Centro de Investigación y de Estudios Avanzados
del Instituto Politécnico Nacional

Departamento de Ingeniería Eléctrica

Sección de Computación

Orquestación de servicios Web orientada a aspectos

Tesis que presenta

Nancy Noemí Cova Suazo

para obtener el Grado de

Maestra en Ciencias

en la Especialidad de

Ingeniería Eléctrica

Director de la Tesis

Dr. José Oscar Olmedo Aguirre

México, D.F. **Octubre 2005**

"*Aférrate a la instrucción,*
no la dejes; guárdala,
porque ella es tu vida".

Proverbios 4:13.

A Dios, por caminar a mi lado en todo momento y brindarme su infinito amor.

A mi mamita y a mi padre, ella por ser la mujer que más admiro y amo en el mundo y él por ser el hombre que con sus cuidados y amor forjó la mujer que soy.

A mis hermanos, por ser los mejores amigos que jamás pude tener.

A mi niño, porque a lo largo de estos años he conocido el amor junto a él. Te amo :->.

Agradecimientos

Agradezco primeramente a Dios por llenar mi camino de bendiciones y pruebas, darme una familia tan bonita, regalarme a mis muñecas preciosas y ser mi todo en la vida; este sueño no lo podría haber logrado sin él.

A mis padres a quienes amo inmensamente, por su amor, sus desvelos y preocupaciones, porque son el motor de mi vida y me han regalo la herencia más hermosa e invaluable que se le puede dar a un hijo, la educación.

A mis hermanos Toño, Quelo, Malena, Anita, Silvia y Faby por ser mis mejores amigos, apoyarme incondicionalmente, aconsejarme y porque cada logro lo hacemos juntos.

A mi niño bonito Jesús porque nunca pensé que alguien pudiera hacerme tan feliz como él, porque cada vez que lo necesito está ahí para destraumarme, por hacerme reír, por su amor y todo su apoyo.

A mi hermanita Marisol que rápidamente llegó a tener un lugar muy especial en mi corazón, convirtiéndose en un integrante más de mi familia, por ser una niña entregada, dulce, comprensiva y porque siempre estuvimos al pie del cañón construyendo nuestros sueños.

Al Dr. Oscar Olmedo porque me brindó tiempo, paciencia, comprensión, dedicación y aliento para que finalmente pudiera ver cristalizado todo el trabajo depositado en esta tesis.

A Jonathán a quien no tengo palabras para agradecerle todo el cariño que me ha brindado desde que le conozco.

A los amigos que compartieron conmigo esta experiencia, en los cuales sólo encontré cariño y aliento, y a los que compartirán esta dicha por darle color a mi vida.

A mis sinodales, Dr. Luis Gerardo de la Fraga y Dr. Guillermo Morales, por revisar mi tesis, permitiéndome a través de sus valiosas correcciones obtener un mejor trabajo.

A Sofí por ser una personita tan linda conmigo, recibirme siempre con una sonrisa y por entregar el corazón día a día en todo lo que hace.

Al IPN porque ha forjado mi educación durante estos últimos 9 años, porque gracias a ella desperté a un mundo de conocimiento que atesoraré por toda la vida.

Al CINVESTAV por concederme la oportunidad de pertenecer a sus filas de estudiantes y contribuir en mi educación a lo largo de estos dos años.

A CONACyT por sustentar económicamente mis estudios de maestría, permitiéndome así culminar esta meta.

Al personal administrativo de servicios escolares y de la biblioteca de Ingeniería eléctrica por las facilidades otorgadas en cada trámite y consultas bibliográficas que realicé.

... a todos ellos sólo puedo decirles GRACIAS

Resumen

La orquestación de servicios Web es un área de gran interés debido a que ésta permite que aplicaciones de negocios heterogéneas puedan comunicarse, acoplarse y conjuntarse en una aplicación orquestada para obtener un mayor potencial en su funcionalidad.

Desafortunadamente el proceso de orquestación de servicios Web no es una tarea sencilla debido a que involucra la coordinación de aplicaciones distribuidas, donde la localización de los recursos es primordial. Aunque actualmente existen lenguajes y arquitecturas de coordinación que permiten llevar a cabo la orquestación de servicios, dichas propuestas presentan restricciones significativas en cuanto a flexibilidad, modularidad y adaptabilidad de componentes.

Por tal razón, esta tesis pretende brindar las facilidades necesarias para realizar el diseño conceptual y la especificación de un proceso de negocios como si éste fuera centralizado. Para lograrlo, la introducción de un nuevo lenguaje de coordinación llamado *Business Process Coordination Language (BPCL)* se justifica debido a que en los lenguajes actuales es indispensable especificar la ubicación de los servicios, es decir, no soportan transparencia de localidad de recursos. Además, el diseño e implementación de una infraestructura de coordinación que dé soporte al lenguaje resulta necesaria, ya que mediante ella podrán ser descubiertos e invocados los servicios Web que constituyan al proceso. Dichas operaciones serán encapsuladas mediante mecanismos de orientación a aspectos, haciendo posible la obtención de una aplicación orquestada de manera sencilla, dinámica y transparente a los desarrolladores.

Abstract

The Web services orchestration is an area of great interest because it allows that heterogeneous applications of business can communicate, compose and coordinate them in an orchestrated application to get greater potential in its functionality.

Unfortunately the process of orchestration of Web services is not an easy task because it involves the coordination of distributed applications, where the localization of the resources is fundamental. Although at the moment there are languages and architectures of coordination, which allow to carry out the orchestration of services, these approaches have significant restrictions about of flexibility, modularity and adaptability of components.

For such reason, this thesis seeks to offer the necessary facilities to carry out of the conceptual design and the specification of a business process like it were centralized. To achieve it, the introduction of a new language of coordination called Business Process Coordination Language (BPCL) is justified because in the current languages is indispensable to specify the localization of services, i. e., they do not support transparency of local resources. Moreover, the design and the implementation of a coordination infrastructure that supports the language is necessary, by means of this will be able to be discovered and invoked the Web services that constitute to the process. These operations are going to be encapsulated by aspect-oriented mechanisms, making possible the obtaining of an orchestrated application in a simple, dynamic and transparent way to the developers.

Índice general

Lista de Figuras xv

Lista de Tablas xviii

1. Introducción 1
- 1.1. Motivación . . . 1
- 1.2. Planteamiento del problema . . . 3
- 1.3. Propuesta . . . 5
  - 1.3.1. Objetivos del proyecto . . . 6
- 1.4. Contribuciones . . . 6
- 1.5. Organización de la tesis . . . 7

2. Antecedentes 9
- 2.1. Automatización de servicios en la Web . . . 9
- 2.2. Marco teórico . . . 10
  - 2.2.1. Servicios Web . . . 10
  - 2.2.2. Estándares WSDL, SOAP y UDDI . . . 12
  - 2.2.3. WebSphere SDK for Web Services . . . 16
  - 2.2.4. Orquestación de servicios Web . . . 18
  - 2.2.5. Programación orientada a aspectos . . . 20
- 2.3. Trabajo relacionado . . . 24
  - 2.3.1. Business Process Execution Language . . . 24
  - 2.3.2. BPEL for Java . . . 26
  - 2.3.3. Aspect-Oriented for BPEL . . . 26
  - 2.3.4. Dynamic Business Rules for WS Composition . . . 27
  - 2.3.5. Reflective Engineering for Web Services . . . 28
  - 2.3.6. Comparación con tecnologías existentes . . . 29
- 2.4. Sumario . . . 31

3. Business Process Coordination Language **33**
3.1. Descripción general . . . 33
3.2. Características . . . 34
3.3. Directivas de coordinación . . . 35
3.3.1. Etiquetas de datos . . . 36
3.3.2. Etiquetas de comunicación . . . 37
3.3.3. Etiquetas de coordinación . . . 38
3.4. Esquema del lenguaje . . . 40
3.5. Modelo de comunicación . . . 42
3.5.1. RPC . . . 44
3.5.2. Rendezvous . . . 46
3.6. Sumario . . . 49

4. Infraestructura de coordinación **51**
4.1. Descripción general . . . 51
4.2. Diseño . . . 52
4.2.1. Esquema conceptual . . . 52
4.2.2. Módulos de coordinación . . . 54
4.3. Implementación . . . 57
4.3.1. Traductor del lenguaje . . . 57
4.3.2. Búsqueda e invocación de un servicio Web . . . 64
4.3.3. Publicación de un servicio Web orquestado . . . 66
4.4. Caso de estudio . . . 68
4.5. Sumario . . . 76

5. Conclusiones **77**
5.1. Contribuciones del trabajo . . . 77
5.2. Trabajo futuro . . . 79

A. XML Scheme para procesos BPCL **81**

B. Documentación de la biblioteca de coordinación BPCL.jar **85**
B.1. Etapa de Coordinación . . . 85
B.2. Etapa de Traducción . . . 92
B.3. Etapa de Invocación . . . 97

C. Archivos de implementación para el caso de estudio **101**
C.1. Servicio Web (Web Service) 102
C.2. Servidor RMI (RMI Server) 102
C.3. Proveedores de servicios (Suppliers) 107
C.3.1. Store 107
C.3.2. Blackboard 109
C.4. Cliente (Client) 111
C.5. Generación del servicio Web orquestado 115
C.5.1. Descripción del servicio Web 116
C.5.2. Archivo de ejecución 118

D. Descripción del CD que acompaña a la tesis **121**

E. Glosario **125**

Bibliografía **131**

Índice de figuras

1.1. Entidades involucradas en el proceso de orquestación de servicios Web. 4

2.1. Arquitectura de servicios Web. 11
2.2. Ejemplo de una descripción WSDL. 14
2.3. Ejemplo de un mensaje SOAP de petición. 15
2.4. Ejemplo de un mensaje SOAP de respuesta. 15
2.5. Orquestación de servicios Web. 19
2.6. Contraste entre un esquema de programación tradicional y uno orientado a aspectos. 22
2.7. Ejemplo de un *aspecto* escrito en AspectJ. 23

3.1. Tipos de actividades en BPCL. 35
3.2. Encabezado típico para un proceso BPCL. 40
3.3. Fragmento del esquema para BPCL. 41
3.4. Modelo de comunicación RPC. 45
3.5. (a) Descripción de un proceso cliente y un (b) proceso servidor en BPCL utilizando el modelo RPC. 46
3.6. Modelo de comunicación *Rendezvous*. 47
3.7. (a) Descripción de un proceso cliente y un (b) proceso servidor en BPCL utilizando el modelo *Rendezvous*. 48
3.8. Generalización de la primitiva *accept* en BPCL utilizando el modelo *Rendezvous*. 49

4.1. Esquema tradicional de comunicación cliente-servidor. 52
4.2. Esquema conceptual para la búsqueda de servicios Web. 53
4.3. Esquema conceptual para la invocación de servicios Web. 53
4.4. Módulos principales para la publicación de un servicio Web orquestado. 55
4.5. Proceso de búsqueda e invocación de un servicio Web. 56

4.6. Composición de actividades. 57
4.7. Algoritmo de traducción para obtener la implementación en Java del flujo de actividades de una especificación BPCL. 61
4.8. Fragmento del esquema de traducción BPCL a Java. 62
4.9. Clase Wrapper asociada a un servicio Web. 64
4.10. Aspecto asociado a una clase *Wrapper*. 66
4.11. Proceso de publicación de un servicio Web orquestado. 67
4.12. Organización general de los participantes. 69
4.13. Diagrama UML de secuencia para el caso de estudio. 71
4.14. Especificación para la compra de artículos escrita en BPCL. 75

A.1. Esquema XML para procesos BPCL. 84

C.1. Listado del archivo BrokerBindingImpl. 102
C.2. Listado del archivo BrokerRMIServer. 103
C.3. Listado del archivo BrokerRMIInterface. 103
C.4. Listado del archivo BrokerRMIImpl. 107
C.5. Listado del archivo StoreWrapper. 107
C.6. Listado del archivo AspectStoreWrapper. 109
C.7. Listado del archivo BlackboardWrapper. 110
C.8. Listado del archivo AspectBlackboardWrapper. 111
C.9. Listado del archivo BrokerInit. 112
C.10. Listado del archivo BrokerWrapper. 113
C.11. Listado del archivo AspectBrokerWrapper. 115
C.12. Listado del archivo InvokeBroker. 115
C.13. Listado del archivo Broker.wsdl. 118
C.14. Listado del archivo Main. 119

Índice de tablas

2.1. Herramientas para el desarrollo de servicios Web en WSDK. 17
2.2. Sintaxis de un punto de corte en AspectJ. 22

3.1. Etiquetas de datos en BPCL. 36
3.2. Etiquetas de comunicación para atender peticiones en BPCL. 37
3.3. Etiquetas de comunicación para invocar servicios Web en BPCL. . . . 38
3.4. Etiquetas de coordinación en BPCL. 39

Capítulo 1

Introducción

El objetivo de este capítulo es presentar los aspectos que motivaron el desarrollo de nuestra propuesta de trabajo, cuál es la problemática que pretende atacar, en qué consiste la propuesta, qué objetivos persigue y cuáles son sus principales contribuciones. Hasta este momento sólo será presentado de manera general el proyecto desarrollado y en capítulos posteriores realizaremos un análisis detallado del mismo.

1.1. Motivación

La ingeniería de software y en general el ámbito de la computación es una disciplina que está en constante evolución. Cada día surgen nuevas técnicas y metodologías que intentan mejorar la calidad y la eficiencia de los productos de software [1]. El auge en el desarrollo de tecnologías Web, en particular los servicios Web, son un claro ejemplo del poderoso avance que los sistemas de información proveen en nuestros días.

El modelo de servicios Web representa un nuevo paradigma en el desarrollo de sistemas distribuidos que proveerá una plataforma para todas las transacciones de comercio electrónico (*e-commerce*) y de negocio a negocio (*Business to Business, B2B*), de forma tal que podamos tener acceso a un conjunto de servicios a través de la red de forma ágil y eficiente [2].

WSDL, SOAP y UDDI funcionan como especificaciones abiertas que facilitan el desarrollo de una colección de software basado en servicios Web. WSDL ofrece un lenguaje formal para definirlos [3], SOAP permite la interacción entre servicios heterogéneos [4], mientras que UDDI proporciona una amplia infraestructura que permite

al usuario publicarlos y descubrirlos [5]. Los servicios Web al ser desarrollados bajo estas tecnologías estándares tienen la capacidad de interactuar, cooperar entre sí en busca de un objetivo común a pesar de sus diferencias en cuanto a diseño, lenguajes de programación y/o plataformas de ejecución, ofreciendo así una solución simple y de bajo costo para resolver problemas de interoperabilidad e integración de aplicaciones. Algunos ejemplos típicos de servicios Web son: la compra de artículos por Internet, ejecución de transacciones bancarias, reservación de boletos, vuelos, hospedaje, entre muchos otros.

La tecnología de servicios Web además de incrementar el potencial de los sistemas de negocios al permitir la integración de aplicaciones distribuidas y heterogéneas, brinda un base sólida para que las aplicaciones puedan ser coordinadas, de tal manera que no sólo ejecuten sus propias actividades, sino que realicen transacciones más complejas. Al efectuar dicha coordinación se obtendrá como resultado un *servicio Web orquestado*, el cual estará constituido por otros servicios Web, mismos que posiblemente serán ofrecidos por diferentes organizaciones [6].

Es importante señalar la diferencia que existe entre algunos términos que con frecuencia son indebida e indistintamente utilizados: *composición, colaboración* y *coordinación de servicios*. Las técnicas de *composición* tienen como objetivo que los servicios comuniquen datos entre sí, la *colaboración* requiere de una serie de reglas que determinen cuándo y bajo qué condiciones tienen lugar esas comunicaciones, mientras que la *coordinación* es una tarea más compleja ya que ésta involucra la composición y la colaboración de servicios. Para llevarla a cabo deben ser especificados cada uno de los participantes del proceso de negocios, sus dependencias lógicas, cómo se comunicarán, las operaciones que deben ser invocadas, su orden y cómo será realizado el manejo de excepciones durante el flujo.

Actualmente existen diversos enfoques de trabajo que permiten llevar a cabo la coordinación u orquestación de servicios, BPEL4WS y BPEL4J representan algunos de ellos. BPEL4WS es un lenguaje estático composicional orientado al modelado de procesos orquestados [7], el cual cuenta con su propio motor de ejecución conocido como BPWS4J [8], mientras que BPEL4J extiende la funcionalidad de BPEL ofreciendo la pauta para que dos lenguajes de programación BPEL y Java puedan ser utilizados de manera conjunta, permitiendo así gozar de las ventajas que cada uno de ellos provee [9]. Lamentablemente, dichas propuestas presentan limitaciones significativas en cuanto a flexibilidad, modularidad y adaptabilidad de componentes, restando así dinamismo y claridad en el proceso de orquestación.

Dichas limitaciones pueden ser contrarrestadas gracias a la introducción de un reciente paradigma de programación denominado *programación orientada a aspectos (POA)* [1], el cual tiene como objetivo identificar y modularizar los conceptos que deben ser ejecutados por un sistema de negocios, separando los conceptos funcionales básicos de los conceptos adicionales, proporcionando además mecanismos que hagan posible abstraerlos y componerlos dinámicamente para conformar un sistema [1].

La siguiente lista presenta los principales beneficios obtenidos al desarrollar aplicaciones de negocios orientadas a aspectos.

Ventajas

- Una codificación menos enredada, más natural y más reducida.
- Mayor facilidad para razonar sobre las tareas, ya que éstas serán separadas y tendrán una dependencia mínima.
- Mayor facilidad para depurar y hacer modificaciones dinámicas.
- Un código reutilizable, que puede ser acoplado y desacoplado cuando sea necesario [1].

Finalmente, es importante puntualizar que la integración de *aplicaciones Web orientadas a aspectos* cada vez toma mayor fuerza debido a que dichas aplicaciones promueven mayor flexibilidad, dinamismo, adaptabilidad, reutilización de componentes y modularización al desarrollar sistemas de negocios.

1.2. Planteamiento del problema

El modelo de servicios Web es un enfoque práctico para integrar clientes, proveedores y aplicaciones de negocios, el cual se basa en una tecnología que permite acelerar el desarrollo de los sistemas Web distribuidos. Es por ésto que muchas empresas han empezado a adoptarlos y han intentado conectarlos y coordinarlos, de manera que al trabajar conjuntamente brinden un alto rendimiento a la organización. Desafortunadamente el proceso de orquestación de servicios Web no es una tarea sencilla debido

[1] Cristina Videira Lopes, Karl J. Lieberherr, Gregor Kiczales y su grupo de trabajo introdujeron el término de *programación orientada a aspectos* en 1995 [1].

a que involucra la coordinación de aplicaciones distribuidas.

Al desarrollar aplicaciones locales, el proceso de orquestación resulta relativamente sencillo puesto que las entidades y recursos involucrados estarán disponibles en una misma máquina. No obstante, cuando la orquestación deba ser realizada de manera distribuida, estos elementos residirán en distintas máquinas, por tal razón, será necesario considerar diversos aspectos como: el descubrimiento de recursos, mecanismos de control de concurrencia y sincronización, mecanismos de comunicación, de persistencia de datos, manejo de excepciones, compensación de transacciones, tolerancia a fallos, etc. En la figura (1.1) se muestran las entidades participantes en la orquestación de servicios Web.

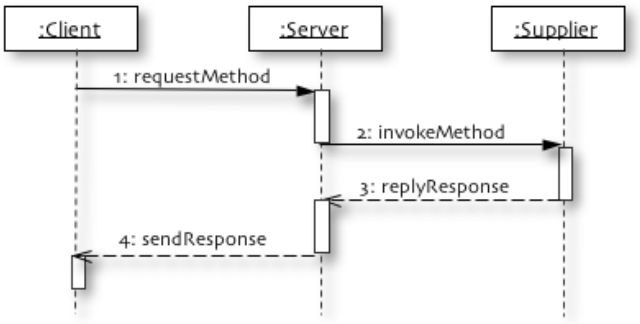

Figura 1.1: Entidades involucradas en el proceso de orquestación de servicios Web.

A pesar de que en la actualidad existen lenguajes y arquitecturas de coordinación (presentados a detalle en el capítulo 2) que permiten llevar a cabo la orquestación de servicios, dichas propuestas presentan restricciones significativas en cuanto a flexibilidad, modularidad y adaptabilidad de componentes. Llevan a cabo la orquestación desde una perspectiva poco dinámica y clara, ya que la localización de los recursos debe ser explícitamente especificada y la modularización de funcionalidades adicionales en un proceso de negocios no es soportada.

Otro gran inconveniente de los esquemas de coordinación actuales es que no permiten depurar o probar aplicaciones orquestadas en cada una de sus etapas de desarrollo cuando no existe una disponibilidad de recursos remotos. Por ejemplo, al presentarse una situación donde los servicios implicados no estuvieran disponibles o simplemente

una conexión en red no pudiera realizarse, el desarrollo o incluso la ejecución del proceso de negocios tendría que suspenderse, generando así pérdidas a la organización tanto económicas como en tiempo.

Debido a las limitaciones antes expuestas, es necesario proponer nuevas metodologías, enfoques y tecnologías que simplifiquen y faciliten el desarrollo y la depuración de aplicaciones orquestadas.

Por tal razón, en nuestra propuesta de trabajo el problema al cual nos enfrentamos es el de brindar mecanismos que permitan reducir la complejidad del proceso de orquestación de servicios Web, haciendo sencillo y claro tanto el diseño como la implementación de aplicaciones orquestadas.

1.3. Propuesta

La propuesta de trabajo que se presenta en esta tesis pretende brindar las facilidades necesarias para realizar el diseño conceptual y la especificación de un proceso de negocios como si éste fuera centralizado y a través de los servicios Web y la POA sea posible obtener una aplicación distribuida. Con lo anterior se conseguirá que el desarrollador de dichas aplicaciones no deba preocuparse por especificar la localización de los recursos involucrados sino que éste sólo especifique las actividades implicadas en el flujo y su orden válido de ejecución. Y además que el desarrollo, la depuración y la ejecución del proceso orquestado pueda realizarse sin ser forzosa una conexión en red.

Para lograrlo, la introducción de un nuevo lenguaje de coordinación llamado *Business Process Coordination Language (BPCL)* se justifica debido a que en los lenguajes actuales de coordinación resulta necesario especificar la ubicación de los servicios Web al momento de especificar el proceso de orquestación de los mismos, es decir, no soportan transparencia de localidad de recursos (servicios). Además, el diseño e implementación de una infraestructura de coordinación que dé soporte al lenguaje resulta necesaria, ya que mediante ella podrán ser descubiertos e invocados los servicios Web elementales que constituyan al proceso. Cabe señalar que dichas operaciones serán encapsuladas mediante mecanismos de orientación a aspectos. Así, obtendremos como resultado una aplicación orquestada y distribuida de manera sencilla, dinámica y transparente a los desarrolladores.

Específicamente los objetivos que han sido desarrollados para llevar a cabo nuestro trabajo de tesis son listados a continuación.

1.3.1. Objetivos del proyecto

Generales

1. Especificar un lenguaje de coordinación (LC) sencillo para describir el flujo actividades de un servicio Web orquestado.

2. Diseñar e implementar una infraestructura capaz de realizar la ejecución del mismo, aplicando los principios de la POA.

Particulares

1. Obtener aplicaciones distribuidas a través de explotar las ventajas ofrecidas por la POA y los servicios Web.

2. Especificar un LC eficaz para representar el flujo de actividades de servicios Web orquestados.

3. Diseñar e implementar el traductor del LC.

4. Diseñar e implementar una infraestructura de coordinación, capaz de publicar, descubrir y ejecutar servicios Web orquestados.

5. Desarrollar un caso de estudio aplicado a los objetivos anteriores.

1.4. Contribuciones

En nuestra propuesta de trabajo presentamos un nuevo lenguaje composicional llamado BPCL y su correspondiente infraestructura de coordinación, los cuales permitirán describir el flujo de actividades de un servicio Web orquestado, en el cual un desarrollador pueda concentrarse en el diseño conceptual de la aplicación más que en los detalles de distribución de recursos.

Fueron empleados principios simples de diseño y mecanismos de orientación a aspectos para integrar el acoplamiento, comunicación y coordinación de los participantes dentro de la infraestructura. Pretendemos así que sea reducida la complejidad

del proceso de orquestación y se consiga mayor reutilización de código, flexibilidad, modularidad, adaptabilidad y dinamismo en un sistema de negocios.

1.5. Organización de la tesis

La organización general de la tesis abordada en este documento está estructurada por un conjunto de 5 capítulos que describen a detalle nuestra propuesta de trabajo. Además, de una colección de apéndices donde encontraremos información que servirá al lector como una herramienta auxiliar para profundizar en el análisis de los capítulos anteriores.

Capítulos

En el capítulo 2 presentamos los antecedentes generales de nuestra propuesta, cuáles son las tecnologías que sirvieron como base para desarrollarla y analizamos las similitudes, diferencias, ventajas y desventajas de algunos de los principales trabajos relacionados con el nuestro.

En el capítulo 3 describimos a detalle BPCL como lenguaje de coordinación, cuáles son sus características, las directivas de coordinación soportadas, su esquema y el modelo de comunicación utilizado.

En el capítulo 4 abordamos los puntos principales de diseño e implementación de la infraestructura de coordinación realizada y mostramos su funcionamiento creando un caso de estudio que puede ser aplicado en un escenario típico de comercio electrónico.

En el capítulo 5 realizamos una discusión acerca de los resultados obtenidos a través de la tesis presentada, desarrollando a detalle las contribuciones de la misma y el trabajo que puede ser realizado a futuro para incrementar su funcionalidad.

Apéndices

En el apéndice A mostramos el esquema XML asociado a especificaciones de procesos de negocios escritas en BPCL, dicho esquema permitirá validar la correctitud de dichas especificaciones.

En el apéndice B encontraremos la documentación completa de la bliblioteca de coordinación desarrollada en Java, cuáles son las clases que la conforman, constructores, métodos, variables, etc.

En el apéndice C presentamos el conjunto de archivos de implementación (interfaces, aspectos, envolturas, etc.) en Java generados por la infraestructura de coordinación para el caso de estudio propuesto.

En el apéndice D describiremos el contenido del disco compacto que acompaña a este trabajo de tesis, de tal manera que se tenga un fácil acceso al material resultante del desarrollo del mismo.

Finalmente, en el apéndice E encapsulamos en un glosario los términos técnicos más importantes que fueron utilizados en la redacción de esta tesis, con el fin de que el lector entienda ampliamente su uso.

Capítulo 2

Antecedentes

El propósito de este capítulo es presentar los antecedentes generales de nuestra propuesta de trabajo en términos de las tecnologías que sirvieron como base para desarrollarla. Además, analizaremos el enfoque, características, similitudes, diferencias, ventajas y desventajas de algunos de los principales trabajos relacionados con el nuestro. Así, reafirmaremos la necesidad de incorporar un nuevo trabajo de investigación en el ámbito de la orquestación de servicios Web y la POA.

2.1. Automatización de servicios en la Web

El navegador Web es la puerta de entrada a todo un mundo de servicios ofrecidos por empresas y particulares. No obstante, las interacciones en la Web no son sólo entre usuario y aplicación, sino también entre aplicaciones. Las empresas tienen que comunicarse con otras empresas para realizar transacciones más complejas, por ejemplo, al ser auditadas, al compartir recursos con sus filiales, etc. Para que las aplicaciones puedan resolver estas tareas se han creado estándares que se agrupan bajo la denominación de servicios Web. Estos estándares permiten a las empresas categorizar el tipo de servicios que ofrecen, describir cómo deben interactuar con otras aplicaciones y definir cómo será codificada la información intercambiada entre un cliente y el proveedor del servicio.

Los servicios Web serán los bloques básicos para la construcción de nuevas aplicaciones que puedan ser coordinadas u orquestadas. La orquestación de servicios Web es un área de gran interés debido a que esta permite que diversas aplicaciones de negocios ofrecidas por distintos proveedores puedan comunicarse, acoplarse y conjuntarse

en una aplicación orquestada, para obtener un mayor potencial en su funcionalidad.

Al ser WSDL, SOAP y UDDI especificaciones abiertas, facilitan el desarrollo de una colección de software basado en servicios Web. WSDL ofrece un lenguaje formal para definirlos, SOAP permite la interacción entre servicios Web heterogéneos, mientras que UDDI proporciona una amplia infraestructura estandarizada que permite al usuario publicarlos y descubrirlos. Mediante la combinación de estos estándares y de tecnologías como la orquestación y la POA se podrá desarrollar todo un conjunto de aplicaciones de comercio electrónico y B2B, en las cuales su diseño, implementación y mantenimiento sea más rápido y eficiente.

2.2. Marco teórico

2.2.1. Servicios Web

Si tuvieramos varias aplicaciones ya desarrolladas en lenguajes propietarios o en plataformas específicas y quisieramos que éstas pudieran interoperar entre sí, el costo de elegir un único lenguaje o plataforma y migrar dichas aplicaciones al mismo sería un trabajo árduo. Los servicios Web fueron pensados con el fin de hacer frente a este tipo de situaciones. Con los servicios Web es posible reutilizar aplicaciones heterogéneas, de forma tal que puedan comunicarse e interactuar a través del uso de tecnologías estándares desarrolladas en el contexto de Internet.

Así, un servicio Web se define formalmente como "*un componente de software funcional o aplicación Web identificada a través de un URI, cuya interfaz y uso es capaz de ser definida, descrita y descubierta mediante artefactos XML, la cual además soporta interacciones directas con otras aplicaciones de software usando mensajes XML y protocolos basados en Internet*"[10].

La arquitectura sobre la cual descansa todo servicio Web es mostrada en la figura (2.1). Veamos entonces cuáles son los elementos que participan en ésta y cuáles son sus relaciones.

- *Proveedor de servicios Web*. Dicha entidad será la encargada de implementar un servicio y publicarlo en un repositorio. Típicamente este repositorio será un nodo UDDI.

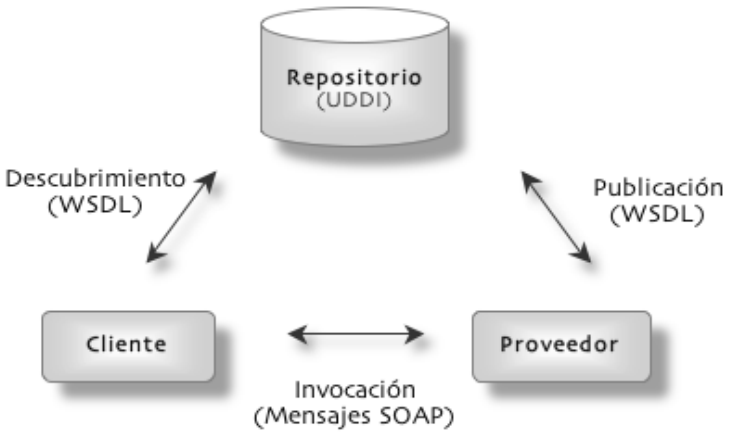

Figura 2.1: Arquitectura de servicios Web.

