmente el estado del proceso.

Cuando no existe soporte nativo OTS, la herramienta de programación puede usar una interface XA de la base de datos (si ésta soporta el estándar X/Oxpen) para coordinar el *two-phase commit* sin necesidad de un objeto *Resource* suministrado por el usuario, pues la interface XA suministra su propio objeto *Resource*. Una vez hecha la operación de registro de recurso, OTS hace llamados a los métodos suministrados por la interface XA para:

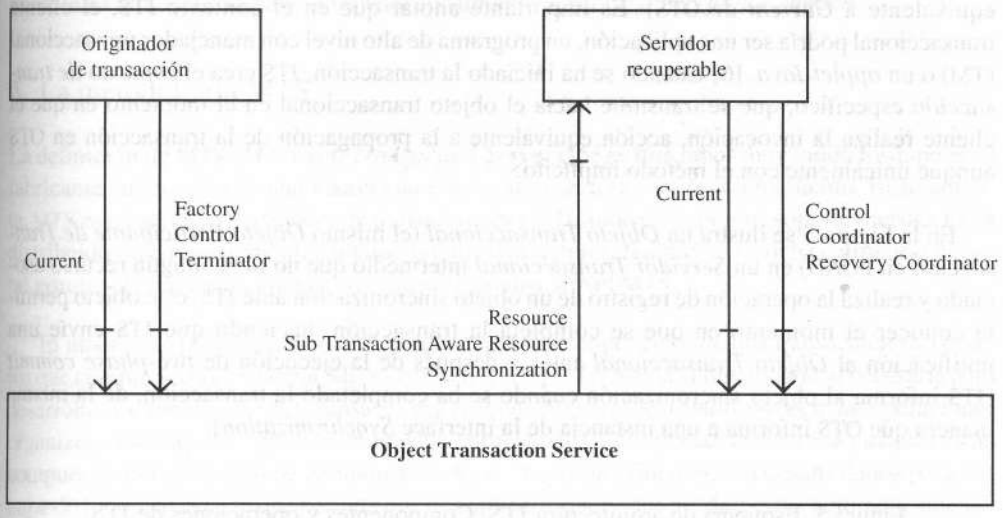
- Inicializar la conexión OTS con la base de datos.
- Indicar a la base de datos que debe participar en la transacción en curso, asociada con un identificador especial único.
- Indicar a la base de datos que finalice su participación en una transacción asociada con un identificador.
- Preparar la base de datos y emitir su estado de *pre-commit*.
- Realizar el *commit* de la transacción en curso sobre la base de datos.

A pesar de la gran ventaja que implica el uso de interface XA, existe un inconveniente significativo: la operación manual de registro XA de la base de datos debe ser realizada antes de que se reciban requerimientos; por lo tanto, toda base de datos que pueda verse afectada debe ser registrada inicialmente y OTS registrará un objeto *Recurso* para todas y cada una de ellas cada vez que se haga una invocación transaccional, implicando exceso de tráfico innecesario en los registros XA de las bases de datos que no intervienen en una transacción.

### *3.2.3. Resumen de interfaces OTS*

Las entidades participantes en una transacción en donde se usan las interfaces IDL definidas por OTS (Figura 4) son: el originador de transacción, encargado de iniciar y terminar la ejecución de la transacción y de invocar a los otros participantes, y el servidor recuperable, proceso que maneja objetos de recursos que deben ser modificados atómicamente. Las interfaces (definidas en un módulo llamado *CosTransactions*) y las entidades a las que corresponde usarlas, están ilustradas en la Figura 4.

Figura 4. Diagrama de componentes e interfaces en OTS.




## 4. La metodología JTS

La especificación JTS *(Java Transaction Service)* es un API sobre OTS para lenguaje Java [6] desarrollado para permitir que aplicaciones transaccionales Java puedan interactuar con otras aplicaciones, manejadores de recursos y manejadores transaccionales. En realidad, JTS es como una traducción de las interfaces OTS en Java (hecha mediante *mapping* usando un nuevo estándar llamado JavaIDL) que hace que una aplicación que maneje transacciones distribuidas en Java se comunique con componentes heterogéneos que pueden estar desarrollados en otro ambiente; de esta manera, se asegura que persigue los mismos objetivos que OTS, especialmente en cuanto a la heterogeneidad de transacciones distribuidas que son completadas usando el protocolo *two-phase commit*. Adicionalmente, los objetivos de interoperabilidad que provee JTS permiten que una aplicación transaccional Java pueda invocar manejadores de recursos como bases de datos o servidores transaccionales desarrollados en un ambiente distinto y que, asimismo, un servidor de este tipo pueda invocar una aplicación Java .