- *Repositorio de servicios Web*. En dicho almacén serán depositadas todas las descripciones de servicios Web. De este modo un servicio podrá ser solicitado por un cliente.

- *Cliente*. Entidad que solicitará un servicio, para realizarlo tendrá que consultar el repositorio donde éste haya sido publicado y obtendrá su descripción WSDL. Esta descripción será utilizada para crear los mensajes SOAP necesarios al invocar el servicio Web deseado.

Concretamente las características principales de los servicios Web pueden ser resumidas en los siguientes puntos.

1. **Son accesibles a través de la Web**. Gracias a la utilización de protocolos de transporte estándares como HTTP y la codificación de sus mensajes en SOAP, lenguaje estándar basado en XML.

2. **Son interoperables**. Debido a que los servicios pueden interacturar independientemente de la plataforma o lenguaje de programación en el cual hayan sido desarrollados.

3. **Se describen a sí mismos**. De esta forma una aplicación conocerá cuál es la interfaz del servicio y podrá integrarlo y utilizarlo de manera automática. Dicha interfaz debe ser escrita en WSDL, el cual (al igual que SOAP) es un lenguaje basado en XML.

4. **Son localizables**. Mediante el uso de un repositorio de servicios Web, mismo que permitirá que los servicios sean almacenados, publicados y descubiertos por distintas aplicaciones. Actualmente los nodos UDDI representan la tecnología más utilizada [5].

Para concluir el estudio de los servicios Web, sólo nos resta analizar las principales tecnologías que les dan soporte. WSDL, SOAP y UDDI son los estándares de desarrollo más representativos, mientras que WSDK es sólo una de las tantas herramientas para implementar servicios.

2.2.2. Estándares WSDL, SOAP y UDDI

Los estándares son definiciones obtenidas por concenso que aprueban, reconocen y promueven organizaciones internacionales para el intercambio de información o del conocimiento. Generalmente estos organismos están formados por el conjunto de empresas más representativas de un sector o de un campo de la producción. Los estándares permiten que las industrias desarrollen componentes con las garantías suficientes de: interoperabilidad, funcionalidad y calidad. Ayudan a desarrollar los bloques básicos sobre los cuales pueden ser construídos desarrollos tecnológicos. Los estándares son extremadamente importantes en la computación, ya que permiten que se combinen productos de diferentes fabricantes para el desarrollo de sistemas, tanto de software como de hardware.

Sin estándares, sólo los productos de una misma compañía podrían ser utilizados de forma conjunta. Actualmente existen estándares para diversos protocolos de comunicación, formatos de datos, lenguajes de programación, etc. Los organismos más importantes de estandarización son: ANSI, IEEE, ISO y W3C.

Los servicios Web se construyen sobre estándares y a su vez pretenden ser un estándar mediante el cual sea posible construir sistemas a partir de piezas heterogéneas, desarrollas por diversos fabricantes, funcionando en distintos sistemas y construídas con distintas tecnologías. Los principales estándares para el desarrollo de servicios Web son WSDL, SOAP y UDDI.

WSDL

Acrónimo de *Web Services Description Language*, WSDL es un lenguaje XML usado para describir la interfaz de programación de un servicio Web [3]. Esta descripción

puede ser vista como un contrato entre el proveedor del servicio y el cliente, mediante el cual el proveedor del servicio indica: qué métodos se pueden invocar, qué tipos de datos utilizan esos métodos, qué protocolo de transporte se utilizará para el envío y recepción de peticiones y cómo se accederá a los servicios.

Un documento WSDL se estructura de la siguiente manera.

- **Definiciones (Definitions)**. Elemento raíz el cual está constituido por los siguientes subelementos.
  - **Tipos (Types)**. Define tipos de datos complejos.
  - **Mensajes (Messages)**. Declara los mensajes que se intercambiarán durante la invocación de las operaciones. Cada operación tendrá un mensaje de petición y uno de respuesta.
  - **Tipos de puerto (PortTypes)**. Define las operaciones que ofrece el servicio. Cada una tendrá asociado un tipo de mensaje de entrada y salida definidos en la etiqueta anterior.
  - **Enlace (Binding)**. Indica el protocolo y formato de los mensajes.
  - **Servicio (Service)**. Especifica una colección de puertos por medio de los cuales es posible acceder al servicio. Cada puerto tendrá un URL asociado (*endpoint*), mismo que indicará donde está localizada la descripción del servicio.

En la figura (2.2) se muestra el ejemplo de una descripción WSDL asociada a un servicio que permite realizar la cotización de artículos.

SOAP

Una vez definida la descripción de un servicio Web, un usuario deberá utilizar un protocolo de mensajes para poder solicitar alguna operación provista por el servicio. SOAP es el acrónimo de *Simple Object Access Protocol*, es un protocolo basado en XML que se usa para codificar los mensajes de petición, respuesta y errores en las peticiones a métodos de servicios Web envíados a través de la red [4].

Los mensajes SOAP son independientes de los sistemas operativos y pueden ser transportados usando una gran variedad de protocolos de Internet, incluyendo SMTP y HTTP. Un mensaje SOAP está estructurado por los siguientes elementos.

```
1:  <?xml version="1.0" encoding="UTF-8"?>
2:  <definitions name="Quotation" ...
3:    <types> ... </types>
4:    <message name="getQuotationRequest">
5:      <part name="article" type="xsd:string"/>
6:      <part name="urlDB" type="xsd:string"/>
7:    </message>
8:    <message name="getQuotationResponse">
9:      <part name="cost" type="xsd:float"/>
10:   </message>
11:   <portType name="QuotationPortType">
12:     <operation name="getQuotation">
13:       <input message="tns:getQuotationRequest"/>
14:       <output message="tns:getQuotationResponse"/>
15:     </operation>
16:   </portType>
17:   <binding name="QuotationBinding" type="tns:QuotationPortType">
18:     <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
19:       ...
20:   </binding>
21:   <service name="Quotation">
22:     <port binding="tns:QuotationBinding" name="QuotationPort">
23:       <soap:address
24:        location="http://127.0.0.1:6080/Quotation/services/QuotationPort"/>
25:     </port>
26:   </service>
27: </definitions>
```

Figura 2.2: Ejemplo de una descripción WSDL.

- **Sobre (Envelope)**. Representa el contenedor del mensaje. Elemento obligatorio utilizado para especificar cómo se codificará el mensaje. Los subelementos por los cuales está constituido son:

  - **Cabecera (Header)**. Elemento opcional en el cual son definidos los actores que proveen el servicio.
  - **Cuerpo (Body)**. Elemento obligatorio en donde se especifican los métodos que serán solicitados con sus respectivos parámetros. Además es posible definir etiquetas de error (*soap faults*) para indicar si se produjeron errores al procesar peticiones.
  - **Attachments (opcional)**. Permite intercambiar o añadir datos que no sean XML al mensaje, por ejemplo imágenes.

En la figura (2.3) se muestra el ejemplo de un mensaje SOAP para la solicitud de un método ofrecido por un servicio Web que permite realizar la cotización de un artículo en una tienda especificada a través de su URL. En la figura (2.4) se muestra el mensaje SOAP de respuesta para el ejemplo anterior.

```
1:  <?xml version="1.0" encoding="UTF-8"?>
2:  <soapenv:Envelope
3:      soapenv:encodingStyle=="http://schemas.xmlsoap.org/soap/encoding/"
4:      xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
5:      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
6:      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
7:      xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">
8:
9:      <soapenv:Body>
10:         <getQuotationRequest xmlns="">
11:             <article>Ensayo sobre la ceguera</article>
12:             <urlDB>comprarDB1</urlDB>
13:         </getQuotationRequest>
14:     </soapenv:Body>
15: </soapenv:Envelope>
```

Figura 2.3: Ejemplo de un mensaje SOAP de petición.

```
1:  <?xml version="1.0" encoding="UTF-8"?>
2:  <soapenv:Envelope
3:      xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
4:      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
5:      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
6:
7:      <soapenv:Body>
8:          <getQuotationResponse xmlns="">
9:              <cost xsi:type="xsd:float">290.10</cost>
10:         </getQuotationResponse>
11:     </soapenv:Body>
12: </soapenv:Envelope>
```

Figura 2.4: Ejemplo de un mensaje SOAP de respuesta.

UDDI

Para que los desarrolladores puedan crear clientes que utilicen servicios Web es necesario que conozcan cuáles son los servicios disponibles, qué es lo que ofrecen y cuáles son sus interfaces. La mejor opción consiste en utilizar Internet para crear un registro estándar donde se pueda almacenar este tipo de información y acceder a ella. Ésto es justamente lo que hace UDDI(*Universal Description, Discovery and Integration*). UDDI es un directorio de servicios Web distribuido el cual permite a los proveedores de servicios Web dar a conocer sus ofertas de una forma estándar de tal manera que los clientes puedan consultarlos e invocarlos [5].

Un registro en el directorio UDDI es un archivo XML que describe una empresa y los servicios que ofrece. Dicho registro consta de tres partes. Las *páginas blancas* describen a la empresa que ofrece el servicio: nombre, dirección, contactos, etc. Las *páginas amarillas* incluyen a los servicios en categorías industriales basadas en

clasificaciones estándar como el *North American Industry Classification System* y la *Standard Industrial Classification*. Las *páginas verdes* describen la interfaz del servicio con suficiente detalle para quienes quieran escribir una aplicación que utilice el servicio Web. Los servicios se definen mediante un documento UDDI llamado ModelType o tModel. En muchos casos, el tModel contiene un archivo WSDL que describe la interfaz para un servicio Web.

El directorio UDDI también ofrece varias formas para buscar servicios, por ejemplo, se puede buscar a los proveedores de un servicio en una zona geográfica concreta o en una empresa de un tipo determinado. El directorio UDDI proporcionará toda la información necesaria, contactos, enlaces y datos técnicos que permitirán decidir cuáles son los servicios que cumplen las condiciones requeridas.

UDDI es una pieza importante en el área de los servicios Web. Sin él, las organizaciones que tienen servicios Web no podrían dar a conocer fácilmente a sus posibles clientes qué es lo que ofrecen, y los clientes no podrían conocer los datos que necesitan para acceder a esos servicios.

2.2.3. WebSphere SDK for Web Services

WebSphere de IBM constituye una plataforma para aplicaciones de comercio electrónico, ofreciendo completa compatibilidad con los estándares de servicios Web y la plataforma *Java^{TM} 2 Platform, Enterprise Edition (J2EE^{TM})*, esta última basada en un enfoque orientado a componentes para el diseño, desarrollo e implementación de aplicaciones empresariales Java [11].

WebSphere está constituida por un conjunto de productos que ofrecen una amplia gama de soluciones al permitir la transformación de negocios tradicionales en negocios electrónicos. Los grupos en los cuales divide sus productos son: *Foundation and Tools*, *Reach and User Experience* y *Business Integration*. El primer grupo consta de los productos que forman la infraestructura básica de la plataforma siendo estos *WebSphere Application Server* y *WebSphere Studio*. WebSphere Application Server incluye dos paquetes con el soporte necesario para el desarrollo de aplicaciones basadas en servicios: *WebSphere SDK for Web Services (WSDK)* y *Emerging Technologies Toolkit (ETTK)*.

El más importante de los paquetes es WSDK ya que éste incluye un conjunto de herramientas para el diseño, implementación y ejecución de aplicaciones Java basa-

das en servicios Web. La funcionalidad de WSDK se basa en especificaciones abiertas como SOAP, WSDL y UDDI [12]. Para poder utilizar este paquete, los servicios desarrollados deben seguir los estándares definidos por la *Web Services Interoperability Organization (WS-I)* [13].

WSDK provee el conjunto de herramientas que permiten habilitar, utilizar y publicar aplicaciones J2EE. Dichas herramientas se muestran en la tabla (2.1).

Elemento	Descripción
Bean2WebService	Crea un servicio Web completo a partir de una clase Java.
EJB2WebService	Crea un servicio Web completo a partir de un módulo EJB.
WSDL2WebService	Crea un servicio Web completo a partir de especificaciones WSDL.
UDDIPublish	Publica un servicio Web en un registro UDDI público o privado.
UDDIUnpublish	Elimina un servicio Web de un registro UDDI público o privado.
Tcpmon	Monitorea mensajes SOAP intercambiados entre clientes y servicios Web.
Appserver	Provee un conjunto de funciones administrativas que permiten iniciar, detener, instalar, desinstalar o listar servicios Web en el contenedor de servicios de WebSphere.

Tabla 2.1: Herramientas para el desarrollo de servicios Web en WSDK.

Por último, *Java Web Services Developer Pack (JWSDP)* representa un entorno de programación alternativo para crear servicios Web basados en Java. Sin embargo, el desarrollo de éstos en dicho entorno resulta un tanto complejo, debido a que un usuario tendría que tener conocimiento alguno del uso de ciertas API's especializadas como: *Java API for XMLProcessing (JAXP), Java API for XML-based RPC (JAX-RPC), Java APIs for XML Registries (JAXR)* y *SOAP with Attachments API for Java (SAAJ)*, para poder implementarlos [14].

Por tal razón, con el fin de satisfacer nuestros objetivos, elegimos WSDK para crear servicios Web, ya que representa una de las plataformas de trabajo más amigables, al brindar grandes facilidades para la generación, instalación e inicialización

de los mismos, permitiéndonos así enfocar nuestra atención en los mecanismos de orquestación de servicios más que en los de generación.

2.2.4. Orquestación de servicios Web

Al ser componentes, los servicios Web pueden verse como cajas negras con la capacidad de ser utilizadas y reutilizadas sin necesidad de conocer como fue implementada su funcionalidad [1]. Dicha propiedad facilita la composición de servicios.

El proceso de orquestación consiste en "*relacionar, organizar y administrar las interacciones entre los servicios Web referentes a la lógica del proceso de negocios*"[15]. Los elementos básicos para llevarla a cabo son:

1. *Procesos*. Vistos como una serie de actividades cuyo fin es la ejecución de una tarea determinada, por ejemplo, *un plan de viaje*, *la compra de artículos electrónicamente, etc.*

2. *Actividades*. Representan reglas bien definidas del proceso de negocios. Para el ejemplo anterior del proceso *plan de viaje* deberán ser contempladas las actividades *consulta de hospedaje* y *reservación de boletos de avión, entre otras*.

3. *Flujo de datos*. Describe la información intercambiada entre actividades.

4. *Flujo de control*. Describe el orden en que serán ejecutadas las actividades del flujo. Dicho orden será especificado en términos de construcciones usuales de programación como la composición: secuencial, concurrente y condicional [6].

Típicamente la orquestación de servicios Web se ejecuta por un sólo nodo coordinador, el cual será responsable de administrar el flujo de datos y el flujo de control entre los componentes (servicios Web). Es decir, éste recibirá las peticiones de los clientes, hará las transformaciones de los datos requeridos e invocará los componentes en base a su especificación. A este modo de ejecución se conoce como *orquestación centralizada*, la cual tiene como principal característica que todos los datos transferidos entre componentes se realizará vía el coordinador, en lugar de ser transferidos directamente desde el punto de generación al punto de consumo [16]. A manera de restricción en nuestra propuesta de tesis utilizaremos el tipo de orquestación centralizada, ya que ésta nos permitirá tener un mayor control del flujo de actividades en cada momento de su ejecución [15].

En la figura (2.5) podemos observar gráficamente cómo se lleva a cabo la orquestación de servicios Web, donde el rectángulo mayor representa el proceso que será orquestado, los rectángulos sombreados pequeños las actividades involucradas en el mismo, las flechas horizontales el flujo de datos intercambiado y las flechas verticales el flujo de control del proceso.

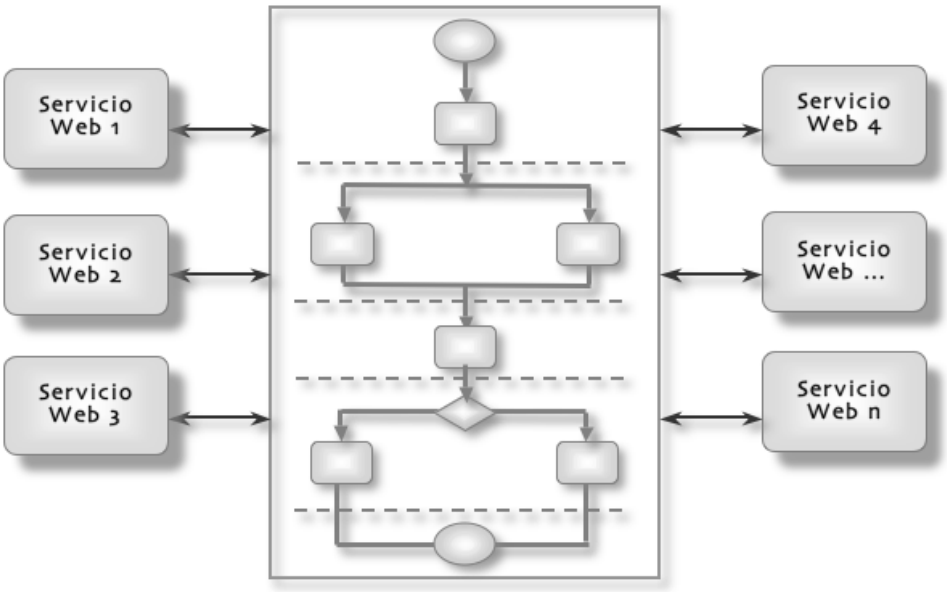

Figura 2.5: Orquestación de servicios Web.

Al orquestar servicios, como primer paso necesitaremos elegir un *lenguaje de cooordinación* que cumpla con un conjunto de características deseables para coordinar correctamente los componentes involucrados. Dicho lenguaje deberá ser *simple, abstracto*, dar *soporte a enlace dinámico de recursos y a actividades primitivas y estructuradas* y además deberá garantizar *persistencia y correlación* a lo largo de las transacciones del flujo [17].

Finalmente, como segundo paso será necesario definir una *infraestructura de coordinación*, la cual ofrecerá todos los mecanismos necesarios en cómputo para dar soporte a las construcciones del lenguaje y además llevará a cabo el descubrimiento, publicación y ejecución de servicios Web orquestados.

2.2.5. Programación orientada a aspectos

Consideremos el ciclo de vida típico de un proyecto de comercio electrónico. En primera instancia, debe decidirse la arquitectura del sistema a ser implementada, después deben diseñarse las interfaces y clases que representen los conceptos del negocio, un cliente, un carrito electrónico, el inventario, autorizaciones de pago, etc. y por último se codificará la funcionalidad asociada a estos conceptos. Es muy probable que al concluir este punto, el código de la aplicación sea claro y comprensible. No obstante, éste será cada vez más difícil de entender a medida que sean incorporadas funcionalidades adicionales al sistema, por ejemplo en cuanto a manejo de excepciones, administración de transacciones, control de concurrencia o reglas del negocio. A este fénomeno lo conocemos como *invasividad de código* [18] y los principales problemas que genera en una aplicación son:

- *Código enredado* (*Tangling*). Occurre cuando en una unidad del programa (clase o método) son implementados múltiples requerimientos de manera entrelazada.

- *Dispersión de código* (*Scattering*). Occurre cuando en múltiples módulos se incluye el código que implementa a un mismo requerimiento [19].

Indudablemente desearíamos que esta situación pudiera ser diferente, de forma tal que sólo fueran definidos módulos que implementaran funcionalidades adicionales sin la necesidad de realizar modificaciones a la funcionalidad básica del sistema, dejándolo todo tan claro como en la fase inicial. Debido a dichos inconvenientes, la POA plantea un modelo viable para evitar o controlar lo más posible la invasividad de código.

Los objetivos que persigue la POA son principalmente el de separar conceptos y el de minimizar sus dependencias. Con el primer objetivo se consigue que las funcionalidades no básicas del sistema sean encapsuladas y con el segundo se consigue la reducción del acoplamiento entre los distintos elementos. La POA introduce la noción de *aspecto* [1] para denotar aquellas funcionalidades adicionales de un sistema de negocios.

Un *aspecto* se define como "*una unidad modular del programa que aparece y afecta a otras unidades modulares del mismo*"[19]. Ejemplos de *aspectos* en un sistema de

[1] Usaremos letras cursivas para distinguir entre el término que denota al concepto computacional de aquel que se usa como sinónimo de la palabra atributo.

comercio electrónico son: *el registro del historial de un programa (logging), la autenticación, la seguridad, el manejo de excepciones, administración de transacciones, el control de concurrencia, implementación de reglas de negocios, etc* [18]. Los elementos que conforman un *aspecto* se listan a continuación:

- Un *punto de unión (join point)* es un sitio perfectamente identificable en la ejecución de un programa, es aquí donde se determinará el momento en el cual serán acoplados los *aspectos*. Algunos ejemplos son la llamada a un método, su ejecución, la creación de un objeto, el acceso a un campo o la captura de excepciones.

- Un *punto de corte (pointcut)* es un constructor donde se especifican los puntos de unión a alcanzar, pueden verse como las reglas para identificar dichos puntos. Además mediante un punto de corte será posible capturar la información referente al contexto de un punto de unión. Por ejemplo, en la ejecución de un método pueden conocerse el objeto sobre el que se invoca el método, así como los argumentos de la llamada y su valor de retorno.

- *Redefinición de comportamiento (advice)* es el código que debe ejecutarse como respuesta a un punto de unión especificado en un punto de corte, el cual puede realizarse antes, durante o después de dicho punto. Ésto quiere decir que es posible modificar la ejecución del código en el punto de unión definido, reemplazarlo o ignorarlo. [18].

A diferencia de los esquemas tradicionales de programación que utilizan un compilador para obtener aplicaciones ejecutables, al desarrollar aplicaciones orientadas a aspectos deberá ser utilizado un proceso adicional llamado *weaving process*, el cual permitirá acoplar *aspectos* a las aplicaciones. El proceso de acoplamiento puede ser realizado *estáticamente*, es decir, en tiempo de carga y de compilación, o *dinámicamente* en tiempo de ejecución [18]. La figura (2.6) muestra la diferencia entre un esquema tradicional y uno orientado a aspectos.

Uno de los principales lenguajes de programación orientados a aspectos es AspectJ, dicho lenguaje y su especificación será abordada en esta sección debido a que nuestra propuesta fue desarrollada bajo este esquema.

AspectJ es una extensión orientada a aspectos de Java, ésto significa que cualquier programa Java será válido en AspectJ. El compilador de AspectJ genera archivos *class* conforme a la especificación Java *byte-code*, por lo que una máquina virtual de

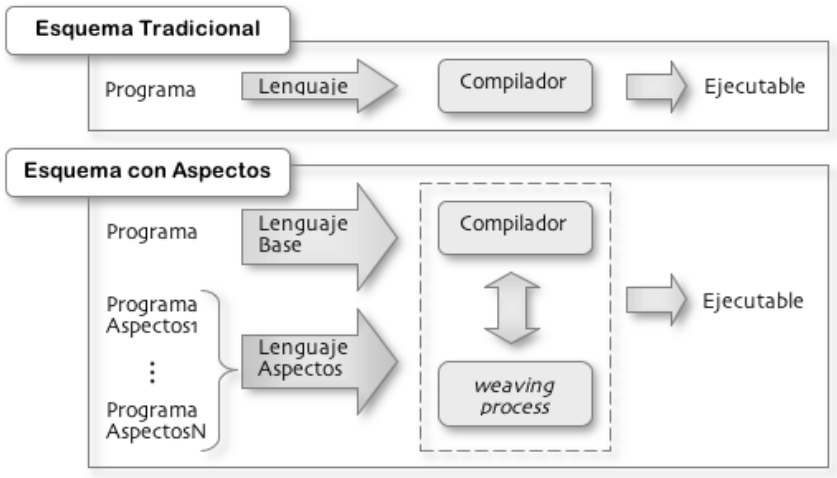

Figura 2.6: Contraste entre un esquema de programación tradicional y uno orientado a aspectos.

Java (JVM) puede ejecutarlos [20].En AspectJ los puntos de corte se pueden declarar anónimos o con un nombre. Al igual que las clases, un punto de corte anónimo se define en el lugar donde se usa, que bien podría ser en un *advice* o en la definición de otro *pointcut*. No obstante, los puntos de corte que se definen con un nombre pueden ser referenciados desde múltiples partes del código haciéndoles así reusables. La sintaxis asociada a cada uno de éstos es la siguiente (2.2).

Punto de corte	Sintaxis

Anónimo	`typeAdvice(): jointPoint( definition* ) { ... }`
	
	`Ej. before(): call( public * MyProcess.* ( .. )) { ... }`

Nombrado	`pointcut namePointcut(): jointPoint( definition* )`
	`typeAdvice: namePointcut() { ... }`
	
	`Ej. pointcut countTime(): execution( public * MyProcess.* ( .. ));`
	`before(): countTime() { ... }`

Tabla 2.2: Sintaxis de un punto de corte en AspectJ.

donde

- typeAdvice. Define el tipo de *advice* a ser acoplado, el cual puede ser *before*, *around* o *after*.

- jointPoint. Define el tipo de punto de unión a ejecutarse.
  - *call*. Llamada a métodos o constructores de clases.
  - *execution*. Ejecución de métodos o constructores de clases.
  - *get*. Lectura de atributos.
  - *set*. Escritura de atributos.
  - *initilization*. Inicialización de objetos.
  - *handler*. Captura de excepciones.
- definition. Especifica el punto de unión que será afectado por un *advice*, su declaración es la siguiente: [visibility] [typeReturn] [class] [method] [args]. Indicando la visibilidad del mismo, es decir, si es público, privado o protegido, tipo de retorno, clase a la que pertenece, nombre del método y sus argumentos. Para definir estos elementos pueden ser utilizados como comodínes * y . que servirán para capturar puntos de unión que comparten características comunes.

Los puntos de corte serán encapsulados en clases denominadas *aspectos*, los cuales se acoplarán estática o dinámicamente. En la figura (2.7) se muestra el ejemplo de un *aspecto* encargado de calcular el tiempo de ejecución de cada una de las funciones de una clase, en este caso *MyProcess*.

```
1:  import java.util.*;
2:
3:  aspect ExecutionTime
4:  {
5:     private long time = 0;
6:
7:     pointcut countTime(): execution(* MyProcess.*(..));
8:
9:     before():countTime()
10:    {
11:       System.out.println("Before of function: "+thisJoinPoint);
12:       time = System.currentTimeMillis();
13:    }
14:
15:    after():countTime()
16:    {
17:       System.out.println("After of function: "+thisJoinPoint+
18:       ", time calculated is: " + (System.currentTimeMillis() - time));
19:    }
20: }
```

Figura 2.7: Ejemplo de un *aspecto* escrito en AspectJ.

Hasta este momento hemos analizado el conjunto de tecnologías predominantes para el desarrollo de aplicaciones basadas en servicios Web, en la siguiente sección

presentaremos distintos enfoques de trabajo que hacen uso de estas tecnologías y cuáles son las principales características, similitudes y diferencias en relación con nuestra propuesta.

2.3. Trabajo relacionado

El proceso de orquestación de servicios Web no es una tarea sencilla, ya que deben ser considerados distintos elementos que interactúan de manera distribuida. Los trabajos relacionados que analizaremos a continuación tienen como común denominador facilitar dicho proceso, estableciendo mecanismos mediante los cuales se obtenga mayor dinamismo en el cambio de requerimientos en la coordinación de servicios.

2.3.1. Business Process Execution Language

La composición de servicios Web es un nuevo paradigma que intenta incrementar el potencial de las aplicaciones Web. Diversos proveedores de tecnologías de información como IBM, Microsoft y Sun MicroSystems han diseñado lenguajes y técnicas para implementar dicho paradigma. Sin embargo el representante más significativo en nuestros días es BPEL4WS o BPEL (Business Process Execution Language for Web Services) desarrollado por IBM, Microsoft y BEA Systems [21].

BPEL es un lenguaje estático composicional orientado al modelado de procesos orquestados, el cual cuenta con su propio motor de ejecución conocido como BPWS4J [8]. BPEL distingue entre procesos abstractos y ejecutables. Los procesos abstractos definen protocolos de negociación, especificando los mensajes intercambiados entre los diferentes participantes del proceso orquestado, mientras que los procesos ejecutables especifican el orden de ejecución de los elementos que constituyen dicho proceso (proveedores de servicios, mensajes intercambiados, manejo de exepciones, etc.). Los procesos abstractos son útiles para describir protocolos de comunicación, mientras que los procesos ejecutables deben ser compilados para obtener servicios invocables. Otras características de BPEL son: el uso de referencias a tipos de puertos (*portType* [2]) contenidos en documentos WSDL, el manejo de compensación de transacciones y la disponibilidad de un mecanismo de captura y gestión de excepciones similar al proporcionado por Java.

[2]Interface abstracta que define el conjunto de operaciones asociadas a un servicio Web.

BPEL está relacionado con otras dos especificaciones WS-Coordination (WS-C) [22] y WS-Transaction (WS-T) [23]. La primera de ellas soporta, integra y unifica diferentes modelos de coordinación permitiendo a diferentes sistemas interoperar. Provee además mecanismos estándares para crear y registrar servicios usando los protocolos definidos en WS-T. Esta última coordina la ejecución de operaciones distribuidas en el ambiente de servicios Web a través de la definición de protocolos para transacciones atómicas, de negocios, compensación, etc.

Ahora, una vez que hemos analizado a grandes rasgos las principales características de BPEL, en el siguiente apartado analizaremos las deficiencias más importantes que presenta este proyecto.

Deficiencias

- No soporta adaptación dinámica de componentes. BPEL al ser un lenguaje estático, ciertos componentes que son suceptibles al cambio durante la composición de servicios no pueden ser modificados dinámicamente. Por ejemplo, para añadir o remover elementos de un servicio, o cambiar algún requerimiento no funcional de la aplicación, el proceso que está siendo orquestado tendría que ser suspendido, modificado y puesto en ejecución nuevamente para que estos cambios sean reconocidos.

- No soporta transparencia de localidad de recursos. Ésto quiere decir que la ubicación de los servicios debe ser explicítamente especificada en el proceso, debido a que BPEL fue concebido como un sistema para ambientes distribuidos. Dicha limitante resta claridad a la descripción del flujo a ser orquestado y dinamismo en el cambio de componentes.

- Sólo permite la invocación de métodos que sean provistos por un servicio Web. Es decir, alguna funcionalidad local o propia al servicio que orquesta no puede ser descrita a menos que ésta sea implementada como un método de un servicio Web.

- No es suficientemente modular. Debido a que la incorporación de requerimientos no funcionales como el manejo de excepciones, el logging, el control de acceso, etc, afectará diversos puntos de la especificación de un proceso, lo cual tiene como consecuencia la invasividad de código redundante en una aplicación.

- Tiene un costo elevado de licencia. Lo cual hace muy poco factible su obtención.

2.3.2. BPEL for Java

BPEL4J es el acrónimo de BPEL for Java, dicho proyecto fue desarrollado por BEA Systems e IBM. Esta propuesta extiende la funcionalidad de BPEL ofreciendo la pauta para que dos lenguajes de programación BPEL y Java puedan ser utilizados de manera conjunta, permitiendo así gozar de las ventajas que cada uno provee. Para realizarlo, BPEL4J permite la incorporación de pequeños fragmentos de código escritos en Java (denominados *snippets*) en la especificación de un proceso BPEL.

Dicha característica permite que puedan ser incorporados métodos propios a la aplicación, es decir, ahora pueden definirse métodos que aceptan como parámetros objetos arbitrarios Java, ya que debido a la naturaleza de BPEL un servicio sólo podía aceptar parámetros descritos en XML, lo cual restringía enormemente la expresividad del lenguaje porque cada función a ser incluida tendría que ser implementada como parte de un servicio Web. Otra ventaja es la posibilidad de expresar el comportamiento que proveee BPEL como clases escritas en Java [9]. Veamos ahora cuáles son las principales deficiencias presentes en este proyecto.

Deficiencias

- Presenta las limitantes inherentes a BPEL. Ver detalles en la sección 2.3.1.
- Añade mayor complejidad a la definición de un flujo de trabajo. Ya que el código escrito en Java y la especificación del flujo de trabajo se intercala o combina, restando claridad a la definición del proceso orquestado.

2.3.3. Aspect-Oriented for BPEL

Aún son pocas las aplicaciones desarrolladas en el ámbito de la programación orientada a aspectos, debido a que el auge ha sido con respecto a la creación de lenguajes de programación, algunos ejemplos son: HyperJ, AspectJ, AspectC++ y AspectR, que permiten escribir este tipo de aplicaciones [24]. A pesar de ésto AO4BPEL (Aspect-Oriented for BPEL) es uno de los trabajos más relevantes hasta el momento que introducen los conceptos de composición de servicios Web y la programación orientada a aspectos.

AO4BPEL fue desarrollado por el Grupo de Tecnología de Software en Darmstadt, Alemania, siendo una extensión orientada a aspectos de BPEL [21]. Dicha extensión

es útil para mejorar la modularidad y adaptabilidad de las especificaciones de composición de servicios Web y brindar soporte para adaptaciones dinámicas tales como la coordinación y el acoplamiento o desacoplamiento de *aspectos* en tiempo de ejecución. BPEL fue escogido como base para hacer que cada actividad definida se convirtiera en un punto de unión y ésto permitiera ejecutar servicios por medio de *aspectos* [25]. Las principales limitantes de AO4BPEL se enlistan a continuación.

Deficiencias

- Presenta las limitantes inherentes a BPEL. Ver detalles en la sección 2.3.1.
- Añade mayor complejidad a la definición de un flujo de trabajo. Debido a que el usuario que describe el proceso de negocios debe tener conocimiento acerca de la especificación de *aspectos*, qué es un punto de unión, de corte, qué es un advice, etc.
- No soporta transparencia de localidad de recursos. En los *aspectos* la localización de recursos también debe ser explícitamente especificada al igual que en BPEL.

2.3.4. Dynamic Business Rules for WS Composition

El artículo de investigación presentado en *Dynamic Business Rules for Web Service Composition* [26] tiene como objetivo abordar un enfoque orientado a reglas dinámicas de negocios para incrementar el grado de dinamismo en la composición de servicios Web. Estas reglas definen o restringen diferentes aspectos de la composición, por ejemplo, al especificar: (a) cuáles son los servicios que tendrían que ser descubiertos, (b) cómo estos deberían ser elegidos y acoplados y (c) cómo la composición se adaptaría a los cambios del negocio, particularmente en este enfoque sólo es considerado este último tipo de reglas.

Actualmente los lenguajes orientados a la definición de procesos no permiten definir dichas reglas de una manera clara, modularizada y reusable, como lo es el caso de BPEL. En BPEL las reglas del negocio se entrelazan con la funcionalidad central del proceso y generalmente en el momento que éstas deben ser modificadas tienen que ser realizados los cambios manual e invasivamente. Otras desventajas de las especificaciones actuales son sus elevados costos por licencia de uso e inherentemente por su complejidad la dificultad para que éstas sean fácilmente aprendidas.

Debido a las limitantes anteriores, el enfoque presentado en esta sección pretende realizar un lenguaje composicional basado en reglas el cual describa especificaciones de procesos utilizando un lenguaje tan cercano al natural como sea posible donde además las reglas del negocio logren ser separadas de los detalles de la implementación de procesos.

Para conseguirlo, los mecanismos de la POA serán utilizados como un medio para desacoplar las reglas de la funcionalidad fundamental de los procesos, obteniendo así *reglas dinámicas*, mismas que permitirán modificar la composición de servicios al basarse en condiciones o patrones históricos de su ejecución. La traducción de dichas especificaciones serán implementadas en el lenguaje de programación orientado a aspectos JAsCO [27]. JAsCO encapsulará las reglas del negocio como *aspectos*, mediante los que podrán ser referenciados puntos de corte que hayan sido previamente ejecutados. Finalmente, cabe señalar que dicha propuesta se encuentra aún en una etapa de planeación y diseño.

2.3.5. Reflective Engineering for Web Services

Actualmente la modificación de un servicio Web implica la disponibilidad, edición, recompilación e invasividad de su código fuente. Dependiendo de la aplicación de servicios, el proceso de desarrollo de unos puede ser más sencillo que otros. Si la aplicación soporta la carga dinámica de otras aplicaciones entonces el desarrollo se convierte en una tarea "sencilla" ya que de lo contrario para modificarla se tendría que suspender el servicio, realizar una nueva versión del mismo, reemplazar la versión anterior por la actualizada y ejecutarla nuevamente.

La propuesta de trabajo expuesta en *Reflective Engineering for Web Services* [28], introduce la *reflexión* como un mecanismo de los sistemas computacionales el cual les permite razonar, actuar por sí mismos y modificar dinámicamente su comportamiento. Aunque este concepto ha sido exitosamente aplicado en algunos campos de investigación como son los sistemas distribuidos, la programación concurrente, la programación orientada a aspectos, etc., su aplicación en el diseño de servicios Web aún no ha sido abordada.

La reflexión aplicada a los servicios Web incrementa su adaptabilidad y flexibilidad. Por lo cual, en esta propuesta se presenta una arquitectura reflexiva para servicios Web (*Reflective Architecture for Web Services, RAWS*), misma que facilita la modificación dinámica de servicios durante su ejecución.

En términos generales, la reflexión en un sistema se clasifica en tres grupos, dependiendo de la información que el sistema pueda manipular. RAWS aplica éstos en el diseño de los servicios Web.

a. *Por introspección*
El sistema es capaz de observar y razonar acerca de sus elementos, pero no puede modificarlos.

b. *Estructural.*
El sistema puede consultar y modificar su estructura en tiempo de ejecución.

c. *Por comportamiento.*
El sistema puede manipular y modificar su comportamiento en tiempo de ejecución.

La principal contribución de este trabajo es la introducción de una arquitectura mediante la cual puedan ser diseñados servicios Web flexibles y adaptables basados en la reflexión, permitiendo la modificación dinámica en la estructura que define e implementa a un servicio Web, por ejemplo, en cuanto su descripción (WSDL), modelo de comunicación (SOAP) y modelos de localización y publicación (UDDI) .

2.3.6. Comparación con tecnologías existentes

Una vez que hemos analizado un conjunto significativo de trabajos relacionados con el nuestro, resulta clara la necesidad de introducir un nuevo enfoque que incorpore mecanismos para solucionar algunas de las deficiencias más relevantes del proceso de orquestación de servicios Web. Las principales características que implementa nuestro trabajo de tesis son las siguientes.

- **Brinda las facilidades necesarias para realizar el diseño de un proceso de negocios como si éste fuera centralizado**, permitiéndonos así enfocarnos en su especificación, tener mayor precisión y claridad durante su descripción y promover la trasparencia de localidad de los servicios involucrados. Para logralo, fue necesario llevar a cabo los siguientes puntos.

  1. Especificación de un lenguaje composicional para la descripción de procesos de negocios llamado *Business Process Coordination Language (BPCL)*, el cual además de dar soporte a un conjunto de actividades primitivas y

estructuradas, permite introducir secciones de código escritas en Java, encapsuladas como llamadas a procedimientos locales. BPCL se introducirá a detalle en el capítulo 3.

2. Creación de un motor de ejecución para procesos BPCL al que denominamos *infraestructura de coordinación*, la cual además de ponerlos en ejecución, es la encargada de acoplar los *aspectos* de búsqueda e invocación de servicios. Aunque sólo nos enfocamos en dichos *aspectos*, éstos fueron contemplados por su gran importancia en la orquestación, debido a que brindan mayor dinamismo en el cambio de participantes y en la invocación de sus respectivas funcionalidades. Cabe señalar que no especificamos los *aspectos* en el lenguaje de coordinación, evitando así que un usuario deba tener algún conocimiento de la POA, y además que los puntos de unión definidos en los *aspectos* están vinculados con las directivas de invocación, ya que durante la orquestación estas actividades representan los puntos de interacción con los distintos proveedores de servicios. La infraestructura de coordinación se analizará a detalle en el capítulo 4.

- **Utiliza el enfoque orientado a aspectos para añadir mayor modularidad y adaptabilidad a un proceso de negocios**. Permitiendo de esta forma que una aplicación inherementemente distribuida pueda ser probada en cada una de sus etapas de desarrollo sin necesidad de que esta esté conectada en red. Es importante puntualizar que la POA se basa y formaliza los principios empleados en la reflexión, de tal manera que sea posible hacer uso de las ventajas que nos provee este mecanismo de una manera más simple. Siendo así, los tipos de reflexión empleados en nuestro trabajo son por introspección y comportamiento.

  a. *Reflexión por introspección*
  Utilizada mediante el *Reflection API* de Java, el cual permite que el sistema de coordinación introducido sea capaz de invocar uniformemente cualquier método de un servicio Web donde su descripción (objeto que realiza la invocación, nombre, parámetros, tipos de datos, etc.) es conocida hasta tiempo de ejecución.

  b. *Reflexión por comportamiento*
  Empleada al incorporar *aspectos*, a través de los cuales es posible que la infraestructura de coordinación implemente mecanismos para realizar el descubrimiento e invocación dinámica de recursos (servicios Web).

2.4. Sumario

La automatización de servicios en la Web a través de la introducción de nuevos enfoques y tecnologías que permitan facilitar el desarrollo y dinamismo de aplicaciones de negocios, es una de los grandes metas que persiguen día a día las organizaciones.

Particularmente, el desarrollo de aplicaciones orquestadas representa un área de gran interés en la actualidad, ya que las empresas al comunicar, acoplar y coordinar sus aplicaciones con las de otras, pueden ver incrementado su rendimiento.

Así pues, dado que las herramientas de trabajo existentes brindan grandes beneficios para logralo, el uso de la coordinación de servicios Web y la POA, como se proponen en este trabajo, representan una combinación poderosa para el desarrollo de aplicaciones dinámicas de comercio electrónico y B2B.

Capítulo 3

Business Process Coordination Language

Como se ha mencionado en capítulos anteriores, el objetivo de nuestra propuesta de trabajo es introducir un nuevo lenguaje composicional por medio del cual la descripción de un proceso orquestado pueda realizarse sin necesidad de especificar detalles de distribución de recursos y además crear una infraestructura de coordinación que entre otras tareas sea la encargada de ejecutar dichos procesos. Así, en este capítulo nos concentraremos en el análisis del lenguaje de coordinación propuesto, al cual denominamos *Business Process Coordination Language (BPCL)*.

3.1. Descripción general

La orquestación de servicios Web representa un enfoque práctico que intenta incrementar el rendimiento de una organización (mejor funcionalidad, mejor calidad de servicios, mayor flexibilidad al cambio, mayor facilidad en el desarrollo de aplicaciones) a través de la integración de un conjunto diverso de aplicaciones de negocios. Para llevarla a cabo como primer paso será necesario elegir un lenguaje de coordinación en el cual pueda especificarse el flujo de trabajo a ser orquestado y como segundo paso para ejecutar dicho flujo será necesario contar con su respectivo motor de ejecución.

Sin embargo, en la mayoría de los lenguajes de coordinación existentes como BPEL4WS y BPEL4J, la descripción de un proceso de negocios resulta ser muy compleja, ya que los detalles de localización de recursos (servicios Web) deben ser conocidos y especificados durante la definición del proceso. Dicha restricción hace que

este tipo de aplicaciones Web sean estáticas y su desarrollo sea más complicado. Como una alternativa que pretende reducir la complejidad del proceso de orquestación de servicios Web en este proyecto de tesis diseñamos un nuevo lenguaje composicional llamado BPCL, en el cual los detalles físicos de distribución de recursos no necesitan ser descritos ya que éstos serán conocidos hasta la ejecución del flujo de trabajo, brindando así mayor claridad y simplicidad en la descripción del mismo.

BPCL está fundamentado en un lenguaje bien definido para desarrollar programas distribuidos llamado *Synchronizing Resources (SR)* [29]. Aunque BPCL sólo soporta un conjunto básico de construcciones definidas en SR para coordinar un proceso de negocios, dichas construcciones fueron acopladas para que éstas funcionaran en ambientes distribuidos orientados a servicios Web. No obstante, el conjunto de primitivas especificado en BPCL cumple con la semática axiómatica formal definida en dicho lenguaje, lo cual permite comprobar la correctitud de las descripciones escritas en BPCL. El motor de ejecución para un proceso de negocios escrito en BPCL será el encargado de descubrir la ubicación de los recursos e invocarlos oportunamente. Los detalles de su funcionamiento son discutidos a profundidad en el capítulo 4. Veamos entonces las características de BPCL.

3.2. Características

Todo lenguaje composicional debe brindar un conjunto de características deseables para poder ejecutar eficientemente la orquestación de servicios, en particular BPCL provee las siguientes:

- *Simplicidad y abstracción.* En BPCL se omiten los detalles de ubicación de los recursos (transparencia de localidad), es decir, no es necesario especificar dónde se encuentra la descripción de cada servicio sino sólo la entidad que lo provee. Dicha característica hace que la especificación de un proceso sea más sencilla y concreta, facilitando así su enlace dinámico.

- *Enlace dinámico.* BPCL ofrece un mecanismo que permite vincular a los proveedores de servicios con sus respectivas ubicaciones a través del descubrimiento de servicios.

- *Soporte de actividades primitivas y estructuradas.* Cada una de las directivas de coordinación pueden verse como actividades, las cuales son necesarias tanto en la comunicación entre servicios como en el control semántico del flujo de

trabajo. Una actividad primitiva (simple) puede verse como aquella que provee una funcionalidad elemental, en contraste con las actividades estructuradas que controlan el flujo de trabajo desde una perspectiva global, especificando qué actividades serán ejecutadas y en qué orden.

- *Persistencia y correlación.* BPCL brinda un mecanismo mediante el cual pueda mantenerse un estado consistente de las variables y mensajes (síncronos o asíncronos) involucrados durante la coordinación del flujo de actividades.

3.3. Directivas de coordinación

Los elementos básicos que constituyen cualquier proceso de negocios a ser orquestado son: procesos, actividades, flujo de datos y flujo de control. Los procesos representan conjuntos de actividades, donde una actividad puede verse como una regla o función bien definida del proceso de negocios y éstas pueden ser primitivas (simples) o estructuradas. El flujo de datos describe la información intercambiada entre actividades y, por otro lado, el flujo de control especifica cómo las actividades serán ejecutadas en términos de construcciones usuales de programación como la composición secuencial, concurrente y condicional. En la figura (3.1) podemos observar el conjunto general de actividades soportadas por BPCL.

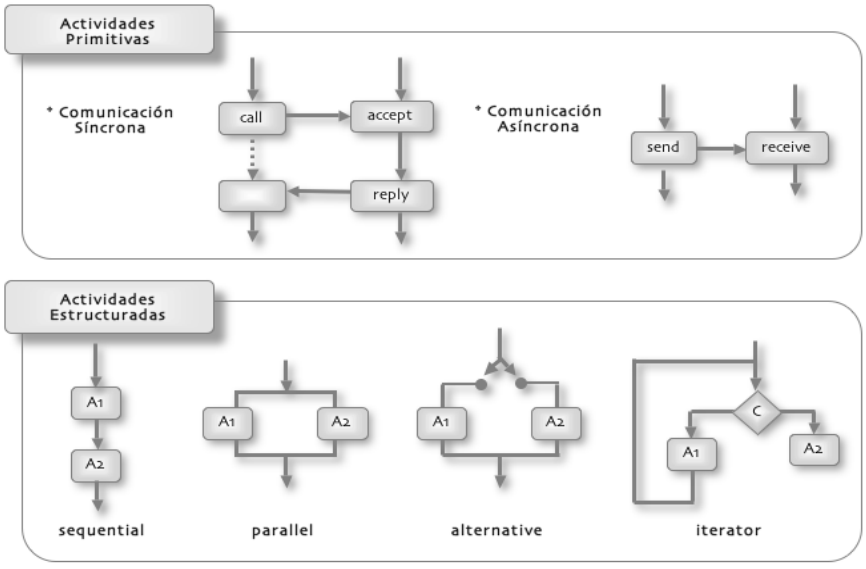

Figura 3.1: Tipos de actividades en BPCL.

Al ser BPCL un lenguaje basado en XML, cualquier proceso deberá ser especificado a través de un conjunto de etiquetas. BPCL clasifica dicho conjunto en etiquetas de datos, de comunicación y de coordinación. A continuación analizaremos a detalle cada una de las primitivas de coordinación soportadas por BPCL de acuerdo a esta clasificación.

3.3.1. Etiquetas de datos

Este tipo de etiquetas se utilizan para definir un proceso de negocios, el conjunto de servicios Web (participantes) involucrados y las variables locales asociadas a cada uno de los métodos orquestados descritos en la especificación.

En el cuadro (3.1) podemos observar cuáles son las directivas BPCL que pertenecen a esta clasificación.

Etiqueta	Prototipo	Descripción
process	`<process name="" javalib="" xmlns:xs="http://.../XMLSchema" xsi:noNamespaceSchemaLocation=""> </process>`	Permite definir un proceso BPCL, ésta contendrá como subelementos a todas las directivas de coodinación asociadas al flujo de trabajo. Como atributos será necesario especificar el nombre del proceso que se orquestará (name), las librerías en Java que den soporte a métodos locales (javalib) y el XMLScheme asociado a cualquier especificación BPCL (xsi:noNamespaceSchemaLocation).
partner	`<partners> <partner name="" type=""/> </partners>`	Permite definir cada uno de los participantes (proveedores de servicios Web) involucrados en la coordinación. Por medio de etiquetas hijas partner se especificará su nombre y su tipo (necesario para descubrir el servicio respectivo).
variable	`<variables> <variable name="" type=""/> </variables>`	Mediante esta etiqueta es posible definir la lista de variables locales a cada método orquestado definido en el proceso.

Tabla 3.1: Etiquetas de datos en BPCL.

3.3.2. Etiquetas de comunicación

Controlan el flujo de datos intercambiado entre las actividades del flujo. Mediante este tipo de etiquetas es posible invocar métodos de servicios Web o locales al proceso orquestado, definen los mensajes síncronos o asíncronos intercambiados entre los participantes. La comunicación síncrona involucra enviar una petición por un método de un servicio y esperar por su respuesta, mientras que la comunicación asíncrona sólo involucra enviar peticiones sin esperar algún valor de retorno.

En el cuadro (3.2) y (3.3) se muestran las directivas BPCL que pertenecen a esta clasificación.

Etiqueta	Prototipo	Descripción

accept	`<accept partner="" operation="">`
	`  <input>`
	`  <part name="" type=""/>`
	`  </input>`
	`</accept>`
		El proceso de negocios atiende métodos síncronos orquestados a través de esta primitiva, la cual podrá ser invocada a través de la actividad call. Mientras la operación apropiada no sea recibida el proceso permanecerá bloqueado. Cabe señalar que será necesario especificar cuáles serán los parámetros de entrada y sus tipos de datos respectivos.

reply	`<reply partner="" operation=""` `  variable=""/>`	Esta actividad es utilizada para enviar una respuesta a una petición previamente aceptada por la primitiva accept. Por la combinación accept, reply una operación síncrona puede ser modelada.

receive	`<receive partner="" operation="">`
	`  <input>`
	`  <part name="" type=""/>`
	`  </input>`
	`</receive>`
		Funciona de manera similar que accept sólo que esta primitiva es utilizada para que el proceso de negocios atienda métodos asíncronos orquestados, mismos que serán invocados por medio de la primitiva send.

Tabla 3.2: Etiquetas de comunicación para atender peticiones en BPCL.

Etiqueta	Prototipo	Descripción

call	`<call partner="" operation="">`
	`  <input>`
	`  <part name=""/>`
	`  </input>`
	`  <output>`
	`  <part name=""/>`
	`  </output>`
	`</call>`
		Primitiva utilizada para invocar un método síncrono de un servicio Web a través del proveedor de servicios correspondiente. Esta primitiva permanecerá en estado de espera hasta que la respuesta de su petición sea recibida. Dicha petición será atendida por la primitiva accept. Además, será necesario especificar sus parámetros de entrada y la variable donde será depositado su valor de retorno.

send	`<send partner="" operation="">`
	`  <input>`
	`  <part name=""/>`
	`  </input>`
	`</send>`
		Permite invocar un método asíncrono de un servicio Web a través del proveedor de servicios correspondiente. Send a diferencia de call no bloquea la ejecución del proceso. Una solicitud realizada por send será atendida por medio de la primitiva receive. La especificación de los parámetros de entrada respectivos serán necesarios.

execute	`<execute operation="">`
	`  <input>`
	`  <part name=""/>`
	`  </input>`
	`  <output>`
	`  <part name=""/>`
	`  </output>`
	`</execute>`
		Mediante esta directiva es posible invocar un método local al proceso, dicho método deberá ser definido en la correspondiente especificación BPCL.

Tabla 3.3: Etiquetas de comunicación para invocar servicios Web en BPCL.

3.3.3. Etiquetas de coordinación

Dichas etiquetas especifican el orden de ejecución de las actividades mediante estructuras convencionales de flujo de control.

En el cuadro (3.4) son mostradas las directivas BPCL que pertenecen a esta clasificación.

Etiqueta	Prototipo	Descripción
sequential	<sequential> act1,act2,..,actn </sequential>	Ejecuta una lista de actividades secuencialmente, es decir, el término de una actividad determina el inicio de la siguiente en la lista. Por ejemplo, para que la act_2 pueda ser procesada, antes debió terminar la ejecución de act_1.
parallel	<parallel> act1,act2,..,actn </parallel>	Define una lista de actividades que será ejecutada en paralelo. Es decir, el inicio de las actividades se hará simultáneamente, mientras que la terminación de la lista determinará el término de parallel.
alternative	<alternative> act1,act2,..,actn </alternative>	Define una lista de actividades que será ejecutada de manera concurrente, pero a diferencia de parallel esta primitiva sólo permitirá sobrevivir a la primera actividad que término su ejecución. Por ejemplo, si la actividad act_2 terminara su ejecución primero, las actividades $act_1, act_3, \cdots, act_n$ no terminarían su procesamiento.
iterator	<iterator variable="" initial="" final="" cond="" next="">act1,act2,..,actn </iterator>	Ejecuta una lista de actividades mientras la condición del ciclo sea verdadera. En esta primitiva es necesario especificar la variable de iteración, su valor inicial y final, la condición a ser evaluada y el tipo de salto entre iteraciones (incremento o decremento).
scope	<scope> act1,act2,..,actn </scope>	Permite definir un método orquestado con sus variables locales y su respectivo flujo de actividades. La descripción del mismo es posible definirla a través de actividades accept y receive.
localOperation	<localOperation> <operation> </operation> </localOperation>	Mediante esta primitiva es posible definir métodos locales al proceso orquestado. La definición de dicho método debe ser escrita directamente en Java ya que el motor de ejecución fue implementado en este lenguaje.

Tabla 3.4: Etiquetas de coordinación en BPCL.

3.4. Esquema del lenguaje

Siempre que se ha definido un nuevo conjunto de etiquetas en documentos XML, es necesario contar con algún tipo de especificación adicional que permita verificar sintácticamente la validez o correctitud de cualquier descripción que ha sido escrita en base a dicho conjunto XML.

Existen dos alternativas para definir la estructura válida para documentos XML, DTD's (Document Type Definition) y XSD's (XML Scheme Definition). Una de las principales desventajas que presentan las DTD's es que es necesario conocer una especificación adicional a XML para poder representarla. Sin embargo, con los XSD's es mucho más sencillo realizarla debido a que éstos utilizan la sintaxis propia de XML, son procesables al igual que cualquier documento XML y son más fáciles de aprender.

En un esquema XML es posible definir: (a) los elementos que pueden aparecer en un documento y sus atributos, los tipos de datos correspondientes y los valores que pueden tomar por defecto, (b) cómo pueden anidarse, es decir, si existe un orden de aparición restringido, (c) el número permitido de elementos hijos y por último (d) si un elemento puede ser vacío o no.

Los esquemas se definen en un documento con extensión .XSD, por tal razón, las descripciones XML basadas en un esquema deben incluir una referencia a este tipo de documentos. Además, los elementos utilizados en la creación de un esquema deben cumplir con la especificación de un espacio de nombres estándar, por ejemplo, http://www.w3.org/2001/XMLSchema. En cualquier proceso escrito en BPCL el encabezado típico sería el mostrado en la figura (3.2).

```
<?xml version="1.0"?>
<process name="" javalib=""
     xmlns:xs="http://www.w3.org/2001/XMLSchema"
     xsi:noNamespaceSchemaLocation="bpcllanguage.xsd">
...
</process>
```

Figura 3.2: Encabezado típico para un proceso BPCL.

El tipo de elementos que pueden definirse en un esquema son:

- Simples. Aquellos que sólo contienen texto y no pueden tener atributos.
- Complejos. Aquellos que contienen a otros elementos hijos y pueden tener atributos. Éstos suelen dividirse en 4 tipos: elementos vacíos, no vacíos con atributos,

con elementos hijos, con elementos hijos y valor propio.

En la figura (3.3) podemos observar un fragmento general del esquema asociado a procesos BPCL (para revisar la especificación completa consultar el apéndice A). En dicho esquema se han definido las construcciones válidas que pueden ser utilizadas para describir flujos de actividades y a groso modo las reglas que obedece son las siguientes:

```
1:  <?xml version="1.0"?>
2:  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
3:
4:  <xs:element name="process" type="processType"/>
5:
6:  <xs:complexType name="processType" >
7:    <xs:sequence>
8:
9:      <xs:element name="partners"  minOccurs="1" maxOccurs="1">
10:       <xs:complexType>
11:         <xs:sequence>
12:           <xs:element name="partner">
13:             <xs:complexType>
14:               <xs:attribute name="name" type="xs:string" use="required"/>
15:               <xs:attribute name="type" type="xs:string" use="required"/>
16:             </xs:complexType>
17:           </xs:element>
18:         </xs:sequence>
19:       </xs:complexType>
20:     </xs:element>
21:
22:     <xsd:choice>
23:       <xs:element name="sequential"  type="seqControlFlowType"/>
24:       <xs:element name="alternative" type="altControlFlowType"/>
25:     </xsd:choice>
26:
27:     <xs:element name="localOperation"  maxOccurs="1">
28:       <xs:complexType>
29:         <xs:sequence>
30:           <xs:element name="operation"/>
31:         </xs:sequence>
32:       </xs:complexType>
33:     </xs:element>
34:
35:   </xs:sequence>
36:   <xs:attribute name="name"    type="xs:string" use="required"/>
37:   <xs:attribute name="javalib" type="xs:string"/>
38: </xs:complexType>
```

Figura 3.3: Fragmento del esquema para BPCL.

- El primer elemento que debe definirse es process (línea 4) con el atributo obligatorio name (línea 36) y el atributo opcional javalib (línea 37).

- Dentro de un elemento process deben establecerse los participantes involucrados en el proceso a través de la etiqueta partners, la cual contiene nodos hijos partner que permiten declarar a cada uno de ellos. Los modificadores *minOccurs*="1" y *maxOccurs*="1" indican que la etiqueta partners es obligatoria y sólo aparecerá en el documento una vez (líneas 9-20).

- Enseguida deben declararse las primitivas que representen el orden en que serán aceptadas las peticiones de clientes. Dentro de éstas es donde se especificarán las etiquetas de datos, comunicación y coordinación que constituyan al flujo orquestado. Además, deben especificarse sus atributos y la forma en que éstas pueden componerse o anidarse. Mediante el indicador *choice* es posible restringir que las peticiones serán recibidas de manera secuencial (sequential) o alternativa (alternative) (líneas 22-25).

- Finalmente, es posible definir una lista de operaciones locales al proceso a través de la etiqueta localOperation que contiene elementos hijos de tipo operation (líneas 27-33).

Como hemos podido apreciar los XSD's pueden brindarnos gran ayuda en el análisis sintáctico de documentos XML. Sin embargo, si deseamos verificar que sean válidos los datos asociados a cada una de las etiquetas permitidas en el lenguaje, será necesario hacer uso de otro tipo de técnicas que nos permitan restringirlo.

3.5. Modelo de comunicación

Un programa es llamado concurrente cuando permite que dos o más procesos sincronizados (en nuestro caso actividades sincronizadas) trabajen cooperativamente. Independientemente del mecanismo de comunicación y sincronización utilizado entre procesos, un lenguaje de programación concurrente deberá proveer las primitivas adecuadas para la especificación e implementación de dicho programa.

Es por éso, que antes de definir a detalle el modelo de comunicación utilizado para representar peticiones cliente-servidor en BPCL, es necesario señalar algunas consideraciones en torno a las estructuras de programación soportadas. Su clasificación y notación es la siguiente:

- *Variables*. Representadas por la primitiva variable a la cual debe ser asociado un nombre y tipo.

- *Estructuras de asignamiento*. Este tipo de estructuras se representa implícitamente cuando se invoca una llamada síncrona (local o remota) donde el valor de retorno se asigna a un variable predefinida.

- *Estructuras de orden*. Pertenecen a esta clasificación las primitivas:

  (a) <sequential> $act_1, act_2, \cdots, act_n$ </sequential> que permite ejecutar una lista de actividades $act_1, \cdots, act_n$ de manera secuencial, es decir, la act_2 podrá ser ejecutada solamente hasta que el procesamiento de la act_1 sea concluido, y

  (b) <parallel> $act_1, act_2, \cdots, act_n$ </parallel> la cual ejecuta una lista de actividades $act_1, \cdots, act_n$ concurrentemente. El efecto de ejecutar paralelamente dicha lista es equivalente a intercalar la ejecución de las operaciones que definan a cada actividad y garantizar que todas las operaciones sean realizadas. Sin embargo, no todos las formas de ordenamiento son aceptables, por eso es necesario establecer mecanismos de sincronización como *exclusión mutua* o *por condición* para prevenir intercalamientos no deseables.