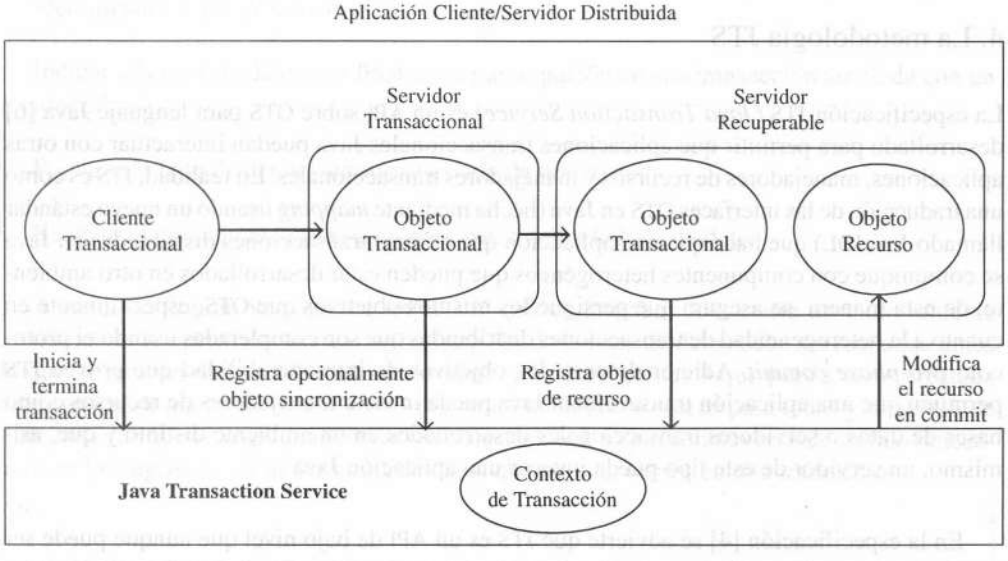
En la especificación [4] se advierte que JTS es un API de bajo nivel que aunque puede ser usado directamente, es ideal usarlo a través de una plataforma de más alto nivel, probablemente debido a su complejidad, para el desarrollo de aplicaciones transaccionales distribuidas. Como JTS está apoyado en OTS, también aprovecha las ventajas de éste último en cuanto a los esfuerzos que están realizando diferentes proveedores para adaptarse al estándar y asegura la interoperabilidad del ambiente Java en la parte transaccional con los diferentes componentes que adoptan la especificación OTS.

Como es de esperar, la arquitectura de JTS no varía casi nada respecto a la arquitectura OTS y las interfaces que se proveen son las mismas, aunque por supuesto, debido a las características del ambiente Java, su mecanismo varía internamente.

El cliente transaccional es el equivalente al originador de transacción en el diagrama de interfaces OTS y se encarga de iniciar y terminar la transacción (operaciones que se proveen en la interface equivalente a *Current* de OTS). Es importante anotar que en el contexto JTS, el cliente transaccional podría ser una aplicación, un programa de alto nivel con manejador transaccional (TM) o un *applet Java*. [6] Cuando se ha iniciado la transacción, JTS crea el *contexto de transacción* específico, que se transmite hacia el objeto transaccional en el momento en que el cliente realiza la invocación, acción equivalente a la propagación de la transacción en OTS aunque únicamente con el método implícito.

En la Figura 5 se ilustra un *Objeto Transaccional* (el mismo *Objeto Participante de Transacción* en FOAD) en un *Servidor Transaccional* intermedio que no tiene ningún recurso asociado y realiza la operación de registro de un objeto sincronización ante JTS; este objeto permite conocer el momento en que se completa la transacción, haciendo que JTS envíe una notificación al *Objeto Transaccional* antes y después de la ejecución de *two-phase commit* (JTS informa al objeto sincronización cuándo se ha completado la transacción, de la misma manera que OTS informa a una instancia de la interface *Synchronization*).

Figura 5. Esquema de arquitectura JTS. Componentes y operaciones de JTS.




Para poder manejar la persistencia, es necesario que el estado de los datos que van a ser afectados sea registrado: esta es la tarea del *Objeto Recuperable*, que registra ante JTS un *Objeto Recurso* a través del cual se maneja directamente el estado de los datos y se coordina el proceso *two-phase commit*. En este sentido, el *objeto recuperable* cumple la misma función del *objeto coordinador* de OTS, pues sobre este último se invoca un método para registrar manualmente un *objeto recurso* que controle los datos que van a ser afectados. El *objeto recuperable* almacena en memoria secundaria la información necesaria para completar localmente una transacción interrumpida por una falla del sistema, una vez se levante el sitio en el que participa. Durante la ejecución del proceso *two-phase commit*, el *objeto recuperable* periódi-

camente refleja el estado de los datos y cuando se restablece la operación en el sitio local después de una falla, el proceso de recuperación se basa en la información que el *objeto recuperable* ha almacenado para completar localmente el proceso *two-phase commit.*

## 5. La metodología MTS

La definición de MTS (*Microsoft Transaction Server*) [5] es una labor muy cuidadosa, pues su fabricante aún no provee una visión clara de su arquitectura y su funcionamiento. Básicamente, MTS es un servicio manejador de transacciones (TM) como OTS (provee soporte transaccional distribuido transparentemente) aunque según otros autores, realmente no debe calificarse como tal porque el código del cliente no realiza llamados al API MTS.

El modelo de programación MTS propone que los componentes de una transacción son objetos que cumplen con el estándar Microsoft ActiveX y proveen toda la funcionalidad necesaria para desarrollar localmente la tarea transaccional (hablando a nivel de usuario, es decir, las operaciones organizacionales que se ejecutan en un sitio participante). Las aplicaciones transaccionales están compuestas por conjuntos de componentes ActiveX que en principio, son desarrollados para trabajar de forma aislada, lo que permite que una aplicación servidor pueda escalar automáticamente.

Para que objetos componentes distribuidos puedan ofrecer servicios remotos y puedan ser efectivamente invocados, es necesario contar con un esquema distribuido que proporcione la arquitectura para integrar objetos en una aplicación. Así como en OTS el esquema de integración de objetos lo proporciona CORBA, en MTS también existe un modelo similar llamado DCOM (*Distributed Component Object Model*), desarrollado a partir de COM, que permite integrar objetos desarrollados por varios fabricantes que cumplen el estándar, pero en un solo sitio), aunque este último está compuesto de un subconjunto del universo de servicios incluidos en CORBA (por ejemplo, DCOM sólo permite integrar objetos entre aplicaciones desarrolladas sobre plataformas Microsoft).

Los principales elementos de la arquitectura MTS (Figura 6) y su papel fundamental son [5]:

- Los *Componentes ActiveX,* encargados de implementar la funcionalidad de la aplicación. Son objetos equivalentes a objetos CORBA que tienen su propia interface y pueden ser accesados remotamente, pero solo desde plataformas Microsoft que soporten DCOM. Cada componente es una aplicación que modela una parte de las actividades de la organización, conocidas como *business rules*. Las aplicaciones son implementadas sobre herramientas que soporten el estándar ActiveX [5] (por lo general son herramientas Microsoft como Visual Basic, Visual C++ o Visual J++) y deben ser compiladas como librerías de encadenamiento dinámico (DLL) para DCOM, lo que les permite funcionar como servidores que se activan cada vez que un servicio es solicitado. El estándar *ActiveX* provee las siguientes capacidades:
  - * Publicar interfaces (similares a las interfaces IDL) a través de las cuales se accede a los servicios ofrecidos por un componente. Un objeto puede ofrecer varias interfaces y permite que un cliente pueda consultar qué interfaces soporta.
  - * Comunicación entre componentes distribuidos, a través de DCOM.

* El mecanismo para que un objeto pueda ser identificado, cargado y ejecutado dinámicamente.