- *Estructuras condicionales*. Primitivas pertenecientes a esta clasificación son:

  (a) <iterator C> $act_1, act_2, \cdots, act_n$ </iterator> utilizada para ejecutar una lista de actividades $act_1, \cdots, act_n$ mientras la condición C a ser evaluada sea verdadera y

  (b) <alternative> $act_1, act_2, \cdots, act_n$ </alternative> dicha primitiva permite lanzar una lista de actividades $act_1, \cdots, act_n$ en forma paralela, pero a diferencia de parallel esta primitiva sólo permitirá sobrevivir a la primera actividad que termine su ejecución, suspendiendo a las actividades restantes. Dicha primitiva es condicional ya que existe una condición implícita a ser evaluada, es decir, se verificará el momento en que se obtenga la primera respuesta de alguna actividad de la lista, para que en tal caso todas las demás actividades sean interrumpidas. Por ejemplo, si la actividad act_2 terminará su ejecución primero, las actividades $act_1, act_3, \cdots, act_n$ no terminarían su procesamiento.

Ahora bien, los programas concurrentes que se comunican a través de mensajes se denominan programas distribuidos. Ésto supone la ejecución sobre una arquitectura de memoria distribuida, aunque también puedan ser ejecutados sobre una arquitectura de memoria compartida.

El modelo cliente-servidor es el esquema dominante en las aplicaciones de procesamiento distribuido en Internet. Siendo el proceso servidor aquel participante que esperará por solicitudes de múltiples clientes. Los mecanismos de comunicación entre ambas entidades son variados, sin embargo, los que mejor se acoplan a dicho modelo son RPC (*Remote Procedure Call*) y *Rendezvous*. La comunicación entre procesos concurrentes servirá para indicar el modo en que se organizan y se trasmiten los datos entre tareas concurrentes. Esta organización requerirá especificar los protocolos para controlar la correctitud y el progreso de la comunicación [29].

La comunicación se logra a través del paso de mensajes síncronos (bloqueantes) o asíncronos (no bloqueantes), ya que mediante ellos será posible establecer un canal (lógico o físico) para trasmitir información entre procesos. Los canales pueden proveer comunicación unidireccional o bidireccional.

La ventaja de RPC y *Rendezvous* es que una operación utiliza dos canales de comunicación, uno del proceso invocador al proceso servidor y del proceso servidor al invocador. En particular, el proceso invocador esperará hasta que la operación solicitada sea completamente ejecutada para así conocer su valor de retorno. La diferencia entre RPC y *Rendezvous* es la manera en que las invocaciones son atendidas. En RPC un proceso intermediario se crea para ejecutar las invocaciones, mientras que en *Rendezvous* el proceso que atiende una petición es el mismo que la ejecuta. Veamos ahora a detalle como funciona cada propuesta [29].

3.5.1. RPC

La noción de llamadas a procedimientos remotos (*Remote Procedure Calls*, RPC) [29] se introdujo con el objetivo de ofrecer un mecanismo estructurado de alto nivel para realizar la comunicación entre procesos pertenecientes a sistemas distribuidos. Mediante RPC un proceso de un sistema (proceso invocador) podrá solicitar un procedimiento de un proceso servidor. Dicho esquema de comunicación se muestra en la figura (3.4) y funciona de la siguiente manera.

El proceso que llama enviará un mensaje al proceso servidor y a continuación esperará (bloqueará su ejecución) hasta recibir la respuesta de su petición. El mensaje de llamada deberá contener los parámetros del procedimiento invocado y el mensaje de respuesta el resultado de la invocación del mismo.

Mientras tanto, en el lado del servidor, éste esperará por la petición del proce-

dimiento. Cuando dicha petición sea recibida, el servidor creará un nuevo hilo de ejecución que se encargará de extraer los parámetros correspondientes de la llamada, ejecutarla y devolver el control al proceso servidor. Dicho proceso enviará como respuesta el valor de retorno al proceso invocador. Cuando el mensaje sea retornado, el proceso que llama finalizará su espera, extraerá el valor de retorno y continuará su ejecución.

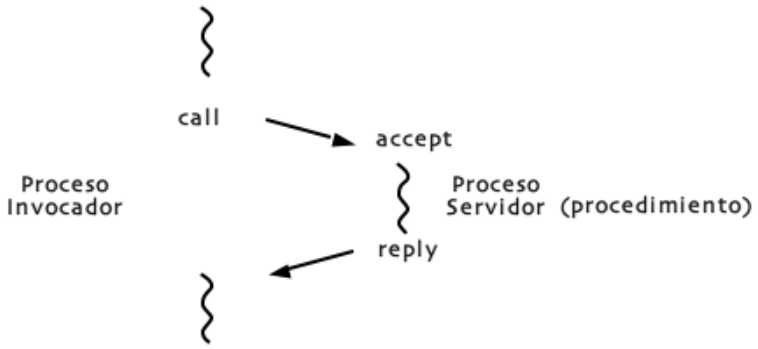

Figura 3.4: Modelo de comunicación RPC.

El servidor puede procesar peticiones mediante diferentes políticas de implementación (*multithreading*): creando un hilo de ejecución (*thread*) para todas las peticiones de todos los clientes, un hilo para todas las peticiones de un cliente o bien creando un nuevo hilo para cada petición de cada cliente, esta última es la más utilizada porque provee de mayor grado de independencia entre las invocaciones realizadas.

En RPC el componente de programación utilizado es un proceso el cual contiene la definición de procedimientos. En la siguiente figura (3.5) podemos observar el esquema general para definir procesos en BPCL, cómo es declarado un procedimiento remoto y cómo se define el cuerpo del mismo.

Mediante la primitiva *call* es posible realizar la invocación de un procedimiento definido en un proceso cliente, mientras que en un proceso servidor a través de la primitiva *accept* será atendida dicha petición y mediante *reply* se dará a conocer el valor de respuesta.

Una desventaja de este esquema es que no es posible condicionar el orden de ejecución de las llamadas realizadas. Ésto significa que los procedimientos definidos

(a)

```
<process name="Cliente"
   xmlns:xs="http://.../XMLSchema"
   xsi:noNamespaceSchemaLocation="">

  <!- Declaracion de participantes ->

  <!- Declaracion de invocaciones ->
  <call partner="" operation="">
    <input>
      <part name=""/>
    </input>
    <output>
      <part name=""/>
    </output>
  </call>

  ...

</process>
```

(b)

```
<process name="Servidor" javalib=""
   xmlns:xs="http://.../XMLSchema"
   xsi:noNamespaceSchemaLocation="">

  <!- Declaracion de participantes ->

  <!- Declaracion de procedimientos visibles ->
  <accept partner="" operation="">
    <input>
      <part name="" type=""/>
    </input>
  </accept>

      <!- Declaracion de variables locales ->
      <!- Instrucciones ->

  <reply partner="" operation="" variable=""/>
  ...
  <!-Declaracion de procedimientos locales->
</process>
```

Figura 3.5: (a) Descripción de un proceso cliente y un (b) proceso servidor en BPCL utilizando el modelo RPC.

en un proceso pueden invocarse en cualquier orden siempre y cuando pertenezcan a dicho proceso. Debido a esta restricción en nuestra propuesta de trabajo utilizamos el modelo de comunicación RPC para realizar invocaciones a métodos de servicios Web elementales, ya que éstos se definen dentro del servicio (proceso) en el cual su invocación no tiene que cumplir un orden determinado como lo es en el caso de servicios orquestados.

Finalmente, ya que el proceso invocador queda bloqueado durante la ejecución del servicio, el modelo RPC puede extenderse para implementar paso de mensajes asíncronos, como fue realizado en nuestra propuesta.

3.5.2. Rendezvous

Rendezvous [29] es un modelo de comunicación y sincronización que permite a dos procesos concurrentes, el solicitante (cliente) y el llamado (servidor) intercambiar datos de forma coordinada. El esquema de comunicación se muestra en la figura (3.6).

El proceso que solicita un método debe esperar en el punto de reencuentro (*rendezvous*) hasta que el proceso llamado llegue allí. Sin embargo, el proceso llamado puede llegar a dicho punto antes que el solicitante y entonces él deberá esperar de igual forma para poder continuar su ejecución. Lo anterior significa que existe la posi-

bilidad de que ambos procesos ejecuten algún otro procesamiento antes y después del punto de reencuentro, lo cual en el modelo RPC no es posible en el lado del servidor.

En el momento de reencuentro los procesos pueden intercambiar datos. Los datos intercambiados corresponden a parámetros de una llamada (del cliente hacia el servidor) y a resultados de dicha llamada (del servidor hacia el cliente) sin necesidad de almacenamiento intermedio en el servidor. Es decir, a diferencia de RPC en este esquema no es necesario crear un *thread* intermedio que realice la ejecución del procedimiento deseado.

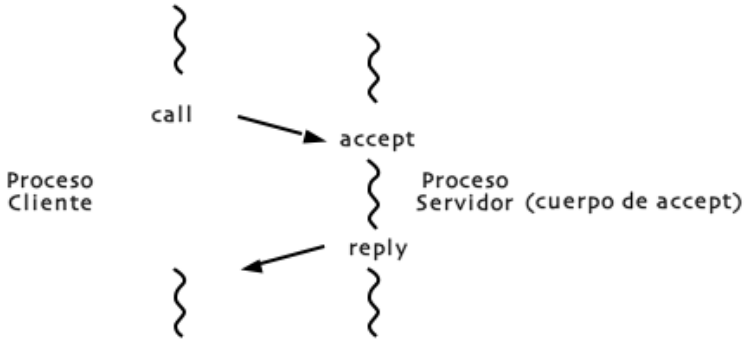

Figura 3.6: Modelo de comunicación *Rendezvous*.

En *Rendezvous* el componente de programación es un proceso y es ahí donde se exportan las operaciones que pueden invocarse. En la figura (3.7) se muestra el formato para realizar la declararación de procesos. A diferencia del modelo RPC, ahora en la definición de un proceso servidor puede describirse el orden en que serán aceptadas las peticiones y el orden de las instrucciones que constituyen a dichas peticiones.

En un proceso servidor se debe especificar su nombre, el conjunto de proveedores de servicios, el conjunto de operaciones (locales y visibles) que lo constituyen, sus respectivas variables y el orden en que las operaciones serán atendidas (*flujo de control de actividades*). Esta última propiedad permite emplear instrucciones de recepción selectiva o condicionada.

Un cliente puede invocar una operación definida en un proceso a través de la primitiva *call*, mientras que el servidor puede aceptar dicha operación a través de la primitiva *accept*. En esta primitiva, el *flujo de control de instrucciones* representa la

(a)

```
<process name="Cliente"
   xmlns:xs="http://.../XMLSchema"
   xsi:noNamespaceSchemaLocation="">

  <!- Declaracion de participantes ->

  <!- Declaracion de invocaciones ->
  <call partner="" operation="">
    <input>
      <part name=""/>
    </input>
    <output>
      <part name=""/>
    </output>
  </call>

  ...

</process>
```

(b)

```
<process name="Servidor" javalib=""
   xmlns:xs="http://.../XMLSchema"
   xsi:noNamespaceSchemaLocation="">

  <!- Declaracion de participantes ->

  <!- Flujo de control de actividades ->

    <!- Declaracion de procedimientos visibles ->
    <accept partner="" operation="">
      <input>
        <part name="" type=""/>
      </input>
    </accept>

        <!- Declaracion de variables locales ->
        <!- Flujo de control de instrucciones ->

    <reply partner="" operation="" variable=""/>
    ...
  <!-Declaracion de procedimientos locales->
</process>
```

Figura 3.7: (a) Descripción de un proceso cliente y un (b) proceso servidor en BPCL utilizando el modelo *Rendezvous*.

distribución en la cual las instrucciones que componen a una operación serán procesadas, por ejemplo, en forma secuencial, paralela o iterativa.

No obstante, *Rendezvous* también permite restringir el orden en que serán atendidas dichas operaciones. En la figura (3.8.a) podemos observar que las operaciones pueden atenderse en forma secuencial. Es decir, para que una operación op_j pudiera atenderse, primero tendría que haberse solicitado la operación op_i, con $i < j$. En contraste con la figura (3.8.b) donde las operaciones serán procesadas en forma concurrente. Sólo que en este caso la primera operación que termine su ejecución podrá dar a conocer su valor de respuesta, suspendiendo así todas las operaciones restantes.

Finalmente, *Rendevous* fue elegido en nuestra propuesta de trabajo como modelo de comunicación entre procesos orquestados, ya que en este caso sí es necesario disponer de un mecanismo que nos permita restringir el orden en que serán ejecutadas las operaciones que constituyen a un proceso, como se requiere en la orquestación de servicios. Además, cabe señalar que la notación que describimos al inicio de la sección 3.5 es la utilizada para describir dichas operaciones y que al igual que sucede en RPC es posible combinar el esquema de paso de mensajes asíncronos a través de las primitivas **send** y **receive** para extender la funcionalidad del modelo *Rendezvous*.

(a)

```
<sequential>

  <accept operation="opi"/>
     act1i, act2i, ... , actni
  <reply/>

  <accept operation="opj"/>
     act1j, act2j, ... , actnj
  <reply/>

  ...

  <accept operation="opk"/>
     act1k, act2k, ... , actnk
  <reply/>

</sequential>
```

(b)

```
<alternative>

  <accept operation="opi"/>
     act1i, act2i, ... , actni
  <reply/>

  <accept operation="opj"/>
     act1j, act2j, ... , actnj
  <reply/>

  ...

  <accept operation="opk"/>
     act1k, act2k, ... , actnk
  <reply/>

</alternative>
```

Figura 3.8: Generalización de la primitiva *accept* en BPCL utilizando el modelo *Rendezvous*.

3.6. Sumario

BPCL representa una alternativa simplificadora en los lenguajes de coordinación existentes (como BPEL4WS y BPEL4J), al permitir al usuario concentrar su atención en la descripción del flujo de coordinación y no en la especificación de la localización de los recursos.

Gracias a ésto, un usuario podrá especificar un proceso de negocios como si éste dispusiera de los recursos localmente, es decir, en la forma más simple y usual de trabajo. Así, por parte de los desarrolladores, la adopción de este nuevo lenguaje será más sencillo y natural.

Capítulo 4

Infraestructura de coordinación

En el capítulo anterior nos enfocamos en el primer objetivo de nuestra propuesta de trabajo: introducir un nuevo lenguaje composicional por medio del cual la descripción de un proceso orquestado pudiera realizarse sin necesidad de especificar detalles de distribución de recursos. Así, ahora toca el turno de profundizar en nuestro segundo objetivo, el cual fue crear una infraestructura de coordinación que proporcionara el conjunto de elementos o servicios que se consideran necesarios para la creación y funcionamiento de procesos de negocios orquestados. En este capítulo abordaremos todos los detalles del diseño e implementación de la infraestructura de coordinación.

4.1. Descripción general

La infraestructura de coordinación propuesta permite generar un servicio Web orquestado a partir de su especificación en BPCL y su descripción WSDL, esta última requerida para realizar la publicación del servicio y permitir así que éste se encuentre disponible en futuras peticiones. Utilizamos las siguientes herramientas: WebSphere SDK for Web Services (WSDK) [12], RMI [30] y AspectJ [20] como base para construir su implementación.

Así mismo, una vez que el servicio ha sido debidamente generado, instalado e inicializado, la infraestructura de coordinación provee un mecanismo para realizar la búsqueda e invocación de servicios orquestados, cabe señalar que este mecanismo es muy similar al utilizado internamente por el sistema para consultar e invocar servicios Web elementales.

A continuación explicaremos con mayor detalle el diseño y la implementación de cada uno de los módulos que conforman nuestra infraestructura de coordinación y concluiremos este capítulo con un caso de estudio que puede ser aplicado a la misma.

4.2. Diseño

4.2.1. Esquema conceptual

Un esquema usual de interacción entre aplicaciones cliente-servidor se muestra en la figura (4.1). Este esquema hace posible que un usuario solicite y obtenga una respuesta de cualquier servicio provisto por un servidor. No obstante, otra importante característica deseable es que la forma de trabajo (local o remota) del mismo sea tan transparente como sea posible al cliente. De esta forma se conseguirá mayor claridad y simplicidad al momento de desarrollar aplicaciones que implementen dicho esquema.

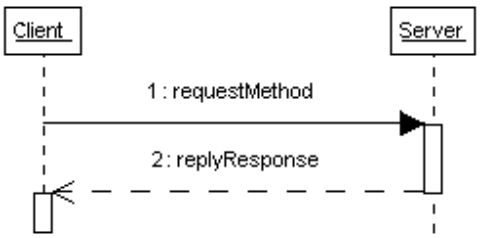

Figura 4.1: Esquema tradicional de comunicación cliente-servidor.

La infraestructura de coordinación propuesta pretende brindar las dos características básicas del esquema anterior. Para logralo el diseño de ésta fue modelada en dos esquemas. El primero de ellos tiene como objetivo realizar la búsqueda de servicios (obtener su descripción), en la figura (4.2) podemos observar que dicha operación está a cargo de la infraestructura, lo cual evita que un cliente tenga conocimiento de la localización de los recursos y promueve que sólo se preocupe por solicitar el servicio deseado.

El segundo esquema se muestra en la figura (4.3), éste realiza la invocación de servicios y trabaja de la siguiente manera. Cuando un cliente realiza una petición a un método de un servicio Web orquestado, este último redireccionará su petición a un servidor RMI. Dicho servidor será el encargado de llevar a cabo la coordinación de

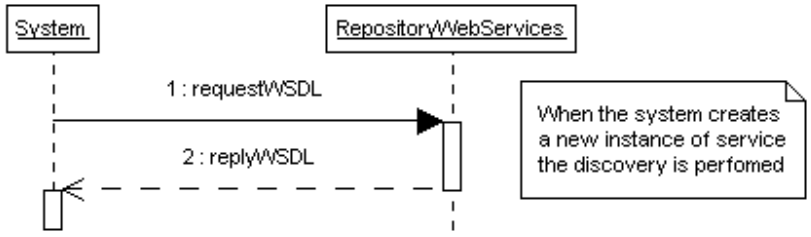

Figura 4.2: Esquema conceptual para la búsqueda de servicios Web.

cada una de las actividades especificadas en el lenguaje, ésto quiere decir que establecerá la comunicación con cada uno de los proveedores de servicios Web elementales. Una vez que éste termine de realizar la coordinación, devolverá el control al servicio Web, el cual tendrá entonces la capacidad de brindar una respuesta a la petición del cliente.

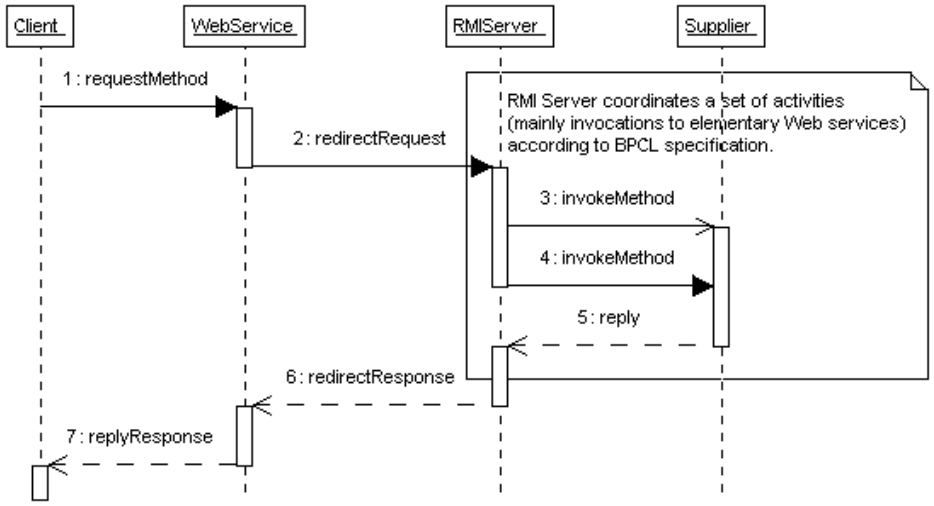

Figura 4.3: Esquema conceptual para la invocación de servicios Web.

Al analizar detenidamente el esquema conceptual resulta un tanto díficil entender el por qué de usar un servidor RMI, siendo que éste podría ser omitido si dejaramos la orquestación a cargo del servidor Web directamente. Su justificación va de la mano con WSDK que es la herramienta que utilizamos para el desarrollo de servicios Web y con la forma en que fue implementada la orquestación en nuestra infraestructura.

De forma tal que pudiese ser vista una actividad como un hilo de ejecución independiente, manejamos las actividades como *threads* y dentro de ellas realizamos la invocación de servicios. Sin embargo, nos enfrentamos con el problema de que WSDK no soporta la creación dinámica de hilos. Por tal razón, decidimos delegar la responsabilidad de orquestación a un servidor interno RMI y a un servidor externo (servidor Web) la responsabilidad de establecer la comunicación con el cliente, de manera que pudiesemos gozar de todas las ventajas de un servicio Web, como son: contar con una descripción estándar y la capacidad de ser publicado y descubierto. Otro punto por el cual decidimos optar por este enfoque fue que WSDK tampoco brinda soporte para aplicar un *aspecto* a un método de un servicio Web, lo cual en nuestro caso resulta indispensable.

Por último, otra pregunta razonable sería por qué no utilizar otra herramienta de desarrollo para evitar estos problemas. Sin embargo, debemos aceptar que WSDK es una de las plataformas más amigables para el desarrollo de servicios, brinda muchas facilidades para la generación, instalación e inicialización de los mismos. Para nuestros fines ésto permitió concentrarnos en los mecanismos de orquestación más que en los de generación de éstos.

Veamos entonces los módulos de coordinación necesarios para brindar soporte a nuestro esquema de diseño.

4.2.2. Módulos de coordinación

La infraestructura de coordinación está compuesta por tres módulos principales, el primero de ellos se muestra en la figura (4.11), el cual tiene como objetivo permitir la publicación de un servicio Web orquestado. El proceso de publicación se descompone en dos etapas, la etapa de generación de código y la de registro del servicio orquestado.

En la etapa de generación, como primera instancia es necesario contar con la descripción WSDL del servicio y escribir un programa fuente en BPCL que defina el flujo de actividades a ser orquestadas. Estas especificaciones servirán como entrada al módulo de traducción. Éste será el encargado de verficar la validez de la descripción BPCL y de convertirla a su correspondiente definición Java. Para cada uno de los participantes involucrados en la orquestación (cliente, servicio Web, servidor RMI, proveedores de servicios) serán generados sus respectivos archivos de implementación y compilación.

La segunda etapa comienza una vez que el traductor genera el archivo de publicación ejecutable que contiene todos los comandos necesarios para que el registro de servicios se lleve a cabo. Dicho proceso es el encargado de crear, compilar, empaquetar, instalar, inicializar y publicar el servicio orquestado. Para realizarlo se utilizó: WSDK [12] con el fin de generar el entorno necesario para que la aplicación pueda verse como servicio Web, RMI [30] que permitirá orquestar el flujo de actividades y AspectJ [20] para realizar la búsqueda e invocación de servicios Web.

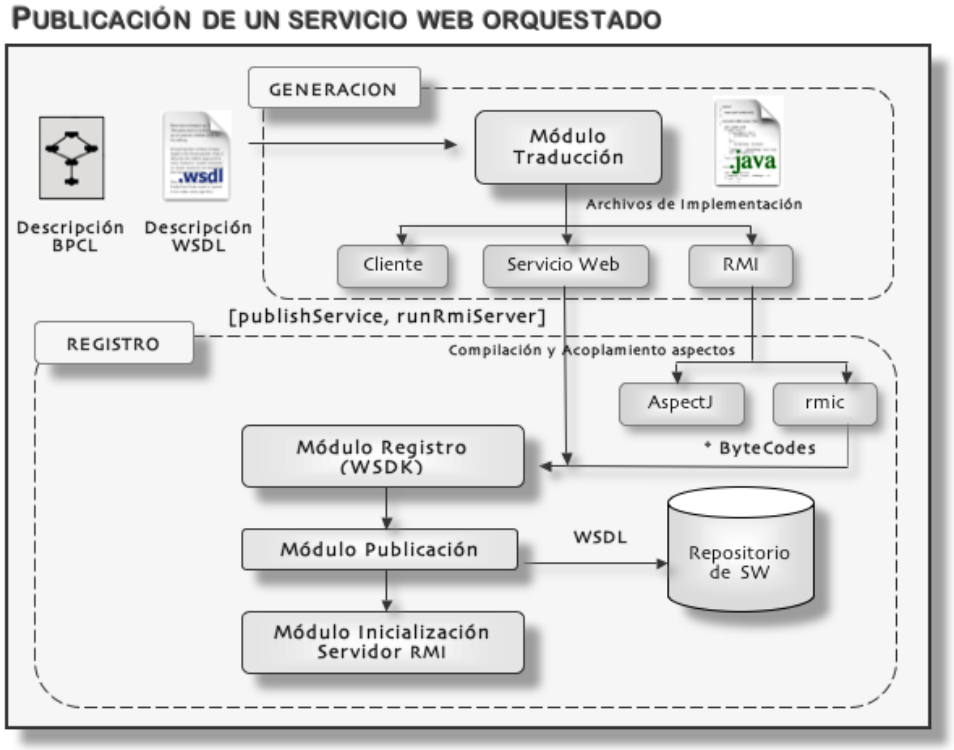

Figura 4.4: Módulos principales para la publicación de un servicio Web orquestado.

El segundo módulo se muestra en la figura (4.5). Éste tiene como objetivo descubrir un servicio que ha sido previamente almacenado en el repositorio de servicios orquestados.

El descubrimiento de servicios Web es indispensable durante la orquestación de servicios debido a que resulta necesario conocer la descripción de éstos para poder realizar su invocación y además porque son altamente suceptibles al cambio. Puede

darse el caso en que sean dados de alta en el repositorio nuevos servicios que provean la misma funcionalidad y sólo se diferencien por su ubicación, o que éstos sean modificados en su composición o localización, o en el peor de los casos que sean removidos del repositorio. Por tal razón se realiza su consulta en un *aspecto* que se activa cada vez que deba ser ejecutado el flujo de actividades orquestadas.

La búsqueda de un servicio (elemental u orquestado) puede realizarse al momento de invocar cada método del mismo, sin embargo, realizarlo así resultaría demasiado costoso porque consultaríamos redundantemente su descripción. En nuestra infraestructura la alternativa que elegimos fue realizar y encapsular esta operación en un *advice* de tipo *around* al momento de crear una instancia del servicio involucrado, con ésto conseguimos realizar sólo una búsqueda por servicio y no una por cada uno de sus métodos.

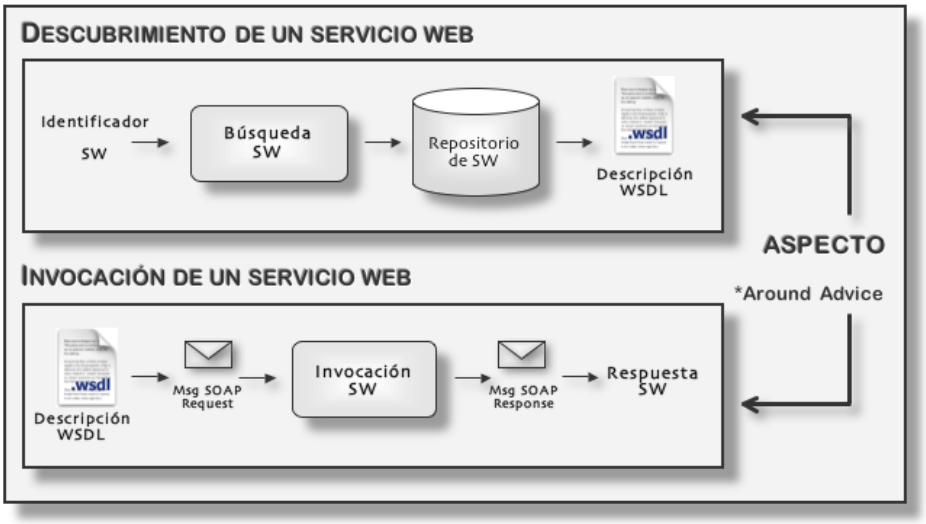

Figura 4.5: Proceso de búsqueda e invocación de un servicio Web.

El tercer módulo se muestra en la figura (4.5) y tiene como objetivo invocar servicios Web. Recordemos que una vez que se ha realizado el descubrimiento de éstos, contamos con su descripción WSDL y por consiguiente podemos invocarlos. Lo que resta entonces es generar los mensajes SOAP de petición necesarios para invocar métodos remotos y analizar el mensaje SOAP de respuesta para analizar y codificar correctamente la información retornada. Al igual que en el módulo anterior encapsulamos la solicitud de servicios como un *aspecto* en un *advice* de tipo *around*

que se activa en el momento en que se detecta la invocación de un método durante la coordinación. En los *aspectos* se implementan los mecanismos necesarios para verificar el momento en que la búsqueda e invocación no puedan llevarse a cabo, de tal manera que sea ejecutada una funcionalidad local definida por el desarrollador.

4.3. Implementación

4.3.1. Traductor del lenguaje

El objetivo de este módulo es proporcionar todos los mecanismos necesarios para mapear el flujo de actividades especificado en BPCL a sus correspondientes primitivas de coordinación en Java.

Para lograrlo el primer paso que realizamos fue diseñar un modelo mediante el cual se pudiera homegeneizar tipos de datos para poder anidar o componer actividades, consiguiendo así que una especificación BPCL fuera los más parecida a su escritura en código Java. En la figura (4.6) podemos observar que cada una de las primitivas de coordinación básicas fueron extendidas de una clase *Activity*, misma que hereda todas las características de la clase *Thread*, y que que las primitivas estructuradas se extienden de la clase *CollaborativeActivity*, resultando entonces que las actividades puedan verse como hilos de ejecución independiente.

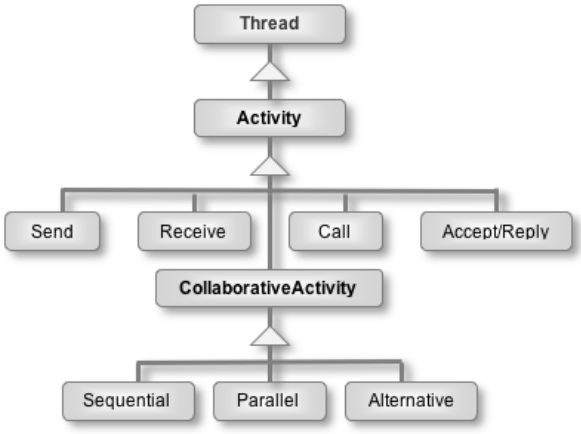

Figura 4.6: Composición de actividades.