Por otra parte, MTS provee los mecanismos que permiten manejar el registro de servidores, la sincronización de recursos compartidos, la seguridad y consistencia y la operación equivalente a la propagación de contexto OTS, además del soporte al manejo transaccional (i.e. concurrencia, recuperación, atomicidad y aislamiento).

Cuando un componente se registra ante MTS (tarea que desempeña el usuario a través del *Transaction Explorer*), se debe especificar si soporta transacciones, no soporta transacciones o requiere una transacción. El API MTS debe ser usado en el código de los objetos componentes para poder acceder a la semántica transaccional; se usan básicamente los métodos que permiten reportar el resultado de ejecución de los componentes al coordinador (necesario para decidir si la operación local se completó o hay que hacer un *abort*). Como en JTS, las aplicaciones cliente pueden acceder a objetos componentes desde el *Web*, usando herramientas como *VBScript* y *Active Server Pages*, que permiten fabricar código embebido en páginas HTML desde las que se invocan los servicios ofrecidos.

- El *Transaction Server Executive*, agente que suministra los servicios *run-time* usados por los componentes. Este elemento, aunque no está claramente definido, juega un papel decisivo en el proceso transaccional porque se encarga de proveer los servicios *run-time* a los componentes, incluyendo el manejo de *threads* y de contexto transaccional. El *Transaction Server Executive* se define como una librería de encadenamiento dinámico (DLL) que se carga en los procesos servidores que corren los componentes de la aplicación remota.

- Los procesos servidores locales que corren las aplicaciones de cada sitio, es decir, el conjunto de componentes. Estos son procesos locales que se encargan de ejecutar los objetos componentes en cada sitio participante. Cada proceso administra un conjunto de componentes y puede atender a varios clientes (probablemente crea un *thread* para cada cliente en colaboración con el *Transaction Server Executive*), se encarga de manejar un dominio de aislamiento de fallas (lo cual implica que debe reflejar el estado local de cada transacción en memoria secundaria).

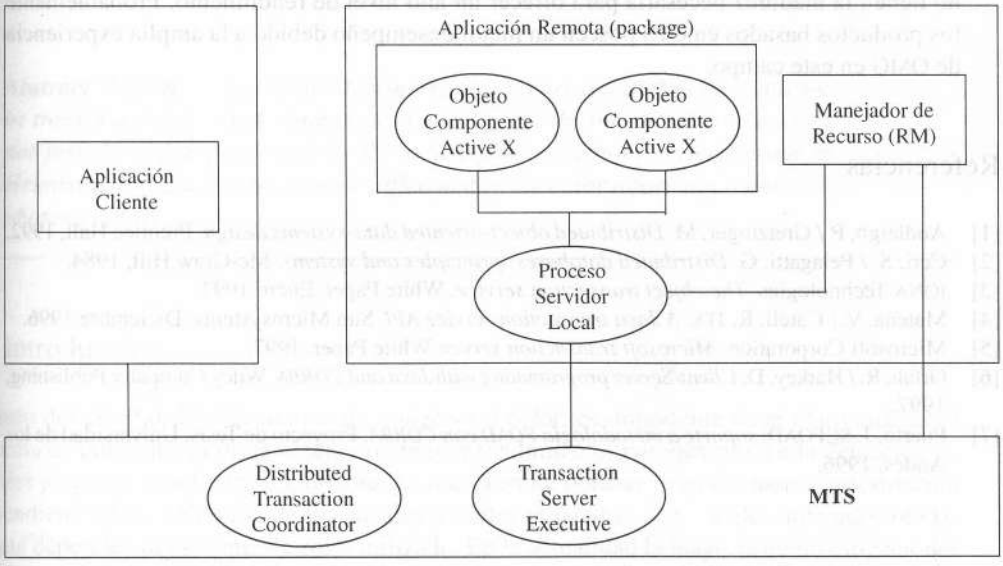
- Los manejadores de recursos encargados del manejo de persistencia como manejadores de bases de datos (en el mismo sentido de OTS y JTS). El manejador de recursos (equivalente al RM del modelo DTP de X/Open) es un sistema que permite manejar la persistencia en el proceso transaccional, es decir, permite que las acciones que se ejecutan sobre un recurso, sean durables. MTS soporta manejadores de recursos como bases de datos que sean compatibles con el protocolo *OLE Transactions* o con la interface XA del estándar X/Open. El papel que cumple el manejador de recurso es el mismo visto en los estándares OTS y JTS: es el último eslabón en la estructura del proceso transaccional, que emite su voto para que el coordinador determine la decisión de *commit* o *abort* general y trabaja en conjunto con el MTS para asegurar las propiedades ACID [2].