El segundo paso realizado fue la creación de una librería de coordinación que pro-

porciona una implementación del modelo de datos antes citado, de los mecanismos de sincronización y control de concurrencia, mecanismos de comunicación entre las actividades y persistencia de datos. A esta librería la llamamos bpcl.jar (documentación detallada en el apéndice B), la cual consta de los elementos listados a continuación. Cabe señalar que estos elementos se clasificaron de acuerdo a la etapa en la cual dan soporte:

- **Etapa de Coordinación**
  - **Activity**: Representa una actividad descrita en BPCL, misma que puede ser iniciada o suspendida.
  - **CollaborativeActivity**: Define conjuntos de actividades por medio de un grupo de *threads*. Además permite suspender la ejecución cada uno de sus elementos con excepción de una actividad que sea solicitada.
  - **Sequential**: Ejecuta un grupo de actividades secuencialmente.
  - **Parallel**: Ejecuta un grupo de actividades de manera concurrente (en paralelo).
  - **Synchronizer**: Funciona al igual que un semáforo tradicional con sus respectivas primitivas de sincronización *down,up*. Esta clase sincroniza las actividades del lenguaje: call, accept/reply, send, receive.
  - **Transaction**: Encapsula la información de una invocación a un servicio Web como una transacción. De ésta forma es posible crear una instancia de su tipo con la cual se conozca el objeto que realizó la invocación, el nombre del método solicitado, sus parámetros y su valor de retorno.
  - **ProxyMethod**: Almacena un objeto de tipo *Transaction* al cual se asocian los semáforos necesarios para sincronizar las invocaciones de servicios.
  - **Proxy**: Hace posible que las peticiones de los participantes involucrados en la invocación de servicios Web sean coordinadas estrictamente en el orden en que fueron descritas en el lenguaje. Es importante señalar que cada método de un servicio se ejecuta a través del Reflection API de Java [31]. Con ésto se consigue realizar las invocaciones a través de un mecanismo uniforme, ya que es posible invocar cualquier método de un objeto que sea conocido hasta tiempo de ejecución.
  - **Variable**: Permite guardar y recuperar la información de un objeto en particular. Mediante esta clase es posible definir un tipo de dato uniforme asociado a las variables locales definidas en cada método orquestado, permitiendo además que sus valores puedan modificarse.

- **Etapa de Traducción**
  - **InformationWsdl**: Es posible obtener información relevante de un servicio Web mediante la lectura de su respectiva descripción WSDL.
  - **GeneratePublishOWS**: Verifica que la especificación escrita en BPCL sea válida. Mediante ésta se generarán todos los archivos de implementación y compilación necesarios para que una aplicación orquestada pueda verse como un servicio Web.
  - **PublishService**: Permite generar un archivo ejecutable mediante el cual son invocados una serie de comandos necesarios para publicar un servicio Web orquestado. Además registra el url de su descripción WSDL en el repositorio de servicios.
  - **MyRandom**: Genera números enteros aleatorios en un intervalo determinado y mediante ella el traductor asigna identificadores únicos a cada una de las actividades del flujo.
- **Etapa de Invocación**
  - **WSDLInvoker**: Clase que permite crear los mensajes SOAP necesarios para invocar las operaciones descritas en el documento WSDL de un servicio Web. Para lograrlo es necesario conocer algunos datos importantes de su descripción: URI del servicio, nombre de la operación, portName, etc.
  - **InvokeService**: Obtiene la información requerida en la clase anterior para invocar métodos de servicios. Además analiza el mensaje SOAP de respuesta para obtener sólo la información deseada.

Una vez que hemos descrito la base sobre la cual funcionará el traductor podemos entonces explicar su mecanismo de trabajo.

La información de entrada al módulo de tradución es la especificación BPCL del flujo de actividades y la descripción WSDL del servicio que será orquestado. El primer paso antes de realizar la traducción es verificar que el documento de entrada sea válido, que esté bien formado, es decir, que cumpla con el esquema del lenguaje (ver detalles en la sección 3.4). Esto se logró a través del verificador de documentos XML Parser del API JDOM de Java [32]. El segundo analizador sintáctico requerido es aquel que se encargue de validar que cada uno de los participantes solicite sólo operaciones que estén definidas en su descripción WSDL y que éstas cumplan con su respectiva especificación (nombre, tipos de parámetros y de retorno). Aunque, este analizador

no fue implementado por falta de tiempo, como trabajo futuro sería útil su desarrollo.

Después de someter el documento BPCL a un análisis sintáctico, el traductor ejecutará una serie de algoritmos de traducción para obtener todos los archivos necesarios que den soporte a la aplicación orquestada (al final de esta sección veremos a detalle estos archivos). La idea sobre la que fueron diseñados es muy similar, por tal razón, sólo abordaremos el funcionamiento del algoritmo de traducción para obtener el mapeo en Java del flujo de actividades especificado en BPCL.

Dicho algoritmo de traducción realiza una búsqueda recursiva a través de cada etiqueta de la especificación del flujo a orquestarse. En cada una de ellas se obtiene la información necesaria para parametrizar su correspondiente esquema de traducción. Se analiza cuándo en la recursión se encuentran etiquetas de entrada o de cierre y mediante un switch el traductor sabe qué código debe generar. Con ésto logramos que la especificación sea lo más similar a su representación en código Java y se facilite así la tarea de traducción. Este algoritmo se muestra en la figura (4.7).

```
1:   void generateCodeImplementation(Node node)
2:   {
3:     NodeList list = node.getChildNodes();
4:     for (int i=0; i < list.getLength(); i++)
5:     {
6:       parent = list.item(i).getParentNode();
7:       if(parent.getLastChild() == list.item(i))
8:       {
9:         /* Close labels */
10:        name = parent.getNodeName();
11:        switch(name)
12:        {
13:          case "accept": //Definition of local variables of a synchronous method.
14:          case "receive": //Definition of local variables of an asynchronous method.
15:          case "scope": //Close method. Hash table of variables and parameters are initialized.
16:          case "iterator": //Close for instruction.
17:          case "sequential": //A sequential activity is defined. Generation of name.
18:          case "parallel": //A parallel activity is defined. Generation of name.
19:        }
20:      }
21:
22:      /* Open labels */
23:      name = list.item(i).getNodeName();
24:      switch(name)
25:      {
26:        case "variables": //Local variables are stored in a hash table.
27:        case "accept": //A synchronous method is defined. Parameters are stored in a hash table.
28:        case "reply": //Return value of a synchronous method is defined.
29:        case "receive": //An asynchronous method is defined. Parameters are stored in a hash table.
30:        case "call": //A call activtiy is defined. Generation of name.
```

```
31:         case "send": //A send activtiy is defined. Generation of name.
32:         case "iterator": //Definition of a instruction for in Java.
33:         case "execute": //Invocation of a local function.
34:         case "localOperation": //Definition of a local function.
35:       }
36:       generateCodeImplementation(list.item(i));
37:     }
38:   }
```

Figura 4.7: Algoritmo de traducción para obtener la implementación en Java del flujo de actividades de una especificación BPCL.

En el esquema de traducción para una actividad *call* o *send*, la primera acción que debe realizarse es obtener la información de dichas etiquetas, es decir, conocer qué entidad provee el método que será invocado, cuáles serán sus parámetros y la variable en la cual será almacenado su valor de retorno (en el caso de la primitiva *call*). Una vez que se cuente con esta información, el traductor generará un identificador único para denominar a esta actividad. El identificador será almacenado en una tabla *hash* la cual relacionará identificadores con actividades.

Enseguida, deberá definirse dicha actividad en Java, en este punto será necesario conocer la descripción de los argumentos que serán enviados, ésto con el fin de saber como será tratada la información intercambiada. Para conocer dicha información el traductor contará con dos tablas *hash* auxiliares para almacenar parámetros y variables locales del método orquestado al que pertenezcan estas actividades. Estas tablas vincularán datos con su tipo. Por último, en el caso de la primitiva *call*, tendrá que verificarse en qué tabla fue almacena la variable de retorno.

Por ejemplo, en la figura (4.8) se muestra un fragmento del esquema de traducción asociado a un par de invocaciones (síncrona y asíncrona respectivamente) mismas que serán lanzadas en orden secuencial. En este ejemplo es posible observar que efectivamente la similitud entre ambas especificaciones BPCL y Java es notoria.

En el archivo de implementación, además deberán ser definidos los participantes involucrados en el proceso. Para realizarlo, el traductor asociará internamente un *Proxy* a cada uno de los participantes. Recordemos que esta entidad provee los mecanismos necesarios para llevar a acabo la orquestación y que utiliza la reflexión para invocar cada una de las peticiones. Una restricción inherente al uso de la reflexión es que el tipo de parámetros de los métodos solicitados deben ser objetos (Float, Inte-

```
<sequential>
  <call partner="pnr1" operation="method1">
    <input>
      <part name="paramn"/>
    </input>
    <output>
      <part name="response"/>
    </output>
  </call>
  <send partner="pnr2" operation="method2">
    <input>
      <part name="paramn"/>
    </input>
  </send>
</sequential>
```

```
Activity c1 = new Activity("call1")
  { public void run(){
    try{
      Object r = pnr1.call(signature_method1,new Object[]{paramn});
      response.set(r);
    } catch(Exception e) { e.printStackTrace(); }
  } };
Activity s2 = new Activity("send2")
  { public void run(){
    try{
      pnr2.send(signature_method2,new Object[]{paramn});
    } catch(Exception e) { e.printStackTrace(); }
  } };
CollaborativeActivity s3 = new Sequential("s3",new Activity[]{c1,s2});
```

Figura 4.8: Fragmento del esquema de traducción BPCL a Java.

ger, String...), sin embargo, los tipos que asociados a cualquier método de un servicio Web son primitivos (float, int...). Por consiguiente, fue necesario implementar un clase *Wrapper* para cada participante mediante la cual se realizará la conversión de tipos de datos y la ejecución del método correspondiente.

Ahora, para definir métodos pertenecientes a un servicio Web orquestado, el traductor verifica las etiquetas **scope** de la especificación para determinarlos. Dichas etiquetas agrupan peticiones de tipo **accept** o **receive** que permitirán conocer el prototipo del método, **variables** para la declaración de variables locales y, por último, el conjunto de actividades que constituyan al método. Cabe señalar que las variables se declararán como **final Variable**. El modificador **final** permitirá que las variables sean reconocidas dentro de cada uno de los hilos que envuelven a las actividades. La clase **Variable** permitirá que sean modificadas, internamente será almacenada una tabla *hash* con los tipos reales de estos objetos consiguiendo así saber como será tratada su información. Por otro lado, para nombrar a las actividades será necesario generar identificadores únicos que las representen.

Hasta este punto se ha descrito como se realizará la orquestación del flujo de trabajo, pero hace falta definir el mecanismo por medio del cual será posible que dicho flujo esté capacitado para recibir peticiones. Para conseguirlo, cada servicio orquestado deberá implementar una función interna de inicialización por cada método orquestado disponible al cliente y una función de arranque donde se especifique que dichos métodos estarán preparados para atender peticiones. El traductor será el encargado de implementar estas funciones de inicialización.

Finalmente, la tarea del traductor termina cuando éste ha generado todos los archivos de implementación y de compilación para cada una de las entidades definidas en el esquema conceptual de diseño de la infraestructura (4.3). A continuación se listan

cada una de las salidas correspondientes:

- Servicio Web (Web Service).
  - BindingImpl. Archivo de implementación donde se redireccionarán cada una las peticiones cliente al servidor RMI.
- Servidor RMI (RMI Server).
  - RMIServer. Archivo mediante el cual el servidor RMI podrá atender y responder a peticiones realizadas por el servidor Web.
  - RMIInterface. Interfaz que describe el conjunto de métodos orquestados disponibles a clientes.
  - RMIImpl. Mediante este archivo el servidor RMI orquestará cada uno de los métodos definidos en la interfaz RMIInterface, en base a la especificación BPCL del proceso de negocios. Además, aquí serán definidas las entidades intermedias llamadas Proxy, las cuales permitirán descubrir e invocar los servicios involucrados a través de sus envolturas (*wrappers*) de datos respectivas.
- Proveedores de Servicios (Suppliers).
  - SupplierWrapper. Envoltura para cada uno de los servicios Web implicados en el proceso. Archivo mediante el cual el desarrollador podrá ejecutar un código local a la aplicación en el momento en que la consulta por el servicio Web involucrado no se encuentre disponible. Además, es importante puntualizar que es aquí y en el *aspecto* AspectSupplierWrapper donde deben realizarse las conversiones a tipos de datos primitivos correspondientes a cada método del servicio, debido a que fue utilizada el API Reflection de Java para realizar las invocaciones.
  - AspectSupplierWrapper. Este archivo representa el *aspecto* de búsqueda e invocación para cada servicio Web elemental, el cual fue diseñado para ser acoplado a la clase SupplierWrapper, de tal manera que pueda trabajarse en forma local o remota.
- Cliente (Client).
  - Init. Archivo mediante el cual será restringido el orden en el que serán aceptadas las peticiones (a métodos orquestados) por parte del cliente.

- Wrapper. Archivo mediante el cual el desarrollador podrá ejecutar un código local a la aplicación en el momento en que la consulta por el servicio Web orquestador no se encuentre disponible. Es aquí y en el *aspecto* AspectWrapper donde deben realizarse las conversiones a tipos de datos primitivos correspondientes a cada método del servicio.
- AspectWrapper. Este archivo representa el *aspecto* de búsqueda e invocación para el servicio Web orquestador, mismo que fue diseñado para ser acoplado a la clase Wrapper, de tal manera que pueda trabajarse en forma local o remota.
- Invoke. Mediante este archivo un cliente podrá insertar el código necesario para invocar métodos orquestados provistos por el servicio Web orquestador, a través de la función *run*. Las líneas de código generadas serán indispensables para poder establecer la comunicación con dicho servicio.

4.3.2. Búsqueda e invocación de un servicio Web

Como lo explicamos en la sección anterior una clase *Wrapper* se crea por cada servicio elemental involucrado en el flujo. Esta clase es la encargada de realizar la búsqueda e invocación de cada uno de los servicios. En la figura (4.9) se muestra el prototipo general que describe a una clase *Wrapper* asociada a un servicio Web.

```
1:  package cinves;
2:  import bpcl.*;
3:
4:  public class ServiceWrapper
5:  {
6:      public ServiceWrapper(String nameService) {
7:          /* Constructor */
8:      }
9:
10:     public typeReturn method_1( parameters ) {
11:         /* Local functionality */
12:     }
13:     ...
14:     public typeReturn method_n( parameters ) {
15:         /* Local functionality */
16:     }
17: }
```

Figura 4.9: Clase Wrapper asociada a un servicio Web.

Sin embargo, para que dichas operaciones (búsqueda e invocación) puedan realizarse dinámicamente éstas fueron encapsuladas en un *aspecto* compuesto por un *advice* de tipo *around*) que se activa en cada momento que deba ser ejecutado el flujo

de trabajo. La búsqueda se realizó al momento de crear una instancia del servicio y la invocación en la llamada a cada uno de sus métodos.

En el *aspecto*, el propósito de la búsqueda es obtener la descripción WSDL de un servicio (elemental u orquestado) a través de la consulta de éste en un repositorio de servicios (la localización de dicho almacén es un dato de configuración de la aplicación). Si la consulta es exitosa entonces se concretiza un objeto de tipo InvokeService con la descripción obtenida y se puede definir un vector con los parámetros necesarios del método remoto a invocar. En caso contrario, puede ejecutarse la definición local que sea especificada por el desarrollador en las clases de envoltura *Wrappers*. Esta particularidad es de gran importancia ya que gracias a ella será posible desarrollar aplicaciones donde no sea esencial contar con una conexión a Internet. El prototipo de un aspecto acoplado a la clase *Wrapper* del servicio en cuestión se muestra en la figura (4.10).

```
1:  public aspect AspectServiceWrapper
2:  {
3:     String wsdl = "";
4:     InvokeService service = null;
5:
6:     /* Discovery of Web service */
7:     void around(String nameService): execution(ServiceWrapper.new(..)) && args(nameService)
8:     {
9:        try {
10:         InvokeService searchSw =
11:             new InvokeService("http://127.0.0.1:6080/Repository/services/RepositoryPort?wsdl");
12:         Vector pars = new Vector(); pars.addElement("nameService");
13:         Vector args = new Vector(); args.addElement(nameService);
14:         wsdl = searchSw.invoke("getUrlWsdlService",pars,args);
15:       }
16:       catch(Exception exp) {
17:          /* Conecction refused, then invocation of local constructor */
18:          proceed(nameService);
19:       }
20:
21:       if(wsdl.compareTo("Service Not Found") != 0)
22:          this.service = new InvokeService(wsdl);
23:       else
24:          proceed(nameService);
25:    }
26:
27:    /* Invocation of Web service */
28:    typeReturn around( parameters ):
29:          execution(* ServiceWrapper.method_1(..)) && args( parameters )
30:    {
31:       if(wsdl.compareTo("Service Not Found") != 0  && service != null) {
32:         try{
33:            /* Remote invocation of Web service */
34:            return service.invoke("method_1",pars,args);
35:         }
```

```
36:         catch(Exception exp) {
37:            /* Conecction refused, then  local invocation  */
38:            return proceed( parameters );
39:         }
40:       }
41:       else
42:          /* Local invocation of service */
43:          return proceed( parameters );
44:    }
45:    ...
46:    typeReturn around( parameters ):
47:           execution(* ServiceWrapper.method_n(..)) && args( parameters )
48:    {
49:       ...
50:    }
51: }
```

Figura 4.10: Aspecto asociado a una clase *Wrapper*.

4.3.3. Publicación de un servicio Web orquestado

Recordemos que el módulo de publicación se divide en dos etapas, la de generación de código y la de registro del servicio. En la figura (4.11) podemos observar a detalle el proceso de publicación. La primera de ellas se lleva a cabo por el traductor (sección 4.3.1), donde se generán todos los archivos de implementación y compilación necesarios. En la segunda etapa el traductor genera un archivo ejecutable donde se especifican cada uno de los comandos necesarios para llevar a cabo el registro del servicio y un archivo ejecutable para que el servidor RMI pueda recibir peticiones. Veamos ahora los pasos necesarios para registrar un servicio.

1. Los archivos creados en la parte de RMI serán compilados utilizando AspectJ como herramienta. Ésto nos permitirá obtener los archivos binarios de la aplicación.

2. Mediante la compilación del archivo RMIImpl y a través del comando *rmic*, obtendremos los archivos binarios Skel y Stub de la aplicación para que el servidor RMI pueda atender peticiones y el servidor Web pueda comunicarse con él.

3. Utilizaremos las herramientas proporcionadas por WSDK (ver sección 2.2.3) para generar el servicio Web con el cual se comunicará el cliente y obtendremos como resultado el contenedor del servicio donde se generarán los archivos propios de una aplicación desarrollada en WebSphere (PortType, Locator, BindingStub, Interface). Además, en dicho contenedor serán depositados los archivos obtenidos en el paso anterior y el respectivo archivo de implementación BindingImpl.

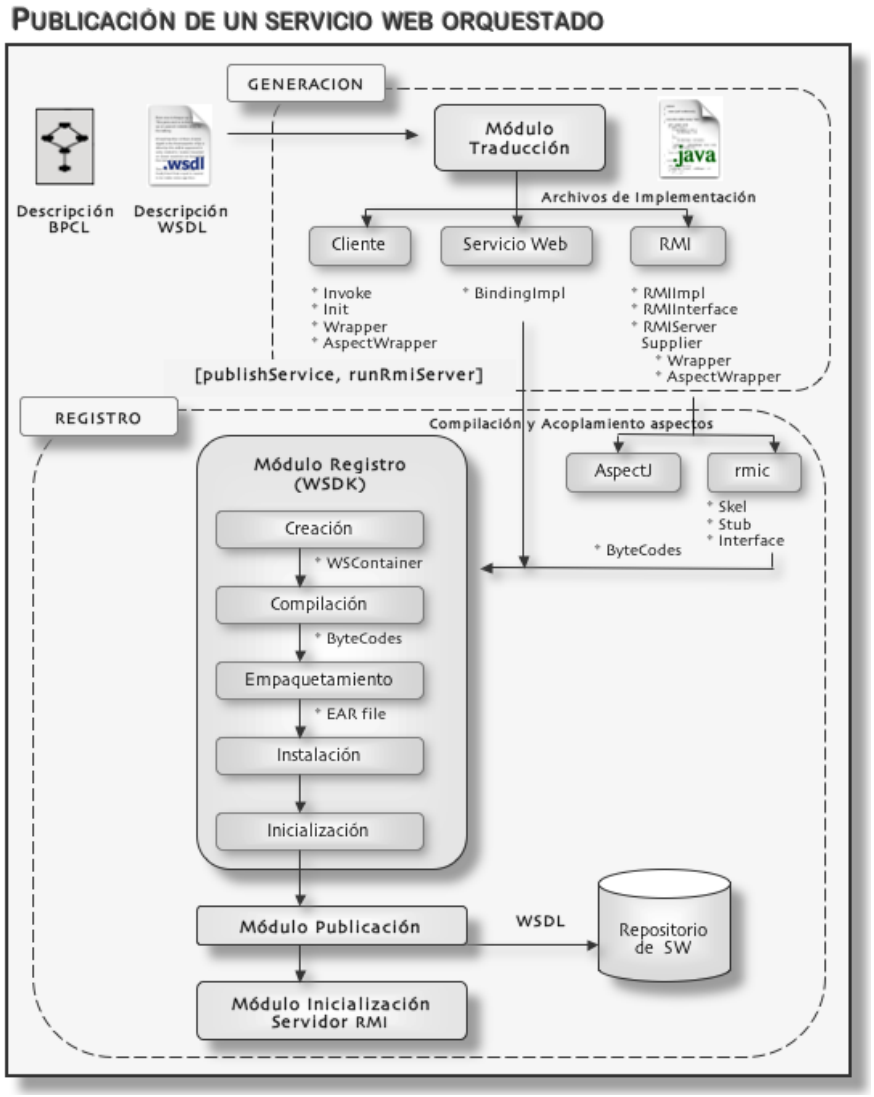

Figura 4.11: Proceso de publicación de un servicio Web orquestado.

4. Todos los archivos anteriores serán compilados utilizando la máquina virtual de Java.

5. Empaquetaremos el servicio Web mediante un archivo .ear.

6. Instalaremos el servicio por medio de su correspondiente archivo de empaquetamiento, con ésto el servicio Web estará registrado debidamente en el contenedor

de servicios de WebSphere.

7. Inicializaremos el servicio Web para recibir peticiones.

8. Publicaremos la descripción WSDL del servicio Web en el repositorio respectivo.

9. Y finalmente, inicializaremos el servidor RMI a través de rmiregistry, el cual permitirá que un servidor RMI reciba peticiones remotas a través de un puerto determinado. Además registrará clases, interfaces y excepciones utilizadas para localizar y nombrar objetos remotos.

Al concluir la publicación sólo restará que sea ejecutado el archivo runRMIServer para que el servicios Web pueda redireccionar sus peticiones al servidor RMI. Una vez hecho lo anterior el cliente podrá escribir su propia implementación para invocar cada uno de los métodos que han sido orquestados.

4.4. Caso de estudio

La compra de artículos electrónicamente representa un escenario típico en aplicaciones de comercio electrónico y B2B, por tal razón fue elegido como nuestro caso de estudio. No obstante, antes de explicar su funcionamiento es necesario describir su entorno de trabajo.

En cualquier escenario aplicado a nuestra infraestructura de coordinación estarán involucrados cinco participantes: un cliente, un servicio Web, un servidor RMI, un conjunto de proveedores de servicios y un conjunto de proxies (uno por cada proveedor). El comportamiento de los participantes es descrito enseguida y sus interacciones se muestran en la figura (4.12).

- Cliente (Client): Es el usuario que realizará peticiones a métodos de un servicio Web orquestado.

- Proveedor (Supplier): Puede existir una o más entidades de este tipo las cuales ofrecerán el conjunto de servicios Web elementales que serán utilizados por el sistema.

- Servicio Web (Web Service): Dicha entidad permitirá recibir las peticiones de clientes y redireccionarlas al servidor RMI.

- Servidor RMI (RMI Server): Será el encargado de llevar a cabo la orquestación de cada uno de los métodos descritos en la especificación BPCL. Redireccionará las respuestas de dichos métodos al servicio Web, de tal manera que este último sea el encargado de hacérselas conocer al cliente. Además está entidad establecerá la comunicación con cada uno de los proveedores de servicios Web.
- Proxy: Permitirá coordinar las peticiones de los participantes involucrados en la invocación de servicios Web y mediante él será posible restringir que las operaciones sean ejecutadas estrictamente en el orden en que fueron descritas en el lenguaje. Además, localizará la ubicación de los servicios necesarios y los invocará, a través de *aspectos*.

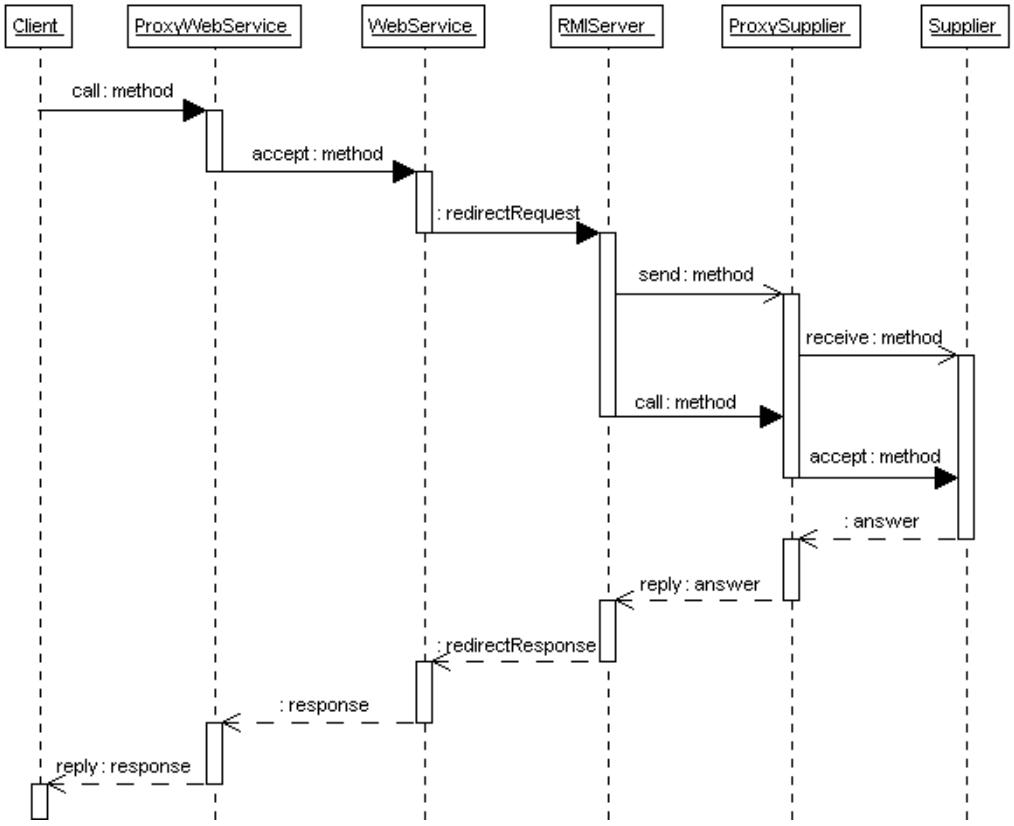

Figura 4.12: Organización general de los participantes.

Ahora bien, el proceso de compra de artículos consiste en conocer la tienda donde el costo de una lista de artículos sea menor, en base a esta operación la decisión de

compra o cancelación puede tomarse. Para obtener la mejor cotización de artículos será necesario coordinar el conjunto de actividades que la constituyen. Primero necesitaremos realizar la cotización de la lista de productos en cada una de las tiendas que estén disponibles y calcular el costo menor entre ellas. Una vez hecho ésto, podremos conocer cuál será la tienda donde el costo de la compra sea menor.

El diagrama UML de secuencia para el caso de estudio se muestra en la figura (4.13). Las interacciones involucradas se describen a continuación:

- El proceso comienza cuando un cliente desea conocer la tienda donde el costo de su lista de artículos sea menor. Para logralo éste solicitará el método síncrono *getBestQuotation* del servicio Web *Broker*. Recordemos que la comunicación entre participantes se realizará a través de una entidad *Proxy*, por lo que el cliente se comunicará con el *Broker* a través de *ProxyBroker*. Dicha entidad permitirá restringrir que la primera petición realizada sea única y exclusivamente la mejor cotización. Luego éste realizará la invocación respectiva al *Broker*. Pasos 1 y 2.

- Una vez que el *Broker* recibe la petición por parte del *Proxy*, éste la redireccionará al servidor RMI denominado *BrokerRMI*. Paso 3.

- La comparación de las cotizaciones de artículos en cada una de las tiendas disponibles permitirá determinar la mejor cotización. Por tal razón, el *BrokerRMI* invocará como primera actividad el método síncrono *getQuotation* del servicio Web provisto por la entidad *Store*. Dicho método retornará el costo total de la compra en una tienda en particular. Una vez que sea obtenido el total, la siguiente actividad invocada será la petición del método asíncrono *setMinCost* de la entidad *Blackboard*. Este método irá calculando el mínimo costo de cada una de las cotizaciones realizadas asociando su tienda respectiva. Pasos 4 al 9.

- El *BrokerRMI* realizará las dos actividades anteriores de manera secuencial y las iterará tantas veces como tiendas estén disponibles. A su vez, ejecutará el conjunto de iteraciones de forma concurrente (en paralelo), de tal manera que el proceso sea agilizado. Paso 10.

- Una vez que el conjunto de cotizaciones sean procesadas, el *BrokerRMI* podrá obtener el id de la tienda donde el mínimo costo haya sido calculado, a través de la operación *getBest* provista por el *Blackboard*. Entonces, dicho id será retornado al servidor *Broker*. Pasos 11 al 15.

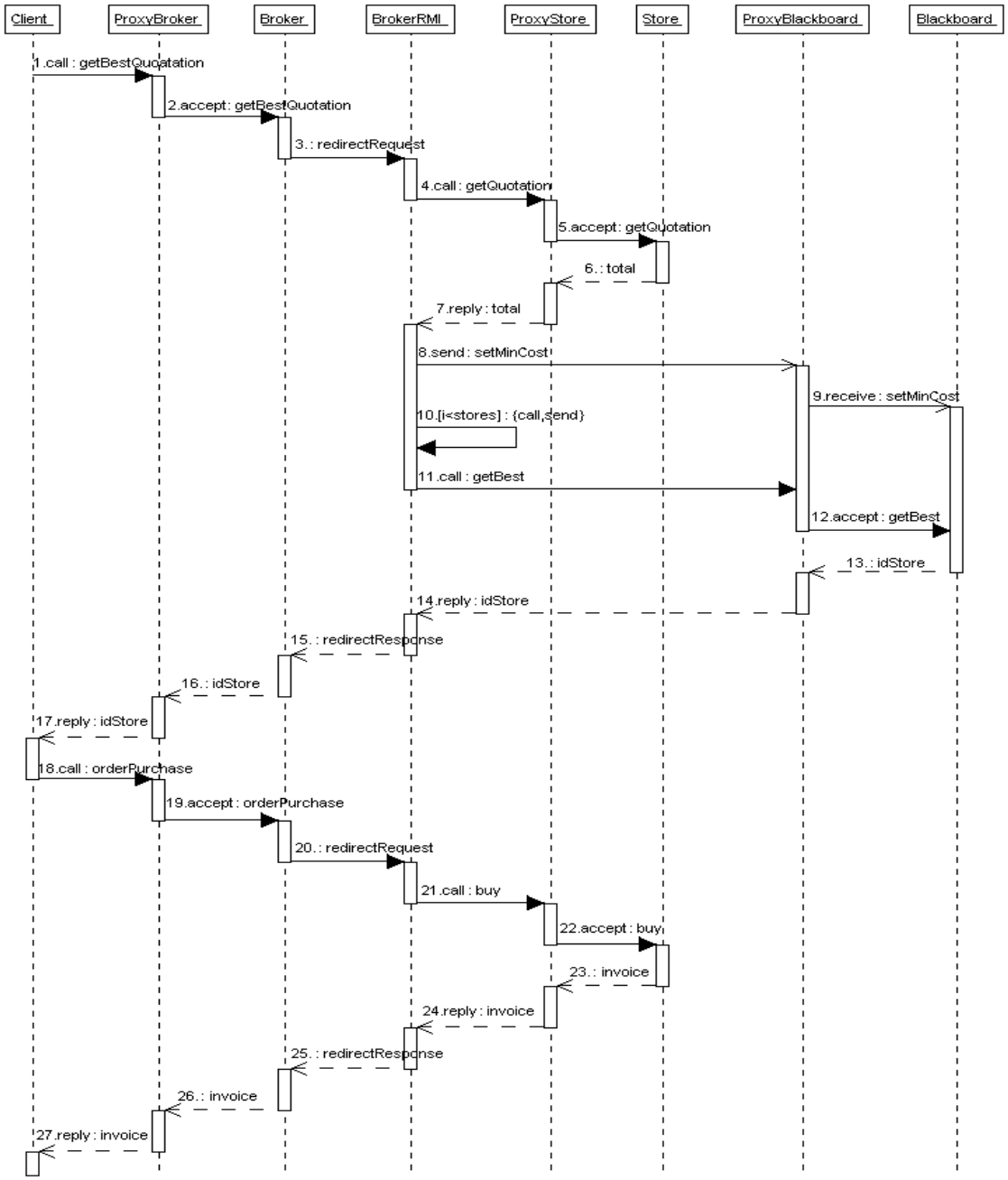

Figura 4.13: Diagrama UML de secuencia para el caso de estudio.

- El *Broker* redireccionará la respuesta del servidor RMI al *ProxyBroker*. Mismo que hará conocer al cliente la respuesta del método orquestado *getBestQuotation*. Pasos 16 y 17.

- Ahora el cliente podrá decidir si continúa con el proceso de compra o cancela su orden. En este caso ilustramos la primera opción, cuando el cliente realiza la compra mediante la operación síncrona *orderPurchase* ofrecida por el *Broker*. Paso 18.

- Al igual que sucedió con el método anterior, *ProxyBroker* atenderá la petición del cliente y la enviará al *Broker*. Dicho servicio redireccionará su petición al servidor RMI y ahora este invocará el método síncrono *buy* provisto por la entidad *Store*. Una vez que se procesa el método, será enviada la factura correspondiente (respuesta) al *Broker*. Pasos 19 al 25.

- El *Broker* redireccionará la respuesta del servidor RMI a la entidad *ProxyBroker*, para que finalmente, dicha entidad dé a conocer al cliente su factura por la compra de artículos. Pasos 26 y 27.

Recordemos que el diagrama de secuencia del caso de estudio (4.13) fue diseñado de acuerdo al conjunto de interacciones realizadas al aplicarlo a la infraestructura de coordinación propuesta. Es por éso que fue necesario especificar explícitamente las entidades *Proxy: Broker, Store, Blackboard* involucradas en el proceso. Sin embargo, cuando un usuario escriba la especificación BPCL del caso de estudio, no tendrá que preocuparse por incluir este tipo de entidades, ya que la infraestructura de coordinación las implementará para coordinar internamente el proceso y localizar la ubicación de los servicios necesarios.

Veamos entonces la especificación para la compra de artículos escrita en BPCL. Dicha especifición se muestra en la figura (4.14).

```
1:  <?xml version="1.0"?>
2:  <process name="Broker" javalib=""
3:      xmlns:xs="http://www.w3.org/2001/XMLSchema"
4:      xsi:noNamespaceSchemaLocation="bpcllanguage.xsd">
5:
6:    <partners>
7:      <partner name="broker" type="Broker"/>
8:      <partner name="stores" type="Store[3]"/>
9:      <partner name="store" type="Store"/>
10:     <partner name="blackboard" type="Blackboard"/>
11:   </partners>
```

```
12:
13:   <sequential>
14:     <scope>
15:       <variables>
16:         <variable name="urlStore" type="String[]"/>
17:         <variable name="total" type="float"/>
18:         <variable name="best" type="String"/>
19:       </variables>
20:
21:       <accept partner="broker" operation="getBestQuotation">
22:         <input>
23:           <part name="nameClient" type="String"/>
24:           <part name="articles" type="String"/>
25:         </input>
26:       </accept>
27:       <sequential>
28:         <execute operation="getUrlStore">
29:           <input> </input>
30:           <output>
31:             <part name="urlStore"/>
32:           </output>
33:         </execute>
34:         <parallel>
35:           <iterator variable="j" initial="0" final="stores.length" cond="lt" next="+">
36:             <sequential>
37:               <call partner="stores[j]" operation="getQuotation">
38:                 <input>
39:                   <part name="urlStore[j]"/>
40:                   <part name="nameClient"/>
41:                   <part name="articles"/>
42:                 </input>
43:                 <output>
44:                   <part name="total"/>
45:                 </output>
46:               </call>
47:               <send partner="blackboard" operation="setMinCost">
48:                 <input>
49:                   <part name="urlStore[j]"/>
50:                   <part name="total"/>
51:                 </input>
52:               </send>
53:             </sequential>
54:           </iterator>
55:         </parallel>
56:         <call partner="blackboard" operation="getBest">
57:           <input> </input>
58:           <output>
59:             <part name="best"/>
60:           </output>
61:         </call>
62:       </sequential>
63:       <reply partner="broker" operation="getBestQuotation" variable="best"/>
64:     </scope>
65:     <alternative>
66:       <scope>
```

```
67:         <variables>
68:           <variable name="invoice" type="String"/>
69:         </variables>
70:
71:         <accept partner="broker" operation="orderPurchase">
72:           <input>
73:             <part name="url" type="String"/>
74:             <part name="nameClient" type="String"/>
75:             <part name="articles" type="String"/>
76:           </input>
77:         </accept>
78:         <call partner="store" operation="buy">
79:           <input>
80:             <part name="url"/>
81:             <part name="nameClient"/>
82:             <part name="articles"/>
83:           </input>
84:           <output>
85:             <part name="invoice"/>
86:           </output>
87:         </call>
88:         <reply partner="broker" operation="orderPurchase" variable="invoice"/>
89:       </scope>
90:
91:       <scope>
92:         <variables>
93:           <variable name="confirmation" type="String"/>
94:         </variables>
95:
96:         <accept partner="broker" operation="orderCancel">
97:           <input>
98:             <part name="url" type="String"/>
99:             <part name="nameClient" type="String"/>
100:          </input>
101:        </accept>
102:        <call partner="store" operation="cancelBuy">
103:          <input>
104:            <part name="url"/>
105:            <part name="nameClient"/>
106:          </input>
107:          <output>
108:            <part name="confirmation"/>
109:          </output>
110:        </call>
111:        <reply partner="broker" operation="orderCancel" variable="confirmation"/>
112:      </scope>
113:    </alternative>
114:  </sequential>
115:
116:  <localOperation>
117:    <operation>
118:      public String[] getUrlStore() {
119:        String[] urlStore = {"comprarDB1","comprarDB2","comprarDB3"};
120:        return urlStore;
121:      }
```

```
122:    </operation>
123:  </localOperation>
124:
125:</process>
```

Figura 4.14: Especificación para la compra de artículos escrita en BPCL.

En la línea (2), el inicio del proceso a ser orquestado se define por la etiqueta de apertura *process*. En las líneas (6-11) podemos ver como se realiza la declaración de los participantes involucrados en el proceso, donde se consigue omitir los detalles de la ubicación de los recursos. De la línea (14-64) se encuentra envuelto por una etiqueta *scope* el flujo de actividades que componen al método orquestado *getBestQuotation*, la definición de sus parámetros, variables locales y valor de retorno. De manera similar sucede con los métodos *orderPuchase* (66-89) y *orderCancel* (91-112).

El orden en que el usuario debe invocar dichas operaciones es restringido por la etiqueta *sequential* (13) y *alternative* (65), donde se especifica que la primera actividad a ser invocada por un cliente será *getBestQuotation* y que alternativamente puede esperarse una petición compra o cancelación (*orderPuchase* y *orderCancel* respectivamente).

En la sección de las líneas (116-123) se declara la operación *getUrlStore* que será local al *Broker*. Su funcionalidad fue escrita directamente en Java debido a que esta operación no será provista por un servicio Web externo.

Finalmente, el término del proceso a ser orquestado se define por la etiqueta de cierre *process* (125).

En las figuras (4.13) y (4.14) podemos observar que existe una correspondencia muy cercana entre el diagrama de secuencia del caso de estudio y su correspondiente especificación en BPCL, debido a que para escribir dicha especificación no fue necesario tomar en cuenta los detalles de distribución de los recursos. Es importante recordar que esta tarea quedará a cargo de la infraestructura de coordinación, una vez que el proceso de traduccción sea puesto en marcha.

Debido a que los archivos de implementación en Java generados por la infraestructura de coordinación para cada una de las entidades involucradas en la aplicación son numerosos, éstos se muestran en el apéndice C, donde pueden analizarse con mayor detenimiento.

4.5. Sumario

Mediante la infraestructura de coordinación propuesta se consigue que el desarrollador de una aplicación orquestada pueda concentrarse en el diseño conceptual de la aplicación más que en los detalles de localización de los participantes y que sea más clara y simple la especificación del proceso de negocios. Además, ésta permitirá el desarrollo, depuración y ejecución de aplicaciones orquestadas cuando no exista una disponibilidad de recursos remotos.

Además, es importante señalar que en dicha infraestructura pueden ser desarrolladas aplicaciones orquestadas en Java sin necesidad de contar forzosamente con una descripción BPCL, ya que el lenguaje fue creado solamente con el fin de facilitar esta tarea a los usuarios. No obstante, si éstos conocen los elementos necesarios para crear una aplicación directamente en la infraestructura pueden realizarlo sin ningún problema.

Capítulo 5

Conclusiones

El trabajo presentado en esta tesis representa una alternativa novedosa para llevar a cabo la orquestación de servicios Web, brindando grandes beneficios en cuanto a costo y funcionalidad, en comparación con otros enfoques desarrollados, haciendo uso de las bondades que actualmente ofrece la tecnología y permitiendo resolver un buen número de situaciones reales, recurrentes y relevantes en el área de comercio electrónico.

Al reducir la complejidad del proceso de orquestación, un sistema de negocios será más flexible, modular, adaptable y dinámico, por tal razón, la facilidad para ejecutar éste proceso, fue sin duda nuestra principal aportación. No obstante, además de lograr cada uno de los objetivos planteados en nuestra propuesta, al introducir a BPCL como un lenguaje composicional, crear una infraestructura de coordinación y crear una biblioteca de clases (bpcl.jar) que da soporte a dicho proceso, realizamos la redacción de un artículo de publicación [33] acerca del trabajo expuesto, el cual fue presentado en un congreso internacional del área de cómputo. Veamos ahora a detalle las contribuciones tecnológicas ofrecidas por este trabajo.

5.1. Contribuciones del trabajo

1. Lenguaje de coordinación

   La especificación de un nuevo lenguaje composicional donde la descripción de un proceso de negocios orquestado pudiera realizarse sin necesidad de especificar detalles de distribución de recursos, es decir, que un usuario tuviera la

capacidad de describir dicho proceso como si éste fuera local, fue la principal aportación de *Business Process Coordination Language (BPCL)*. Además, fue realizada una especificación adicional (esquema del lenguaje), la cual permitiera verificar sintácticamente la validez o correctitud de cualquier descripción escrita en BPCL.

2. Infraestructura de coordinación

   Se realizó el diseño e implementación de una infraestructura de coordinación encargada de crear, publicar y ejecutar procesos orquestados, opcionalmente a partir de su respectiva especificación BPCL y su descripción WSDL. Además, fue creada una librería de datos (clases) que permitiera dar soporte a todas las primitivas de coordinación BPCL en su respectiva representación Java. Es importante señalar que en dicha infraestructura pueden ser desarrolladas aplicaciones orquestadas sin necesidad de contar con una descripción BPCL, ya que el lenguaje fue creado solamente con el fin de facilitar esta tarea a los usuarios. No obstante, si éstos conocen los elementos necesarios para crear una aplicación directamente en la infraestructura pueden realizarlo sin mayor problema.

3. Enfoque orientado a aspectos

   Fueron empleados mecanismos de orientación a aspectos para integrar el acoplamiento, comunicación y coordinación de los participantes dentro de la infraestructura. Para realizar la búsqueda e invocación dinámica de cada uno de los servicios involucrados en el proceso, estas operaciones fueron encapsuladas en un *aspecto*, el cual se activa en cada momento que deba ser ejecutado el flujo de trabajo. En el *aspecto*, el propósito de la búsqueda es obtener la descripción WSDL de un servicio (elemental u orquestado) a través de la consulta de éste en un repositorio de servicios. Si la consulta es exitosa entonces se creará un objeto que invoque remotamente al servicio, en caso contrario, podrá ejecutarse una funcionalidad local especificada por el desarrollador. Esta particularidad es de gran importancia ya que gracias a ella será posible desarrollar aplicaciones donde no sea esencial contar con una conexión a Internet.

4. Proceso de ingeniería

   El acoplamiento de las contribuciones anteriores en un sistema de coordinación, definen el proceso de ingeniería utilizado para el desarrollo de aplicaciones distribuidas. Dicho proceso tiene grandes ventajas, ya que permite que una

aplicación inherementemente distribuida pueda ser probada en cada una de sus etapas de desarrollo sin necesidad de que esta esté conectada a Internet. Lo anterior es posible gracias a que los mecanismos de acceso remoto a los recursos fueron implementados en *aspectos*, mismos que permiten ejecutar un recurso localmente, en el momento en que se detecte que un recurso remoto no se encuentre disponible. Así por ejemplo, si un servidor o servicio Web remoto al que se deseará invocar no estuviera en funcionamiento, esto no afectaría el flujo orquestado. De esta manera, una aplicación podrá ser depurada más fácilmente al no tener una dependencia forzosa de una conexión en red. No obstante, al momento de conectar los componentes en red, dicha operación será transparente al usuario. Por tal razón, el uso de este proceso de ingeniería reduce la complejidad del desarrollo de aplicaciones distribuidas.

5.2. Trabajo futuro

Aunque nuestro trabajo de tesis provee una base sólida para el desarrollo de aplicaciones de comercio electrónico y B2B, aún queda bastante trabajo por realizar, de manera tal que su robustez y potencialidad sea incrementada. Así, en la siguiente lista enumeramos las mejoras más significativas que pueden realizarse.

1. En el proceso de traducción, además de utilizar el esquema XML del lenguaje para verificar la correctitud de una descripción BPCL, es necesario implementar un analizador sintáctico que se encargue de validar que en dicha descripción, cada uno de los participantes especificados solicite sólo operaciones que estén definidas en su descripción WSDL y que éstas cumplan con su respectiva definición (nombre, tipos de parámetros y tipo de retorno).

2. Incorporar las especificaciones WS-Coordination (WS-C) [22] y WS-Transaction (WS-T) [23] a nuestra propuesta. Debido a que la primera de ellas soporta, integra y unifica diferentes modelos de coordinación permitiendo a diferentes sistemas interoperar. Provee además mecanismos estándares para crear y registrar servicios usando los protocolos definidos en WS-T. WS-T coordina la ejecución de operaciones distribuidas en el ambiente de servicios Web a través de la definición de protocolos para permitir realizar transacciones atómicas, compensación de transacciones y control de excepciones.

3. La introducción de *aspectos* directamente en una especificación BPCL, permitiría a un usuario definir cualquier funcionalidad adicional al proceso y no sólo

aquellas que pemiten realizar la búsqueda e invocación dinámica de servicios. No obstante, a pesar de sus ventajas una de las implicaciones que se generarían es que el usuario tendría que tener cierto conocimiento en el uso de los conceptos de la orientación a aspectos.

4. En nuestra propuesta fue utilizado AspectJ [20] como lenguaje para escribir *aspectos*. Sin embargo, AspectJ sólo permite el acomplamiento de dichos *aspectos* en tiempo de carga y de compilación, por tal razón, sería interesante utilizar algún otro lenguaje orientado a aspectos que permita realizar el acoplamiento de éstos en tiempo de ejecución, por ejemplo la *Java Tool Interface* [34]. Así el dinamismo de las aplicaciones desarrolladas se incrementaría ya que los *aspectos* realmente serían módulos independientes de la aplicación, compilándose de forma aislada y acoplándose hasta su ejecución.

5. El acoplamiento de nuestra propuesta con el trabajo presentado en [35] [36], permitiría que la infraestructura de coodinación soportara el manejo de transacciones largas. Ésto quiere decir, que sería posible establecer y atender transacciones encoladas o en estado de espera, cuando los recursos a los cuales se desee acceder no estén disponibles en el momento que sea realizada una petición.

6. La implementación de un repositorio estándar de servicios Web como lo es un nodo UDDI, para permitir el descubrimiento y la publicación de servicios, el cual además permitiera trabajar de forma local y remota. Local cuando no estuviera disponible una conexión a Internet y remota para actualizar la información utilizada en el repositorio cuando se trabaja de forma local.

7. Por último, la implementación de un módulo auxiliar de mantenimiento de información que pueda ser acoplado al servicio Web orquestador, con el fin de que éste sea el encargado de mantener la coherencia y consistencia de una aplicación orquestada. Por ejemplo, si el servicio que orquesta hace uso de ciertos datos necesarios para coordinar el flujo de trabajo y éstos son suceptibles a ser modificados, el módulo auxiliar permitiría mantenerlos actualizados. Otro beneficio de este módulo podría obtenerse cuando fuera realizada la implementación de un nodo UDDI en la infraestructura de coordinación, ya que éste sería el encargado de actualizar la información del repositorio local a través de un repositorio remoto cada vez que éste contara con una conexión en Internet.

Apéndice A

XML Scheme para procesos BPCL

Un esquema XML permite verificar sintácticamente la validez o correctitud de cualquier descripción que ha sido escrita en base a un conjunto de etiquetas en documentos XML.

En un esquema XML es posible definir: (a) los elementos que pueden aparecer en un documento y sus atributos, los tipos de datos correspondientes y los valores que pueden tomar por defecto, (b) cómo pueden anidarse, es decir, si existe un orden de aparición restringido, (c) el número permitido de elementos hijos y por último (d) si un elemento puede ser vacío o no.

El esquema XML asociado a cualquier especificación escrita en BPCL es mostrado en el siguiente listado A.1.

```
1:  <?xml version="1.0"?>
2:  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
3:
4:  <xs:element name="process" type="processType"/>
5:
6:  <xs:complexType name="processType">
7:    <xs:sequence>
8:
9:      <xs:element name="partners" minOccurs="1" maxOccurs="1">
10:       <xs:complexType>
11:         <xs:sequence>
12:           <xs:element name="partner">
13:             <xs:complexType>
14:               <xs:attribute name="name" type="xs:string" use="required"/>
15:               <xs:attribute name="type" type="xs:string" use="required"/>
16:             </xs:complexType>
17:           </xs:element>
```

```
18:         </xs:sequence>
19:       </xs:complexType>
20:     </xs:element>
21:
22:     <xsd:choice>
23:       <xs:element name="sequential"  type="seqControlFlowType"/>
24:       <xs:element name="alternative" type="altControlFlowType"/>
25:     </xsd:choice>
26:
27:     <xs:element name="localOperation"  maxOccurs="1">
28:       <xs:complexType>
29:         <xs:sequence>
30:           <xs:element name="operation"/>
31:         </xs:sequence>
32:       </xs:complexType>
33:     </xs:element>
34:
35:   </xs:sequence>
36:   <xs:attribute name="name"    type="xs:string" use="required"/>
37:   <xs:attribute name="javalib" type="xs:string"/>
38: </xs:complexType>
39:
40: <xs:complexType name="seqControlFlowType">
41:   <xs:sequence>
42:     <xs:element name="sequential"   type="seqControlFlowType"/>
43:     <xs:element name="alternative"  type="altControlFlowType"/>
44:     <xs:element name="scope"        type="scopeType" maxOccurs="1"/>
45:   </xs:sequence>
46:   <xs:attribute name="name" type="xs:string" use="required"/>
47: </xs:complexType>
48:
49: <xs:complexType name="altControlFlowType">
50:   <xs:sequence>
51:     <xs:element name="sequential"   type="seqControlFlowType"/>
52:     <xs:element name="alternative"  type="altControlFlowType"/>
53:     <xs:element name="scope"        type="scopeType" maxOccurs="1"/>
54:   </xs:sequence>
55:   <xs:attribute name="name" type="xs:string" use="required"/>
56: </xs:complexType>
57:
58: <xs:complexType name="scopeType">
59:   <xsd:complexContent>
60:     <xsd:extension base="instructionFlowType">
61:       <xs:sequence>
62:         <xsd:all>
63:           <xs:element name="variable"     type="variableType"/>
64:           <xsd:choice>
65:             <xs:element name="receive"    type="inRequestType"/>
66:             <xsd:all>
67:               <xs:element name="accept"   type="inRequestType"/>
68:               <xs:element name="reply"    type="replyType"/>
69:             </xsd:all>
70:           </xsd:choice>
71:         </xsd:all>
72:       </xs:sequence>
```

```
73:      </xsd:extension>
74:    </xsd:complexContent>
75:    <xs:attribute name="name" type="xs:string" use="required"/>
76:  </xs:complexType>
77:
78:  <xs:complexType name="variableType">
79:    <xs:sequence>
80:      <xs:element name="variables">
81:        <xs:complexType>
82:          <xs:sequence>
83:            <xs:element name="variable">
84:              <xs:complexType>
85:                <xs:attribute name="name" type="xs:string" use="required"/>
86:                <xs:attribute name="type" type="xs:string" use="required"/>
87:              </xs:complexType>
88:            </xs:element>
89:          </xs:sequence>
90:        </xs:complexType>
91:      </xs:element>
92:    <xs:sequence>
93:  </xs:complexType>
94:
95:  <xs:complexType name="inRequestType">
96:    <xs:sequence>
97:      <xs:element name="input">
98:        <xs:complexType>
99:          <xs:sequence>
100:           <xs:element name="part">
101:             <xs:complexType>
102:               <xs:attribute name="name" type="xs:string" use="required"/>
103:               <xs:attribute name="type" type="xs:string" use="required"/>
104:             </xs:complexType>
105:           </xs:element>
106:         </xs:sequence>
107:       </xs:complexType>
108:     </xs:element>
109:   </xs:sequence>
110:   <xs:attribute name="partner"    type="xs:string" use="required"/>
111:   <xs:attribute name="operation"  type="xs:string" use="required"/>
112: </xs:complexType>
113:
114: <xs:complexType name="replyType">
115:   <xs:attribute name="partner"    type="xs:string" use="required"/>
116:   <xs:attribute name="operation"  type="xs:string" use="required"/>
117:   <xs:attribute name="variable"   type="xs:string" use="required"/>
118: </xs:complexType>
119:
120: <xs:complexType name="instructionFlowType">
121:   <xs:sequence>
122:     <xs:element name="sequential"  type="instructionFlowType"/>
123:     <xs:element name="parallel"    type="instructionFlowType"/>
124:     <xs:element name="iterator"    type="iteratorType"/>
125:     <xs:element name="call"        type="outRequestType"/>
126:     <xs:element name="execute"     type="outRequestType"/>
127:     <xs:element name="send"        type="sendType"/>
```

```
128:   </xs:sequence>
129: </xs:complexType>
130:
131: <xs:complexType name="iteratorType">
132:   <xs:sequence>
133:     <xs:element name="sequential" type="instructionFlowType"/>
134:     <xs:element name="parallel"   type="instructionFlowType"/>
135:   </xs:sequence>
136:   <xs:attribute name="variable"  type="xs:string" use="required"/>
137:   <xs:attribute name="initial"   type="xs:string" use="required"/>
138:   <xs:attribute name="final"     type="xs:string" use="required"/>
139:   <xs:attribute name="cond"      type="xs:string" use="required"/>
140:   <xs:attribute name="next"      type="xs:string" use="required"/>
141: </xs:complexType>
142:
143: <xs:complexType name="outRequestType">
144:   <xs:sequence>
145:     <xs:element name="input"    type="inoutType">
146:     <xs:element name="output"   type="inoutType">
147:   </xs:sequence>
148:   <xs:attribute name="partner"    type="xs:string" use="required"/>
149:   <xs:attribute name="operation"  type="xs:string" use="required"/>
150: </xs:complexType>
151:
152: <xs:complexType name="sendType">
153:   <xs:sequence>
154:     <xs:element name="input"  type="inoutType">
155:   </xs:sequence>
156:   <xs:attribute name="partner"    type="xs:string" use="required"/>
157:   <xs:attribute name="operation"  type="xs:string" use="required"/>
158: </xs:complexType>
159:
160: <xs:complexType name="inoutType">
161:   <xs:sequence>
162:     <xs:element name="part">
163:       <xs:complexType>
164:         <xs:attribute name="name" type="xs:string" use="required"/>
165:       </xs:complexType>
166:     </xs:element>
167:   </xs:sequence>
168: </xs:complexType>
169:
170: </xs:schema>
```

Figura A.1: Esquema XML para procesos BPCL.

Apéndice B

Documentación de la biblioteca de coordinación BPCL.jar

Este apéndice tiene como objetivo presentar el conjunto de atributos, constructores y métodos que constituyen cada una de las clases que conforman la biblioteca de coordinación denominada **bpcl.jar**.

B.1. Etapa de Coordinación

Class Activity

```
java.lang.Object
  java.lang.Thread
    bpcl.Activity
```

public class **Activity**

Representa una actividad descrita en BPCL, misma que puede ser iniciada o suspendida.

Autor: Nancy N. Cova Suazo, BPCL version 1.0, CINVESTAV-IPN, 2005.

Attribute Summary	
CollaborativeActivity **group**	Grupo al cual pertenece una actividad.

Constructor Summary
Activity(String name) Constructor que inicializa una actividad con su nombre.
Activity(String name, CollaborativeActivity group) Constructor que inicializa una actividad con su nombre y grupo padre.

Method Summary	
CollaborativeActivity **getGroup**()	Obtiene el grupo al que pertenece una actividad.
void **setGroup**(CollaborativeActivity group)	Incorpora una actividad al grupo.
void **started**()	Comienza la ejecución de una actividad.
void **stopped**()	Suspende la ejecución de una actividad.

Class CollaborativeActivity

```
java.lang.Object
  java.lang.Thread
      bpcl.Activity
          bpcl.CollaborativeActivity
```

public class **CollaborativeActivity**

Define conjuntos de actividades por medio de un grupo de *threads*. Además, permite suspender la ejecución cada uno de sus elementos con excepción de una actividad que sea solicitada.

Autor: Nancy N. Cova Suazo, BPCL version 1.0, CINVESTAV-IPN, 2005.

Attribute Summary	
Activity[ ] **activity**	Lista de actividades que colaborarán.

Constructor Summary
CollaborativeActivity(String name, Activity[ ] act) Constructor que inicializa un grupo de actividades con su nombre.
CollaborativeActivity(String name, Activity[ ] act, CollaborativeActivity group) Constructor que inicializa un grupo de actividades con su nombre y grupo padre.

Method Summary	
void **stopGroupButNotMe**()	Suspende la ejecución de un grupo de actividades, excepto por la primer actividad del conjunto que término su ejecución.

Class Sequential

```
java.lang.Object
  java.lang.Thread
      bpcl.Activity
          bpcl.CollaborativeActivity
            bpcl.Sequential
```

public class **Sequential**

Ejecuta un grupo de actividades secuencialmente.

Autor: Nancy N. Cova Suazo, BPCL version 1.0, CINVESTAV-IPN, 2005.

Constructor Summary
Sequential(String name, Activity[ ] activity) Constructor que inicializa un grupo de actividades con su nombre. Éste será ejecutado secuencialmente.
Sequential(String name, Activity[ ] activity, CollaborativeActivity group) Constructor que inicializa un grupo de actividades con su nombre y grupo padre. Éste será ejecutado secuencialmente.

Method Summary	
void **run**()	Ejecuta un grupo de actividades de manera secuencial.

Class Parallel

```
java.lang.Object
  java.lang.Thread
      bpcl.Activity
          bpcl.CollaborativeActivity
            bpcl.Parallel
```

public class **Parallel**

Ejecuta un grupo de actividades de manera concurrente (en paralelo).

Autor: Nancy N. Cova Suazo, BPCL version 1.0, CINVESTAV-IPN, 2005.

Constructor Summary
Parallel(String name, Activity[ ] activity) Constructor que inicializa un grupo de actividades con su nombre. Éste será ejecutado concurrentemente.
Parallel(String name, Activity[ ] activity, CollaborativeActivity group) Constructor que inicializa un grupo de actividades con su nombre y grupo padre. Éste será ejecutado concurrentemente.

Method Summary	
void **run**()	Ejecuta un grupo de actividades en paralelo.

Class Synchronizer

```
java.lang.Object
  bpcl.Synchronizer
```

public class **Synchronizer**

Funciona al igual que un semáforo tradicional con sus respectivas primitivas de sincronización *down* y *up*. Esta clase sincroniza las actividades del lenguaje: call, accept/reply, send, receive.

Autor: Nancy N. Cova Suazo, BPCL version 1.0, CINVESTAV-IPN, 2005.

Attribute Summary	
int **counter**	Contador asociado al semáforo.

Constructor Summary
Synchronizer(int counter) Constructor que inicializa el contador del semáforo.

Method Summary	
int **cn**()	Regresa el valor asociado al contador del semáforo.
void **dn**()	Decrementa el valor del contador del semáforo en una unidad, si éste es menor que cero entonces la actividad actual se bloqueará.
void **up**()	Incrementa el valor del contador del semáforo en una unidad, pero antes se verifica si el valor de éste es menor que cero, en tal caso una actividad será desbloqueada.

Class Transaction

```
java.lang.Object
  bpcl. Transaction
```

public class **Transaction**

Encapsula la información de una invocación a un servicio Web como una transacción. De ésta forma es posible crear una instancia de su tipo con la cual se conozca el objeto que realizó la invocación, el nombre del método solicitado, sus parámetros y su valor de retorno.

Autor: Nancy N. Cova Suazo, BPCL version 1.0, CINVESTAV-IPN, 2005.