- Los *Dispensadores de recursos* que manejan información no persistente como variables de estado de recursos (por ejemplo, conjuntos de conexiones a bases de datos o *pooling*). Es un servicio que no tiene un equivalente directo en los manejadores de transaccciones vistos

anteriormente y que probablemente está diseñado para agilizar la ejecución del proceso, su tarea es mantener en memoria principal datos compartidos entre los componentes de un proceso. Un dispensador de recursos suministrado en MTS es el *ODBC Resource dispenser*, que maneja una lista de conexiones a bases de datos compatibles con el estándar *ODBC* para que puedan ser usadas eficientemente.

- El coordinador de transacción o *Distributed Transaction Coordinator* que controla todos los elementos participantes en una transacción, es decir, el equivalente al objeto *control de transacción* de la metodología FOAD. El coordinador de transacción es el mismo agente coordinador de FOAD que controla el proceso general a través del protocolo *two-phase commit* y coordina a los participantes, aunque no es claro si también es el encargado de crearlos.
- La interface gráfica de usuario *Transaction Server Explorer* que permite administrar el manejo de transacciones.

Figura 6. Esquema aproximado de la arquitectura MTS.
Elementos participantes en el proceso transaccional.




## 6. Conclusiones

El análisis de los modelos más conocidos para soportar el manejo de transacciones distribuidas en ambientes orientados a objetos permite obtener las siguientes conclusiones:

- Los modelos desarrollados para brindar soporte a transacciones distribuidas tienen una gran gama de funcionalidades que, por una parte, permiten desarrollar aplicaciones más

robustas pero al mismo tiempo las hace más complejas, implicando dedicación significativa a la comprensión de su manejo. Por ésta razón, *Sun Microsystems* recomienda que su JTS sea usado a través de un API de más alto nivel.

- Las implementaciones de servicios transaccionales que operan sobre CORBA, aún no permiten que se puedan desarrollar aplicaciones que usen tales servicios sobre *Internet* sin tener que montar en el sitio cliente el conjunto de herramientas propietarias necesarias para acceder al resto de componentes. Por ejemplo, la versión de OTS de Groupe Bull, OrbTP, consiste en un conjunto de servidores en donde se implementan los objetos que manejan la transacción y un conjunto de librerías que deben ser encadenadas con los clientes y servidores de aplicaciones transaccionales.

- Es una gran ventaja que el cliente transaccional pueda operar desde un *Web Browser*, funcionalidad hasta ahora ofrecida por JTS y MTS.

- La gran limitación de MTS es que solamente funciona sobre el estándar DCOM, lo cual impide que el gran universo de objetos transaccionales que residen en plataformas Unix no puede ser usado bajo este servicio.

- Finalmente, hay que tener en cuenta que estas tecnologías son de reciente desarrollo y aún no tienen la madurez necesaria para ofrecer un alto nivel de rendimiento. Probablemente los productos basados en OTS poseen un mejor desempeño debido a la amplia experiencia de OMG en este campo.

## Referencias

[1] Andleigh, P. / Gretzinger, M. *Distributed object-oriented data-systems design*. Prentice Hall, 1992.
[2] Ceri, S. / Pelagatti, G. *Distributed databases: principles and systems*. Mc-Graw Hill, 1984.
[3] IONA Technologies. *The object transaction service*. White Paper. Enero 1997.
[4] Matena, V. / Catell, R. JTS: *A Java transaction service API*. Sun Microsystems. Diciembre 1996.
[5] Microsoft Corporation. *Microsoft transaction server*. White Paper. 1997.
[6] Orfali, R. / Harkey, D. *Client/Server programming with Java and CORBA*. Wiley Computer Publishing, 1997.
[7] Puerto, J. SCFOAD, *soporte a metodología FOAD con CORBA*. Proyecto de Tesis. Universidad de los Andes. 1996.