Attribute Summary	
Object **caller**	Objeto asociado a la entidad solicitante.
ProxyMethod **method**	Nombre del método solicitado.
Object[ ] **parameters**	Parámetros del método solicitado.
Object **result**	Resultado asociado al método solicitado.

Class ProxyMethod

```
java.lang.Object
  bpcl. ProxyMethod
```

public class **ProxyMethod**

Almacena un objeto de tipo Transaction al cual se asocian los semáforos necesarios para sincronizar las invocaciones de servicios.

Autor: Nancy N. Cova Suazo, BPCL version 1.0, CINVESTAV-IPN, 2005.

Constructor Summary
ProxyMethod(Method method) Constructor que inicializa un método como un ProxyMethod, de tal manera que la transacción solicitada pueda sincronizarse.

Attribute Summary	
Method **delegate**	Objeto del método solicitado.
Transaction **transaction**	Transacción asociada a la entidad solicitante.
Synchronizer **prologue**	Semáforo asociado al prólogo de la transacción.
Synchronizer **epilogue**	Semáforo asociado al epílogo de la transacción.
Synchronizer **proceed**	Semáforo asociado al cuerpo principal de la transacción.
Synchronizer **lock**	Semáforo utilizado para bloquear el acceso a la región crítica.

Class Proxy

```
java.lang.Object
  bpcl. Proxy
```

public class **Proxy**

Hace posible que las peticiones de los participantes involucrados en la invocación de servicios Web sean coordinadas estrictamente en el orden en que fueron descritas en el lenguaje. Es importante señalar que cada método de un servicio se ejecuta a través del Reflection API de Java. Con ésto se consigue realizar las invocaciones a través de un mecanismo uniforme, ya que es posible invocar cualquier método de un objeto que sea conocido hasta tiempo de ejecución.

Autor: Nancy N. Cova Suazo, BPCL version 1.0, CINVESTAV-IPN, 2005.

Attribute Summary	
Object **callee**	Objeto de la entidad solicitante.
Hashtable **methodTable**	Tabla hash asociada a los métodos solicitados.

Constructor Summary
Proxy(Object object) Constructor que inicializa el objeto que invocará alguna primitiva del lenguaje.

Method Summary
Object[ ] **accept**(String signature, Transaction transaction)
Ejecuta la invocación síncrona de servicios Web.
Object **call**(String signature, Object[] arguments)
Solicita la invocación síncrona de servicios Web.
ProxyMethod **getMethod**(String signature)
Transforma la descripción de un método en un tipo ProxyMethod.
Object **getRepresented**()
Regresa el objeto al cual pertenecen los métodos que han sido solicitados.
Object[ ] **reply**(String signature, Transaction transaction)
Retorna la respuesta de una invocación síncrona de servicios Web.
Object **receive**(Transaction transaction)
Ejecuta la invocación asíncrona de servicios Web.
Object **send**(String signature, Object[] arguments)
Solicita la invocación asíncrona de servicios Web.

Class Variable

```
java.lang.Object
  bpcl. Variable
```

public class **Variable**

Permite guardar y recuperar la información de un objeto en particular. Mediante esta clase es posible definir un tipo de dato uniforme asociado a las variables locales definidas en cada método orquestado, permitiendo además que sus valores puedan modificarse.

Autor: Nancy N. Cova Suazo, BPCL version 1.0, CINVESTAV-IPN, 2005.

Attribute Summary	
Object **object**	Objeto asociado a una clase que referencia tipos de datos.

Method Summary	
Object **get**()	Recupera el valor de un objeto.
void **set**(Object value)	Asigna un valor a un objeto.

B.2. Etapa de Traducción

Class InformationWsdl

```
java.lang.Object
  bpcl. InformationWsdl
```

public class **InformationWsdl**

Es posible obtener información relevante de un servicio Web mediante la lectura de su respectiva descripción WSDL.

Autor: Nancy N. Cova Suazo, BPCL version 1.0, CINVESTAV-IPN, 2005.

Attribute Summary	
Vector **operations**	Vector de operaciones.
String **defineOperation**	Prototipo de una operación al estilo Java.

Method Summary
void **getEachOperations**(String wsdl, String portType, boolean isRemote) Consulta las operaciones de un servicio.
Vector **getInformation**(String wsdl, boolean isRemote) Obtiene información general de un servicio (su nombre, paquete de trabajo, url del wsdl, lista de operaciones).
void **getInputPartsMessage**(String wsdl, String operation, boolean isRemote) Genera el prototipo de una operación al estilo Java.
Vector **getNodesGeneralInformation**(Node node) Obtiene información general de la descripción wsdl de un servicio Web.
void **getNodesOperation**(Node node, String portType) Añade cada operación del servicio a un vector de operaciones.
void **getNodesRequest**(Node node, String operation) Consulta los datos de entrada de cada operación.
Vector **getOperations**(String wsdl, int mode, boolean isRemote) Obtiene las operaciones de un servicio.

Class GeneratePublishOWS

```
java.lang.Object
  bpcl.GeneratePublishOWS
```

public class **GeneratePublishOWS**

Verifica que la especificación escrita en BPCL sea válida. Mediante ésta se generarán todos los archivos de implementación y compilación necesarios para que una aplicación orquestada pueda verse como un servicio Web.

Autor: Nancy N. Cova Suazo, BPCL version 1.0, CINVESTAV-IPN, 2005.

Attribute Summary

String **pathWSDK**	Ruta de trabajo de wsdk.
String **WSDKLIB**	Ruta de trabajo de wsdk lib.
String **outPATH**	Ruta de trabajo para el código de salida.
String **urlRepository**	Localización del repositorio de servicios Web.
PrintWriter **out**	Flujo de salida para escribir los archivos de implementación.
int **portRmi**	Puerto por el cual el servidor rmi recibirá peticiones.

Constructor Summary

GeneratePublishOWS(String wsdk, String urlRepository, int portRmi)
Constructor que inicializa los datos de configuración necesarios por la infraestructura de coordinación para dar de alta un servicio Web orquestado.

Method Summary

void **createOutPATH**()
Crea el directorio de salida.

boolean **findChildScope**(Node tmp)
Indica si el nodo actual es padre de algún nodo scope.

void **generateAndPublish**(String xmlFile, String wsdlFile)
Se encarga de verificar que la especificación BPCL esté bien escrita.

void **generateAspectPartnerWrapper**(String partner, String pack, String nameProcess)
Crea los archivos de aspectos para cada participante del flujo de actividades.

Method Summary
void **generateBindingImpl**(String sw, String wsdlFile, String pack) Genera el archivo de implementación del servicio Web.
void **generateClientInvoke**(String nameProcess, String pack) Genera el archivo de invocación del cliente.
void **generateFilesCompile**(String nameProcess, String pack, Vector partners, String xmlFile, String wsdlFile) Crea los archivos de compilación del servidor rmi.
Node **generateFunctionDocument**(Node node, Hashtable varsTable, Hashtable parsTable, boolean isChildIte, String indIterator, Node runNode, Hashtable nodes, boolean isInit) Genera el código asociado a las operaciones disponibles a clientes. La inicialización de éstas debe estar decrita en el archivo de implementación del servidor rmi.
void **generateInitFunctionDocument**(String wsdlFile) Crea la función de inicialización en el archivo de implementación del servidor rmi, la cual permite que éste reciba peticiones.
Node **generateInitializeDocument**(Node node, String nameProcess, String pack, Hashtable varsTable, Node runNode, Hashtable nodes) Genera el archivo de inicialización del servicio Web.
String **generateNameAct**(String name, Hashtable nodes) Genera el nombre de una etiqueta única asociada a un nodo almacenado en una tabla hash.
Vector **generatePartnersDocument**(Node node, String nameProcess) Genera la sección de declaración de participantes en el archivo de implementación del servidor rmi.
void **generatePartnerWrapper**(String partner, String pack, String nameProcess) Crea los archivos de envoltura para cada participante del flujo de actividades.
Vector **generateRMIImpl**(Node node, String wsdlBroker, String nameProcess, String pack) Genera el archivo de implementación del servidor rmi. En este archivo la orquestación del flujo de actividades es ejecutada.
void **generateRMIInterface**(String nameProcess, String wsdlFile, String pack) Genera el archivo que define la interfaz del servidor rmi.
void **generateRMIServer**(String nameProcess, String pack) Genera el archivo servidor del servidor rmi.
String **getChildrenName**(Node tmp, Hashtable nodes) Regresa el nombre de las etiquetas de cada uno de los nodos hijos del nodo actual.

Method Summary
String **getChildrenNameWithoutScope**(Node tmp, Hashtable nodes) Regresa el nombre de las etiquetas de cada uno de los nodos hijos del nodo actual, excepto si el hijo es de tipo scope.
String **getNodeName**(Node tmp, Hashtable nodes) Regresa el nombre de la etiqueta de una actividad.
String **getProcessNameDocument**(Node node) Obtiene el nombre del proceso orquestado.
void **makeService**(String xmlFile, String wsdlFile) Crea el directorio donde los archivos de implementación generados por el traductor serán almacenados. Además, esta función invoca la función de generación y publicación de un servicio Web orquestado.

Class PublishService

```
java.lang.Object
  bpcl. PublishService
```

public class **PublishService**

Permite generar un archivo ejecutable mediante el cual son invocados una serie de comandos necesarios para publicar un servicio Web orquestado. Además registra el url de su descripción WSDL en el repositorio de servicios.

Autor: Nancy N. Cova Suazo, BPCL version 1.0, CINVESTAV-IPN, 2005.

Attribute Summary	
String **urlRepository**	Localización del repositorio de servicios Web.
String **pathWSDK**	Ruta de trabajo de wsdk.
String **WSDKLIB**	Ruta de trabajo de wsdk lib.
String **outPATH**	Ruta de trabajo para el código de salida.
String **fileImpl**	Archivo de implementación del servicio Web.
String **wsdl**	Archivo wsdl del servicio.
String **sw**	Nombre del servicio Web.
String **pack**	Paquete de trabajo del servicio.

Constructor Summary

PublishService(String urlRepository, String WSDKLIB, String pathWSDK, String outPATH) Constructor que inicializa los datos de configuración necesarios por la infraestructura de coordinación para dar publicar un servicio Web orquestado.

Method Summary

String **changeToPathWsdk**()	Cambia la ruta de trabajo actual por la ruta de trabajo de wsdk.
String **compileRMI**()	Compila los archivos del lado del servidor rmi.
String **copyFilesClient**()	Copia los archivos cliente a su directorio respectivo.
String **copyRmiClassesToWsdk**()	Copia las clases rmi a la ruta de wsdk.
String **createRMIdirectory**()	Genera el directorio de trabajo del servidor rmi.
String **executeFileCompile**()	Ejecuta el archivo de compilación del servicio Web.
String **generateEar**()	Genera el archivo de empaquetamiento (.ear) del servicio Web.
void **generateFileCompile**()	Genera el archivo de compilación del servicio Web.
String **generateService**()	Ejecuta el comando wsdl2webservice.
String **initializeService**()	Inicializa el servicio Web.
String **installService**()	Instala el servicio Web.
String **moveFileCompileWsdk**()	Copia el archivo de compilación del servicio Web a la ruta de wsdk.
String **moveFileImplWsdk**()	Copia el archivo de implementación del servicio Web a la ruta de wsdk.
String **moveFileWsdlWsdk**()	Copia el archivo wsdl a la ruta de wsdk.
void **publishServiceRepository**()	Publica la descripción del servicio orquestado en el repositorio.
String **runRmiregistry**()	Ejecuta rmiRegistry.
String **setDirectories**()	Inicializa las variables de entorno con los directorios de trabajo utilizados.
void **publish**(String fileServiceImpl, String fileWsdl, String sw, String pack, String urlWsdl, Vector partners)	Genera un archivo ejecutable por medio del cual es posible invocar una serie de comandos necesarios para publicar un servicio Web orquestado.
void **setVariables**(String fileImpl, String wsdl, String sw, String packagem, String urlWsdl)	Instancia las variables globales de esta clase.

Class MyRandom

```
java.lang.Object
  bpcl. MyRandom
```

public class **MyRandom**

Genera números enteros aleatorios en un intervalo determinado y mediante ella el traductor asigna identificadores únicos a cada una de las actividades del flujo.

Autor: Nancy N. Cova Suazo, BPCL version 1.0, CINVESTAV-IPN, 2005.

Method Summary	
int **nrand**(int module)	Retorna un número aleatorio que esté dentro de un intervalo definido.

B.3. Etapa de Invocación

Class WSDLInvoker

```
java.lang.Object
  bpcl. WSDLInvoker
```

public class **WSDLInvoker**

Clase que permite crear los mensajes SOAP necesarios para invocar las operaciones descritas en el documento WSDL de un servicio Web. Para lograrlo es necesario conocer algunos datos importantes de su descripción: URI del servicio, nombre de la operación, portName, etc.

Autor: Nancy N. Cova Suazo, BPCL version 1.0, CINVESTAV-IPN, 2005.

Method Summary	
Vector **getElementInputParts**(String operationName, String attr) Obtiene un vector con los elementos de entrada de las operaciones del servicio.	

Method Summary
Vector **getEncodingStyles**(Binding binding, String operationName) Obtiene el estilo de codificación para enlazar una operación.
String **getLocation**() Obtiene la localización del wsdl del servicio.
String **getLocationURI**(String port) Obtiene la localización del URI del servicio, dado un puerto.
String **getNamespaceURI**(Binding binding, String operationName) Obtiene el espacio de nombres del servicio, dado un enlace y una operación.
Operation **getOperation**(String name) Obtiene un objeto operación en base a su nombre.
List **getOperations**() Obtiene la lista de operaciones del servicio.
Operation **getPortOperation**(String operation, String portName) Obtiene el puerto de una operación en base a su nombre.
List **getPortOperations**(String portName) Obtiene una lista de puertos de operaciones en base a una localización wsdl.
Map **getServicePorts**() Obtiene todos los puertos del servicio.
SOAPMessage **senderSOAP**(String url, String nameinputop, String namespaceinput, Vector nameparts, Vector datos, isComplex, String complexName, String use, String soapAction) Envia un mensaje SOAP de petición para invocar un servicio Web.
void **setLocation**(String wsdlLocation) Asigna la localización wsdl del servicio.
String **SOAPToString**(SOAPMessage msg) Convierte un mensaje SOAP a cadena.

Class InvokeService

```
java.lang.Object
  bpcl. InvokeService
```

public class **InvokeService**

Obtiene la información requerida en la clase WSDLInvoker para invocar métodos de servicios. Además analiza el mensaje SOAP de respuesta para obtener sólo la información deseada.

Autor: Nancy N. Cova Suazo, BPCL version 1.0, CINVESTAV-IPN, 2005.

Attribute Summary	
String **wsdlLocation**	Localización de la descripción wsdl del servicio.

Constructor Summary
InvokeService(String wsdlLocation) Constructor que inicializa la localización de la descripción wsdl de un servicio Web involucrado en el flujo de actividades.

Method Summary
String **getRes**(Node node) Obtiene el resultado de la invocación a un método de un servicio, después de haber analizado su correspondiente mensaje SOAP de respuesta.
String **invoke**(String operationName, Vector part, Vector dataPart) Invoca un método de un servicio Web.
String **SOAPToString**(SOAPMessage msg) Convierte un mensaje SOAP de respuesta a una cadena.
Document **StringToDOM**(String messageSOAPstring) Convierte un mensaje SOAP de respuesta a un documento DOM.

Apéndice C

Archivos de implementación para el caso de estudio

En este apéndice mostraremos el conjunto de archivos de implementación que genera la infraestructura de coordinación [1] presentada en el capítulo 4 en base a la especificación BPCL para el caso de estudio *compra de artículos electrónicamente* propuesto en la sección 4.4.

Dicho archivos de implementación se generán en base al conjunto de entidades participantes en un esquema de coordinación aplicado a nuestra propuesta 4.3. Estas entidades y sus salidas correspondientes se listan a continuación.

- Servicio Web (Web Service): Archivo de implementación BindingImpl donde se redireccionan las peticiones al servidor RMI.
- Servidor RMI (RMI Server): Archivo de implementación donde se lleva a cabo la orquestación RMIImpl, la interfaz de los métodos orquestados RMIInterface, el servidor RMIServer para establecer la comunicación remota.
- Proveedores de Servicios (Suppliers): Envolturas SupplierWrapper para cada uno de los servicios Web elementales y sus respectivos *aspectos* AspectSupplierWrapper.
- Cliente (Client): Archivo de invocación del servicio orquestado Invoke, de inicialización Init, envoltura Wrapper y *aspecto* AspectWrapper respectivos.

[1] Nota: Los archivos de acompilación han sido omitidos.

C.1. Servicio Web (Web Service)

BrokerBindingImpl

- Mediante este archivo será posible redireccionar cada una de las peticiones cliente al servidor RMI.

```
1:  package cinves;
2:
3:  import java.net.*;
4:  import java.rmi.*;
5:
6:  public class BrokerBindingImpl implements cinves.BrokerPortType
7:  {
8:     public static BrokerRMIInterface orb = null;
9:
10:    public String getBestQuotation(String nameClient,String articles)
11:      throws java.rmi.RemoteException {
12:      return orb.getBestQuotation(nameClient,articles);
13:    }
14:
15:    public String orderPurchase(String url,String nameClient,String articles)
16:      throws java.rmi.RemoteException {
17:      return orb.orderPurchase(url,nameClient,articles);
18:    }
19:
20:    public String orderCancel(String url,String nameClient)
21:      throws java.rmi.RemoteException {
22:      return orb.orderCancel(url,nameClient);
23:    }
24:
25:    public void initProcessBroker()
26:      throws java.rmi.RemoteException {
27:      try
28:      {
29:        orb = (BrokerRMIInterface) Naming.lookup("//127.0.0.1:3838/BrokerRMIServer");
30:        orb.initProcessBroker();
31:      } catch(Exception e){ e.printStackTrace(); }
32:    }
33: }
```

Figura C.1: Listado del archivo BrokerBindingImpl.

C.2. Servidor RMI (RMI Server)

BrokerRMIServer

- Mediante este archivo el servidor RMI podrá atender y responder a peticiones realizadas por el servidor Web.

```
1:  package cinves;
2:
3:  import java.net.*;
4:  import java.rmi.*;
5:
6:  public class BrokerRMIServer
7:  {
8:     public static void main(String[] args) {
9:        try
10:       {
11:          BrokerRMIImpl impl = new BrokerRMIImpl();
12:          Naming.rebind("//127.0.0.1:3838/BrokerRMIServer", impl);
13:          System.out.println("BrokerRMIImpl ready --- Connected to BrokerRMI..!");
14:       } catch (Exception e) { e.printStackTrace(); }
15:    }
16: }
```

Figura C.2: Listado del archivo BrokerRMIServer.

BrokerRMIInterface

- Interfaz que muestra el conjunto de métodos orquestados disponibles a clientes.

```
1:  package cinves;
2:
3:  import java.rmi.*;
4:
5:  public interface BrokerRMIInterface extends Remote
6:  {
7:     public String getBestQuotation(final String nameClient,final String articles)
8:        throws RemoteException;
9:
10:    public String orderPurchase(final String url,final String nameClient,final String articles)
11:       throws RemoteException;
12:
13:    public String orderCancel(final String url,final String nameClient)
14:       throws RemoteException;
15:
16:    public void initProcessBroker()
17:       throws RemoteException;
18: }
```

Figura C.3: Listado del archivo BrokerRMIInterface.

BrokerRMIImpl

- Mediante este archivo el servidor RMI orquesta cada uno de los métodos definidos en la interfaz BrokerRMIInterface, en base a la especificación BPCL del caso de estudio. Además, aquí son definidas las entidades intermedias llamadas

Proxy, las cuales permitirán descubrir e invocar los servicios involucrados *Store* y *Blackboard* a través de sus envolturas de datos respectivas.

```
1:  package cinves;
2:
3:  import bpcl.*;
4:  import java.rmi.*;
5:  import java.rmi.server.*;
6:  import java.sql.*;
7:
8:  public class BrokerRMIImpl extends UnicastRemoteObject implements BrokerRMIInterface
9:  {
10:    public final Proxy[] stores = { new Proxy(new StoreWrapper("Store")),
11:                                    new Proxy(new StoreWrapper("Store")),
12:                                    new Proxy(new StoreWrapper("Store")) };
13:    public final Proxy store = new Proxy(new StoreWrapper("Store"));
14:    public final Proxy blackboard = new Proxy(new BlackboardWrapper("Blackboard"));
15:
16:    public BrokerRMIImpl() throws RemoteException {}
17:
18:    public void initProcessBroker()
19:    {
20:       try{
21:          Activity initF1 = new Activity("initF1") { public void run(){
22:             try{ init_getBestQuotation(); } catch(Exception e) { e.printStackTrace(); } } };
23:          Activity initF2 = new Activity("initF2") { public void run(){
24:             try{ init_orderPurchase(); } catch(Exception e) { e.printStackTrace(); } } };
25:          Activity initF3 = new Activity("initF3") { public void run(){
26:             try{ init_orderCancel(); } catch(Exception e) { e.printStackTrace(); } } };
27:
28:          CollaborativeActivity par = new Parallel("par",new Activity[]{initF1,initF2,initF3});
29:          par.started();
30:          System.out.println("\n Initialization of orchestrated methods is ready ... \n");
31:       } catch(Exception ex) {}
32:    }
33:
34:    public String getBestQuotation(final String nameClient, final String articles)
35:    {
36:       System.out.println(" The getBestQuotation is being executed ...");
37:       final Variable urlStore = new Variable();
38:       final Variable total = new Variable();
39:       final Variable best = new Variable();
40:       try {
41:          urlStore.set(getUrlStore());
42:          CollaborativeActivity[] mseq50 = new Sequential[stores.length];
43:          for(int jj=0; jj<stores.length; jj++)
44:          {
45:             final int j = jj;
46:             Activity call50 = new Activity("call50"+String.valueOf(j)){ public void run(){
47:                try{
48:                   Object r = stores[j].call("java.lang.Float
49:                               getQuotation(java.lang.String,java.lang.String,java.lang.String)",
50:                               new Object[]{((String[])urlStore.get())[j],nameClient,articles});
51:                   total.set(r);
52:                } catch(Exception e) { e.printStackTrace(); }
```

```
53:            } };
54:            Activity send50 = new Activity("send50"+String.valueOf(j)){ public void run(){
55:               try{
56:                  blackboard.send("void setMinCost(java.lang.String,java.lang.Float)",
57:                              new Object[]{((String[])urlStore.get())[j],(Float)total.get()});
58:               } catch(Exception e) { e.printStackTrace(); }
59:            } };
60:            CollaborativeActivity seq50 = new Sequential("seq50"+String.valueOf(j),
61:                        new Activity[]{call50,send50});
62:            mseq50[j] = seq50;
63:         }
64:         CollaborativeActivity par50 = new Parallel("par50",mseq50);
65:         Activity call28 = new Activity("call28"){ public void run(){
66:            try{
67:               Object r = blackboard.call("java.lang.String getBest()",new Object[]{});
68:               best.set(r);
69:            } catch(Exception e) { e.printStackTrace(); }
70:         } };
71:         CollaborativeActivity seq28 = new Sequential("seq28",new Activity[]{par50,call28});
72:         seq28.started();
73:         seq28.stopped();
74:      } catch(Exception exp){}
75:      System.out.println(" the processing has finished. \n");
76:      return (String)best.get();
77:   }
78:
79:   public String orderPurchase(final String url, final String nameClient, final String articles)
80:   {
81:      System.out.println(" The orderPurchase is being executed ...");
82:      final Variable invoice = new Variable();
83:      try {
84:         Activity call42 = new Activity("call42"){ public void run(){
85:            try{
86:               Object r = store.call("java.lang.String
87:                           buy(java.lang.String,java.lang.String,java.lang.String)",
88:                           new Object[]{url,nameClient,articles});
89:               invoice.set(r);
90:            } catch(Exception e) { e.printStackTrace(); }
91:         } };
92:         call42.started();
93:         call42.stopped();
94:      } catch(Exception exp){}
95:      System.out.println(" the processing has finished. \n");
96:      return (String)invoice.get();
97:   }
98:
99:   public String orderCancel(final String url, final String nameClient)
100:  {
101:     System.out.println(" The orderCancel is being executed ...");
102:     final Variable confirmation = new Variable();
103:     try {
104:        Activity call20 = new Activity("call20"){ public void run(){
105:        try{
106:             Object r = store.call("java.lang.String
107:                           cancelBuy(java.lang.String,java.lang.String)",
```

```
108:                          new Object[]{url,nameClient});
109:              confirmation.set(r);
110:           } catch(Exception e) { e.printStackTrace(); }
111:        } };
112:        call20.started();
113:        call20.stopped();
114:     } catch(Exception exp){}
115:     System.out.println(" the processing has finished. \n");
116:     return (String)confirmation.get();
117:  }
118:
119:  public String[] getUrlStore()
120:  { String[] urlStore = {"comprarDB1","comprarDB2","comprarDB3"}; return urlStore; }
121:
122:  public void init_getBestQuotation() throws Exception
123:  {
124:     CollaborativeActivity[] mseq20 = new Sequential[stores.length];
125:     for(int jj=0; jj<stores.length; jj++)
126:     {
127:        final int j = jj;
128:        Activity accept20 = new Activity("accept20"+String.valueOf(j)){ public void run(){
129:           try{
130:              Transaction t = new Transaction();
131:              Object[] p = stores[j].accept("java.lang.Float
132:                        getQuotation(java.lang.String,java.lang.String,java.lang.String)",t);
133:              Object r = stores[j].reply(t);
134:           } catch(Exception e) { e.printStackTrace(); }
135:        } };
136:        Activity receive20 = new Activity("receive20"+String.valueOf(j)){ public void run(){
137:           try{
138:              Object[] p = blackboard.receive("void
139:              setMinCost(java.lang.String,java.lang.Float)",new Transaction());
140:           } catch(Exception e) { e.printStackTrace(); }
141:        } };
142:        CollaborativeActivity seq20 = new Sequential("seq20"+String.valueOf(j),
143:        new Activity[]{accept20,receive20});
144:        mseq20[j] = seq20;
145:     }
146:     CollaborativeActivity par20 = new Parallel("par20",mseq20);
147:     Activity accept42 = new Activity("accept42"){ public void run(){
148:        try{
149:           Transaction t = new Transaction();
150:           Object[] p = blackboard.accept("java.lang.String getBest()",t);
151:           Object r = blackboard.reply(t);
152:        } catch(Exception e) { e.printStackTrace(); }
153:     } };
154:     CollaborativeActivity seq42 = new Sequential("seq42",new Activity[]{par20,accept42});
155:     seq42.started();
156:  }
157:
158:  public void init_orderPurchase() throws Exception
159:  {
160:     Activity accept16 = new Activity("accept16"){ public void run(){
161:        try{
162:           Transaction t = new Transaction();
```

```
163:            Object[] p = store.accept("java.lang.String
164:            buy(java.lang.String,java.lang.String,java.lang.String)",t);
165:            Object r = store.reply(t);
166:         } catch(Exception e) { e.printStackTrace(); }
167:      } };
168:      accept16.started();
169:   }
170:
171:   public void init_orderCancel() throws Exception
172:   {
173:      Activity accept34 = new Activity("accept34"){ public void run(){
174:         try{
175:            Transaction t = new Transaction();
176:            Object[] p = store.accept("java.lang.String
177:            cancelBuy(java.lang.String,java.lang.String)",t);
178:            Object r = store.reply(t);
179:            } catch(Exception e) { e.printStackTrace(); }
180:         } };
181:      accept34.started();
182:   }
183:}
```

Figura C.4: Listado del archivo BrokerRMIImpl.

C.3. Proveedores de servicios (Suppliers)

C.3.1. Store

StoreWrapper

- Archivo mediante el cual el desarrollador podrá ejecutar un código local a la aplicación en el momento en que la consulta por el servicio Web *Store* no se encuentre disponible. Además, es importante puntualizar que es aquí y en el *aspecto* AspectStoreWrapper donde deben realizarse las conversiones a tipos de datos primitivos correspondientes a cada método del servicio, debido a que fue utilizada el API Reflection de Java para realizar las invocaciones.

```
1:  package cinves;
2:  import bpcl.*;
3:
4:  public class StoreWrapper
5:  {
6:     public StoreWrapper(String nameService){ }
7:     public Float getQuotation(String url,String nameClient,String articles){ return new Float(0); }
8:     public String buy(String url,String nameClient,String articles){ return ""; }
9:     public String cancelBuy(String url,String nameClient){ return ""; }
10: }
```

Figura C.5: Listado del archivo StoreWrapper.

AspectStoreWrapper

- Este archivo representa el *aspecto* de búsqueda e invocación para el servicio Web *Store*, el cual fue diseñado para ser acoplado a la clase StoreWrapper, de tal manera que pueda trabajarse en forma local o remota.

```
1:  package cinves;
2:
3:  import bpcl.*;
4:  import java.util.*;
5:
6:  public aspect AspectStoreWrapper
7:  {
8:     String wsdl = "";
9:     InvokeService store = null;
10:
11:    void around(String nameService): execution(StoreWrapper.new(..)) && args(nameService)
12:    {
13:       try {
14:         InvokeService searchSw =
15:             new InvokeService("http://127.0.0.1:6080/Repository/services/RepositoryPort?wsdl");
16:         Vector pars = new Vector(); pars.addElement("nameService");
17:         Vector args = new Vector(); args.addElement(nameService);
18:         wsdl = searchSw.invoke("getUrlWsdlService",pars,args);
19:       } catch(Exception exp){ System.out.println("Connection to Repository was refused: "+exp);
20:                               proceed(nameService); }
21:
22:       if(wsdl.compareTo("Service Not Found") != 0)
23:          this.store = new InvokeService(wsdl);
24:       else
25:          proceed(nameService);
26:    }
27:
28:    Float around(String url,String nameClient,String articles):
29:          execution(* StoreWrapper.getQuotation(..)) && args(url,nameClient,articles)
30:    {
31:       if(wsdl.compareTo("Service Not Found") != 0 && store != null)
32:       {
33:          try {
34:             Vector pars = new Vector();
35:             pars.addElement("url"); pars.addElement("nameClient"); pars.addElement("articles");
36:             Vector args = new Vector();
37:             args.addElement(url); args.addElement(nameClient); args.addElement(articles);
38:             return new Float(Float.parseFloat(store.invoke("getQuotation",pars,args)));
39:          } catch(Exception exp){ System.out.println("Connection to Service was refused: "+exp);
40:                                  return proceed(url,nameClient,articles); }
41:       }
42:       else
43:          return proceed(url,nameClient,articles);
44:    }
45:
46:    String around(String url,String nameClient,String articles):
47:           execution(* StoreWrapper.buy(..)) && args(url,nameClient,articles)
48:    {
```

```
49:       if(wsdl.compareTo("Service Not Found") != 0 && store != null)
50:       {
51:          try {
52:             Vector pars = new Vector();
53:             pars.addElement("url"); pars.addElement("nameClient"); pars.addElement("articles");
54:             Vector args = new Vector();
55:             args.addElement(url); args.addElement(nameClient); args.addElement(articles);
56:             return store.invoke("buy",pars,args);
57:          } catch(Exception exp){ System.out.println("Connection to Service was refused: "+exp);
58:                                  return proceed(url,nameClient,articles); }
59:       }
60:       else
61:          return proceed(url,nameClient,articles);
62:    }
63:
64:    String around(String url,String nameClient):
65:           execution(* StoreWrapper.cancelBuy(..)) && args(url,nameClient)
66:    {
67:       if(wsdl.compareTo("Service Not Found") != 0 && store != null)
68:       {
69:          try {
70:             Vector pars = new Vector();
71:             pars.addElement("url"); pars.addElement("nameClient");
72:             Vector args = new Vector();
73:             args.addElement(url); args.addElement(nameClient);
74:             return store.invoke("cancelBuy",pars,args);
75:          } catch(Exception exp){ System.out.println("Connection to Service was refused: "+exp);
76:                                  return proceed(url,nameClient); }
77:       }
78:       else
79:          return proceed(url,nameClient);
80:    }
81: }
```

Figura C.6: Listado del archivo AspectStoreWrapper.

C.3.2. Blackboard

BlackboardWrapper

- Archivo mediante el cual el desarrollador podrá ejecutar un código local a la aplicación en el momento en que la consulta por el servicio Web *Blackboard* no se encuentre disponible.

```
1:  package cinves;
2:  import bpcl.*;
3:
4:  public class BlackboardWrapper
5:  {
6:      public BlackboardWrapper(String nameService) { }
7:      public void setMinCost(String url,Float costo) {  }
```

```
8:     public String getBest() { return ""; }
9:     public Float getMinCost() { return new Float(0); }
10: }
```

Figura C.7: Listado del archivo BlackboardWrapper.

Es aquí y en el *aspecto* AspectBlackboardWrapper donde deben realizarse las conversiones a tipos de datos primitivos correspondientes a cada método del servicio, debido a que fue utilizada el API Reflection de Java para realizar las invocaciones.

AspectBlackboardWrapper

- Este archivo representa el *aspecto* de búsqueda e invocación para el servicio Web *Blackboard*, el cual fue diseñado para ser acoplado a la clase BlackboardWrapper, de tal manera que pueda trabajarse en forma local o remota.

```
1:  package cinves;
2:
3:  import bpcl.*;
4:  import java.util.*;
5:
6:  public aspect AspectBlackboardWrapper
7:  {
8:     String wsdl = "";
9:     InvokeService blackboard = null;
10:
11:    void around(String nameService):
12:          execution(BlackboardWrapper.new(..)) && args(nameService)
13:    {
14:       try {
15:         InvokeService searchSw =
16:             new InvokeService("http://127.0.0.1:6080/Repository/services/RepositoryPort?wsdl");
17:         Vector pars = new Vector(); pars.addElement("nameService");
18:         Vector args = new Vector(); args.addElement(nameService);
19:         wsdl = searchSw.invoke("getUrlWsdlService",pars,args);
20:       } catch(Exception exp){ System.out.println("Connection to Repository was refused: "+exp);
21:                               proceed(nameService); }
22:
23:       if(wsdl.compareTo("Service Not Found") != 0)
24:          this.blackboard = new InvokeService(wsdl);
25:       else
26:          proceed(nameService);
27:    }
28:
29:    void around(String url,Float costo):
30:         execution(* BlackboardWrapper.setMinCost(..)) && args(url,costo)
31:    {
32:       if(wsdl.compareTo("Service Not Found") != 0 && blackboard != null)
33:       {
34:          try {
```

```
35:             Vector pars = new Vector();
36:             pars.addElement("url"); pars.addElement("costo");
37:             Vector args = new Vector();
38:             args.addElement(url); args.addElement(costo.toString());
39:             blackboard.invoke("setMinCost",pars,args);
40:          } catch(Exception exp){ System.out.println("Connection to Service was refused: "+exp);
41:                                  proceed(url,costo); }
42:       }
43:       else
44:          proceed(url,costo);
45:    }
46:
47:    String around(): execution(* BlackboardWrapper.getBest(..))
48:    {
49:       if(wsdl.compareTo("Service Not Found") != 0 && blackboard != null)
50:       {
51:          try {
52:             return blackboard.invoke("getBest",new Vector(),new Vector());
53:          } catch(Exception exp){ System.out.println("Connection to Service was refused: "+exp);
54:                                  return proceed(); }
55:       }
56:       else
57:          return proceed();
58:    }
59:
60:    Float around(): execution(* BlackboardWrapper.getMinCost(..))
61:    {
62:       if(wsdl.compareTo("Service Not Found") != 0 && blackboard != null)
63:       {
64:          try {
65:             return new Float(Float.parseFloat(blackboard.invoke("getMinCost",
66:                                                    new Vector(),new Vector())));
67:          } catch(Exception exp){ System.out.println("Connection to Service was refused: "+exp);
68:                                  return proceed(); }
69:       }
70:       else
71:          return proceed();
72:    }
73: }
```

Figura C.8: Listado del archivo AspectBlackboardWrapper.

C.4. Cliente (Client)

BrokerInit

- Archivo mediante el cual se restringe el orden en el que serán aceptadas las peticiones (a métodos orquestados) por parte del cliente. Es decir, el caso de uso está diseñado para aceptar siempre una mejor cotización *getBestQuotation* y despúes alternativamente es posible recibir una orden de compra *orderPurchase* o de cancelación *orderCancel*.

```
1:  package cinves;
2:  import bpcl.*;
3:
4:  public class BrokerInit
5:  {
6:     public void init_Broker(final Proxy broker)
7:     {
8:        Activity accept6 = new Activity("accept6"){ public void run(){
9:           try{
10:             Transaction t = new Transaction();
11:             Object[] p = broker.accept("java.lang.String
12:                         getBestQuotation(java.lang.String,java.lang.String)",t);
13:             Object r = broker.reply(t);
14:          } catch(Exception e) { e.printStackTrace(); }
15:       } };
16:       Activity accept31 = new Activity("accept31"){ public void run(){
17:          try{
18:             Transaction t = new Transaction();
19:             Object[] p = broker.accept("java.lang.String
20:                         orderPurchase(java.lang.String,java.lang.String,java.lang.String)",t);
21:             getGroup().stopGroupButNotMe();
22:             Object r = broker.reply(t);
23:          } catch(Exception e) { e.printStackTrace(); }
24:       } };
25:       Activity accept9 = new Activity("accept9"){ public void run(){
26:          try{
27:             Transaction t = new Transaction();
28:             Object[] p = broker.accept("java.lang.String
29:                         orderCancel(java.lang.String,java.lang.String)",t);
30:             getGroup().stopGroupButNotMe();
31:             Object r = broker.reply(t);
32:          } catch(Exception e) { e.printStackTrace(); }
33:       } };
34:       CollaborativeActivity alt9 = new Parallel("alt9",new Activity[]{accept31,accept9});
35:       accept31.setGroup(alt9);
36:       accept9.setGroup(alt9);
37:       CollaborativeActivity seq9 = new Sequential("seq9",new Activity[]{accept6,alt9});
38:       seq9.started();
39:    }
40: }
```

Figura C.9: Listado del archivo BrokerInit.

BrokerWrapper

- Archivo mediante el cual el desarrollador podrá ejecutar un código local a la aplicación en el momento en que la consulta por el servicio Web *Broker* no se encuentre disponible. Es aquí y en el *aspecto* AspectBrokerWrapper donde deben realizarse las conversiones a tipos de datos primitivos correspondientes a cada método del servicio.

```
1:  package cinves;
2:
3:  import bpcl.*;
4:
5:  public class BrokerWrapper
6:  {
7:     public BrokerWrapper(String nameService) { initProcessBroker(); }
8:     public String getBestQuotation(String nameClient,String articles) { return ""; }
9:     public String orderPurchase(String url,String nameClient,String articles) { return ""; }
10:    public String orderCancel(String url,String nameClient) { return ""; }
11:    public void initProcessBroker() {  }
12: }
```

Figura C.10: Listado del archivo BrokerWrapper.

AspectBrokerWrapper

- Este archivo representa el *aspecto* de búsqueda e invocación para el servicio Web *Broker*, el cual fue diseñado para ser acoplado a la clase BrokerWrapper, de tal manera que pueda trabajarse en forma local o remota.

```
1:  package cinves;
2:
3:  import bpcl.*;
4:  import java.util.*;
5:
6:  public aspect AspectBrokerWrapper
7:  {
8:     String wsdl = "";
9:     InvokeService broker = null;
10:
11:    void around(String nameService):
12:         execution(BrokerWrapper.new(..)) && args(nameService)
13:    {
14:       try {
15:         InvokeService searchSw =
16:             new InvokeService("http://127.0.0.1:6080/Repository/services/RepositoryPort?wsdl");
17:         Vector pars = new Vector(); pars.addElement("nameService");
18:         Vector args = new Vector(); args.addElement(nameService);
19:         wsdl = searchSw.invoke("getUrlWsdlService",pars,args);
20:       } catch(Exception exp){ System.out.println("Connection to Repository was refused: "+exp);
21:                               proceed(nameService); }
22:
23:       if(wsdl.compareTo("Service Not Found") != 0)
24:          this.broker = new InvokeService(wsdl);
25:       proceed(nameService);
26:    }
27:
28:    String around(String nameClient,String articles):
29:           execution(* BrokerWrapper.getBestQuotation(..)) && args(nameClient,articles)
30:    {
31:       if(wsdl.compareTo("Service Not Found") != 0 && broker != null)
32:       {
```

```
33:         try {
34:            Vector pars = new Vector();
35:            pars.addElement("nameClient"); pars.addElement("articles");
36:            Vector args = new Vector();
37:            args.addElement(nameClient); args.addElement(articles);
38:            return broker.invoke("getBestQuotation",pars,args);
39:         } catch(Exception exp){ System.out.println("Connection to Service was refused: "+exp);
40:                                 return proceed(nameClient,articles); }
41:      }
42:      else
43:         return proceed(nameClient,articles);
44:   }
45:
46:   String around(String url,String nameClient,String articles):
47:          execution(* BrokerWrapper.orderPurchase(..)) && args(url,nameClient,articles)
48:   {
49:      if(wsdl.compareTo("Service Not Found") != 0 && broker != null)
50:      {
51:         try {
52:            Vector pars = new Vector();
53:            pars.addElement("url"); pars.addElement("nameClient"); pars.addElement("articles");
54:            Vector args = new Vector();
55:            args.addElement(url); args.addElement(nameClient); args.addElement(articles);
56:            return broker.invoke("orderPurchase",pars,args);
57:         } catch(Exception exp){ System.out.println("Connection to Service was refused: "+exp);
58:                                 return proceed(url,nameClient,articles); }
59:      }
60:      else
61:         return proceed(url,nameClient,articles);
62:   }
63:
64:   String around(String url,String nameClient):
65:          execution(* BrokerWrapper.orderCancel(..)) && args(url,nameClient)
66:   {
67:      if(wsdl.compareTo("Service Not Found") != 0 && broker != null)
68:      {
69:         try {
70:            Vector pars = new Vector(); pars.addElement("url");
71:            pars.addElement("nameClient");
72:            Vector args = new Vector(); args.addElement(url);
73:            args.addElement(nameClient);
74:            return broker.invoke("orderCancel",pars,args);
75:         } catch(Exception exp){ System.out.println("Connection to Service was refused: "+exp);
76:                                 return proceed(url,nameClient); }
77:      }
78:      else
79:         return proceed(url,nameClient);
80:   }
81:
82:   void around(): execution(* BrokerWrapper.initProcessBroker(..))
83:   {
84:      if(wsdl.compareTo("Service Not Found") != 0 && broker != null)
85:      {
86:         try {
87:            broker.invoke("initProcessBroker",new Vector(),new Vector());
```

```
88:          } catch(Exception exp){ System.out.println("Connection to Service was refused: "+exp);
89:                                  proceed(); }
90:       }
91:       else
92:          proceed();
93:    }
94: }
```

Figura C.11: Listado del archivo AspectBrokerWrapper.

InvokeBroker

- Mediante este archivo un cliente podrá insertar el código necesario para invocar métodos orquestados provistos por el servicio Web *Broker*, a través de la función *run*. Las líneas de código generadas en éste serán indispensables para poder establecer la comunicación con dicho servicio.

```
1:  package cinves;
2:  import bpcl.*;
3:
4:  public class InvokeBroker
5:  {
6:     public static void main(String[] args) {
7:        try {
8:           InvokeBroker obj = new InvokeBroker();
9:           Proxy broker = new Proxy(new BrokerWrapper("Broker"));
10:          obj.initialize(broker);
11:          obj.run(broker);
12:       } catch (Exception e) { e.printStackTrace(); }
13:    }
14:
15:    public void initialize(Proxy broker) {
16:       BrokerInit startOWS = new BrokerInit();
17:       startOWS.init_Broker(broker);
18:    }
19:
20:    public void run(final Proxy broker) throws Exception {
21:       /* Here insert your own code application */
22:    }
23: }
```

Figura C.12: Listado del archivo InvokeBroker.

C.5. Generación del servicio Web orquestado

Para que un desarrollador pueda generar un servicio Web orquestado, será necesario que éste cuente con una especificación escrita en BPCL del proceso (4.14) y con

una descripción WSDL que defina su interfaz. Además, tendrá que definir una clase principal *Main* mediante la cual será lanzado el proceso de generación del mismo.

C.5.1. Descripción del servicio Web

Broker.wsdl

- Descripción WSDL del servicio a ser orquestado, mediante este archivo el proveedor de servicios *Broker* especificará el conjunto de métodos que estarán disponibles a clientes, les dará a conocer las especificaciones necesarias para poder establecer una correcta comunicación con él.

```
1:  <?xml version="1.0" encoding="UTF-8"?>
2:
3:  <!-- Namespaces -->
4:  <definitions name="Broker"
5:      targetNamespace="http://cinves/Broker.wsdl"
6:      xmlns="http://schemas.xmlsoap.org/wsdl/"
7:      xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
8:      xmlns:tns="http://cinves/Broker.wsdl"
9:      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
10:     xmlns:xsd1="http://cinves/Broker.xsd1">
11:
12:     <!-- Types -->
13:     <types>
14:         <xsd:schema
15:             targetNamespace="http://cinves/Broker.xsd1"
16:             xmlns="http://schemas.xmlsoap.org/wsdl/"
17:             xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
18:             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
19:             xmlns:tns="http://cinves/Broker.wsdl"
20:             xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
21:             xmlns:xsd="http://www.w3.org/2001/XMLSchema"
22:             xmlns:xsd1="http://cinves/Broker.xsd1">
23:         </xsd:schema>
24:     </types>
25:
26:     <!-- Messages (Input and Output) -->
27:     <message name="initProcessBrokerResponse">   </message>
28:
29:     <message name="orderPurchaseResponse">
30:         <part name="invoice" type="xsd:string"/>
31:     </message>
32:
33:     <message name="orderCancelRequest">
34:         <part name="url" type="xsd:string"/>
35:         <part name="nameClient" type="xsd:string"/>
36:     </message>
37:
38:     <message name="orderCancelResponse">
39:         <part name="confirmation" type="xsd:string"/>
```

```
40:     < /message>
41:
42:      <message name="getBestQuotationRequest">
43:          <part name="nameClient" type="xsd:string"/>
44:          <part name="articles" type="xsd:string"/>
45:      </message>
46:
47:      <message name="initProcessBrokerRequest">   </message>
48:
49:      <message name="getBestQuotationResponse">
50:          <part name="best" type="xsd:string"/>
51:      </message>
52:
53:      <message name="orderPurchaseRequest">
54:          <part name="url" type="xsd:string"/>
55:          <part name="nameClient" type="xsd:string"/>
56:          <part name="articles" type="xsd:string"/>
57:      </message>
58:
59:      <!-- PortTypes -->
60:      <portType name="BrokerPortType">
61:
62:          <!-- Operations -->
63:          <operation name="getBestQuotation">
64:              <input message="tns:getBestQuotationRequest"/>
65:              <output message="tns:getBestQuotationResponse"/>
66:          </operation>
67:
68:          <operation name="orderPurchase">
69:              <input message="tns:orderPurchaseRequest"/>
70:              <output message="tns:orderPurchaseResponse"/>
71:          </operation>
72:
73:          <operation name="orderCancel">
74:              <input message="tns:orderCancelRequest"/>
75:              <output message="tns:orderCancelResponse"/>
76:          </operation>
77:
78:          <operation name="initProcessBroker">
79:              <input message="tns:initProcessBrokerRequest"/>
80:              <output message="tns:initProcessBrokerResponse"/>
81:          </operation>
82:      </portType>
83:
84:      <!-- Binding -->
85:      <binding name="BrokerBinding" type="tns:BrokerPortType">
86:         <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
87:
88:         <operation name="getBestQuotation">
89:              <soap:operation
90:                  soapAction="capeconnect:Broker:BrokerPortType#getBestQuotation"/>
91:              <input>
92:                  <soap:body use="literal"/>
93:              </input>
94:              <output>
```

```
95:                 <soap:body use="literal"/>
96:             </output>
97:         </operation>
98:
99:         <operation name="orderPurchase">
100:            <soap:operation soapAction="capeconnect:Broker:BrokerPortType#orderPurchase"/>
101:            <input>
102:                <soap:body use="literal"/>
103:            </input>
104:            <output>
105:                <soap:body use="literal"/>
106:            </output>
107:        </operation>
108:
109:        <operation name="orderCancel">
110:           <soap:operation soapAction="capeconnect:Broker:BrokerPortType#orderCancel"/>
111:            <input>
112:                <soap:body use="literal"/>
113:            </input>
114:            <output>
115:                <soap:body use="literal"/>
116:            </output>
117:        </operation>
118:
119:        <operation name="initProcessBroker">
120:            <soap:operation
121:                soapAction="capeconnect:Broker:BrokerPortType#initProcessBroker"/>
122:            <input>
123:                <soap:body use="literal"/>
124:            </input>
125:            <output>
126:                <soap:body use="literal"/>
127:            </output>
128:        </operation>
129:    </binding>
130:
131:    <!-- Service -->
132:    <service name="Broker">
133:        <port binding="tns:BrokerBinding" name="BrokerPort">
134:            <soap:address location="http://localhost:6080/Broker/services/BrokerPort"/>
135:        </port>
136:    </service>
137:
138: </definitions>
```

Figura C.13: Listado del archivo Broker.wsdl.

C.5.2. Archivo de ejecución

Main

- Un desarrollador tendrá que escribir este archivo para especificar algunos datos de configuración necesarios para generar la aplicación orquestada. Dichos datos serán la ruta de trabajo donde se encuentre instalado WSDK, la ubicación del

repositorio en el cual serán realizadas las consultas a servicios y finalmente el puerto por medio del cual el servidor RMI atenderá peticiones. En este momento el usuario deberá contar con la especificación BPCL del servicio y su descripción WSDL, de tal manera que pueda ejecutar la instrucción *makeService* de la clase *GeneratePublishOWS* para realizar la generación, publicación y puesta en marcha del mismo.

```
1:  import bpcl.*;
2:
3:  public class Main
4:  {
5:     public static void main(String[] args) throws Exception
6:     {
7:        GeneratePublishOWS obj = new GeneratePublishOWS(
8:                                 "..\\wsdk",
9:                                 "http://127.0.0.1:6080/Repository/services/RepositoryPort?wsdl",
10:                                3838);
11:      obj.makeService("Broker.xml","Broker.wsdl");
12:    }
13: }
```

Figura C.14: Listado del archivo Main.

Apéndice D

Descripción del CD que acompaña a la tesis

El disco compacto que acompaña a este trabajo de tesis tiene como objetivo poner al alcance el material resultante del mismo: artículos, documento de tesis, presentaciones (congresos, seminarios) y los elementos necesarios para desarrollar aplicaciones orquestadas utilizando el esquema de coordinación presentado.

La estructura que conforma al CD es la siguiente:

- **Artículos**
  Artículo presentado en "*The 2nd International Conference on Electrical and Electronics Engigeering and of the XI Conference on Electrical Engineering 2005*" [33].
  Ver [../articulos/articuloCIE2005.pdf].
- **Documento de tesis**
  Ver [tesisNNCS.pdf]
- **Presentaciones**
  - Congresos. 2nd ICEEE and XI CEE 2005.
    Ver [../presentaciones/congresos/presCIE05.pdf]
  - Seminarios
    - 1^{er} Seminario
      Ver [../presentaciones/seminarios/semI/presTesis1raSemI.pdf], [presTesis2daSemI.pdf]

- 2^{do} Seminario
  Ver [../presentaciones/seminarios/semII/presTesis1raSemII.pdf], [presTesis2daSemII.pdf]
- 3^{er} Seminario
  Ver [../presentaciones/seminarios/semIII/presTesis1raSemIII.pdf], [presTesis2daSemIII.pdf], [presTesis3raSemIII.pdf]

- **Sistema de coordinación**
  Incluye un manual de referencia que indica los pasos a seguir para desarrollar una aplicación orquestada, aunado con un instructivo que señala los requerimientos necesarios en software para su funcionamiento.
  Ver [../sistemaCoordinacion/manualReferencia.pdf], [requerimientosSoftware.pdf]

  - Biblioteca de coordinación
    Código fuente de la bliblioteca **bpcl.jar** y documentación.
    Ver [../sistemaCoordinacion/bpcl1.0/doc/index.html]
  - Bibliotecas
    Este directorio contiene las blibliotecas mínimas necesarias para ejecutar cualquier aplicación orquestada empleando nuestro esquema de coordinación: bpcl.jar y aspectjrt.jar.
  - Esquema XML del lenguaje de coordinación
    Ver [../sistemaCoordinacion/esquemaXML/bpcllanguage.xsd]
  - Caso de estudio
    Contiene una guía de ejecución donde además se especifican los elementos requeridos para que la aplicación Compra de Artículos Electrónicamente funcione.
    Ver [../sistemaCoordinacion/casoDeEstudio/guiaEjecucion.pdf]
    - Requerimientos
      Este directorio contiene (*a*) el conjunto de *scripts* SQL para dar de alta la base de datos del servicio *Store* y del repositorio de servicios Web *Repository*, y (*b*) los archivos necesarios para instalar los servicios *Repository, Store* y *Blackboard* involucrados en el flujo de trabajo.
    - Generar aplicación
      Los archivos contenidos en esta carpeta son (*a*) la descripción WSDL de la compra de artículos, (*b*) especificación BPCL del proceso de negocios, (c) esquema XML del lenguaje, (d) archivo de configuración y generación, y finalmente (e) archivos de compilación y ejecución.

- Cliente prueba
  Archivos ejemplo para ejecutar una aplicación cliente en la cual se realice una compra de artículos y una cancelación.

- Software. Conjunto de herramientas de programación necesarias para el funcionamiento de aplicaciones basadas en el esquema de coordinación presentado.

  1. Lenguaje de programación
     - Java 2 SDK Runtime environment, SE v1.4.2
     - AspectJ 1.2
  2. Manejador de base de datos
     - MySQL Connector/ODBC 3.51
     - MySQL Servers and Clients 3.23.51
  3. Entorno de trabajo para implementar servicios Web
     - Eclipse SDK 3.0
     - WebSphere SDK for Web Services 5.0 (WSDK 5.0)

Apéndice E

Glosario

- **Actividad**
  Representa reglas bien definidas de un proceso de negocios. Una actividad primitiva (simple) puede verse como aquella que provee una funcionalidad elemental, en contraste con las actividades estructuradas que controlan el flujo de trabajo desde una perspectiva global, especificando qué actividades serán ejecutadas y en qué orden [6].

- **AO4BPEL**
  Desarrollado por el Grupo de Tecnología de Software en Darmstadt, Alemania. AO4BPEL es una extensión orientada a aspectos de BPEL [21]. Dicha extensión es útil para mejorar la modularidad y adaptabilidad de las especificaciones de composición de servicios Web y brindar soporte para adaptaciones dinámicas tales como la coordinación y el acoplamiento o desacoplamiento de *aspectos* en tiempo de ejecución. BPEL fue escogido como base para hacer que cada actividad definida se convirtiera en un punto de unión y ésto permitiera ejecutar servicios por medio de *aspectos* [25].

- **AspectJ**
  Extensión orientada a aspectos de Java, ésto significa que cualquier programa Java será válido en AspectJ. El compilador de AspectJ genera ficheros *class* conforme a la especificación Java *byte-code*, por lo que una máquina virtual de Java (JVM) puede ejecutarlos [20].

- **Aspecto**
  Una unidad modular del programa que aparece y afecta a otras unidades modulares del mismo [19].

- **BPCL**
  Acrónimo de *Business Process Coordination Language*, es un lenguaje composicional por medio del cual la descripción de un proceso orquestado puede realizarse sin necesidad de especificar detalles de distribución de recursos. Además, cuenta con una infraestructura de coordinación que entre otras tareas será la encargada de ejecutar procesos definidos en BPCL.

- **BPEL4WS**
  Acrónimo de *Business Process Execution Language for Web Services* fue desarrollado por IBM, Microsoft y BEA Systems. BPEL4WS es un lenguaje estático composicional orientado al modelado de procesos orquestados, el cual cuenta con su propio motor de ejecución conocido como BPWS4J [21].

- **Flujo de control**
  Describe el orden en que serán ejecutadas las actividades de un proceso de negocios. Dicho orden será especificado en términos de construcciones usuales de programación como la composición: secuencial, concurrente y condicional [6].

- **Flujo de datos**
  En un proceso de negocios, describe la información intercambiada entre una actividad y otra [6].

- **Infraestructura de coordinación**
  Conjunto de elementos o servicios que se consideran necesarios para la creación y funcionamiento de procesos de negocios orquestados. Mediante ella es posible generar, publicar y ejecutar procesos coordinados, opcionalmente a partir de su respectiva especificación BPCL y su descripción WSDL.

- **Invasividad de código**
  Fénomeno que aparece cuando son incorporadas funcionalidades adicionales a un sistema, haciéndolo poco entendible y modular. Los principales problemas que genera son.

  - *Código enredado* (*Tangling*). Occurre cuando en una unidad del programa (clase o método) son implementados múltiples requerimientos de manera entrelazada.
  - *Dispersión de código* (*Scattering*). Occurre cuando en múltiples módulos se incluye el código que implementa a un mismo requerimiento [19].

- **Modelo de comunicación *Rendezvous***
  Rendezvous [29] es un modelo de comunicación y sincronización que permite a dos procesos concurrentes, el solicitante (cliente) y el llamado (servidor) intercambiar datos de forma coordinada. El proceso que solicita un método debe esperar en el punto de reencuentro (*rendezvous*) hasta que el proceso llamado llegue allí. Sin embargo, el proceso llamado puede llegar a dicho punto antes que el solicitante y entonces él deberá esperar de igual forma para poder continuar su ejecución. Lo anterior significa que existe la posibilidad de que ambos procesos ejecuten algún otro procesamiento antes y después del punto de reencuentro, lo cual en el modelo RPC no es posible en el lado del servidor.

- **Modelo de comunicación RPC**
  La noción de llamadas a procedimientos remotos (Remote Procedure Calls, RPC) [29] se introdujo con el objetivo de ofrecer un mecanismo estructurado de alto nivel para realizar la comunicación entre procesos pertenecientes a sistemas distribuidos. Mediante RPC un proceso de un sistema (proceso invocador) podrá solicitar un procedimiento de un proceso servidor. El proceso que llama enviará un mensaje al proceso servidor y a continuación esperará (bloqueará su ejecución) hasta recibir la respuesta de su petición. En el lado del servidor, éste esperará por la petición del procedimiento. Cuando dicha petición sea recibida, el servidor creará un nuevo hilo de ejecución que se encargará de extraer los parámetros correspondientes de la llamada, ejecutarla y devolver el control al proceso servidor. Dicho proceso enviará como respuesta el valor de retorno al proceso invocador. Cuando el mensaje sea retornado, el proceso que llama finalizará su espera, extraerá el valor de retorno y continuará su ejecución.

- **Orquestación de servicios Web**
  Proceso que consiste en relacionar, organizar y administrar las interacciones entre los servicios Web referentes a la lógica del proceso de negocios [15].

- **Proceso**
  Definido mediante una serie de actividades cuyo fin es la ejecución de una tarea determinada [6].

- **Programación orientada a aspectos**
  Reciente paradigma de programación introducido en 1995 gracias a la colaboración de Cristina Videira Lopes, Karl J. Lieberherr, Gregor Kiczales y su grupo de trabajo. Dicho paradigma plantea un modelo viable para evitar o controlar lo más posible la invasividad de código en una aplicación. Los objetivos que

persigue son principalmente el de separar conceptos en un proceso de negocios y el de minimizar sus dependencias. Con el primer objetivo se consigue que las funcionalidades no básicas del sistema sean encapsuladas y con el segundo se consigue la reducción del acoplamiento entre los distintos elementos.

- **Punto de corte *(pointcut)***
  Constructor donde se especifican los puntos de unión a alcanzar, pueden verse como las reglas para identificar dichos puntos. Además, mediante un punto de corte será posible capturar la información referente al contexto de un punto de unión [18].

- **Punto de unión *(joint point)***
  Sitio perfectamente identificable en la ejecución de un programa, es aquí donde se determinará el momento en el cual serán acoplados los *aspectos* [18].

- **Redefinición de comportamiento *(advice)***
  Código que debe ejecutarse como respuesta a un punto de unión especificado en un punto de corte, el cual puede realizarse antes, durante o después de dicho punto. Ésto quiere decir que es posible modificar la ejecución del código en el punto de unión definido, reemplazarlo o ignorarlo [18].

- **Reflexión**
  Mecanismo de los sistemas computacionales el cual les permite razonar, actuar por sí mismos y modificar dinámicamente su comportamiento. Existen tres tipos de reflexión:

  - *Por introspección*. El sistema es capaz de observar y razonar acerca de sus elementos, pero no puede modificarlos.
  - *Estructural*. El sistema puede consultar y modificar su estructura en tiempo de ejecución.
  - *Por Comportamiento*. El sistema puede manipular y modificar su comportamiento en tiempo de ejecución.

- **Servicio Web**
  Componente de software funcional o aplicación Web identificada a través de un URI, cuya interfaz y uso es capaz de ser definida, descrita y descubierta mediante artefactos XML, la cual además soporta interacciones directas con otras aplicaciones de software usando mensajes XML y protocolos basados en Internet [10].

- **SOAP**
  Acrónimo de *Simple Object Access Protocol*, es un protocolo basado en XML que se usa para codificar los mensajes de petición, respuesta y errores en las peticiones a métodos de servicios Web envíados a través de la red [4].

- **Transparencia de localidad de recursos**
  Omisión de la ubicación de servicios Web (recursos) al momento de especificar el proceso de orquestación de los mismos.

- **UDDI**
  Directorio de servicios Web distribuido el cual permite a los proveedores de servicios Web dar a conocer sus ofertas de una forma estándar de tal manera que los clientes puedan consultarlos e invocarlos [5].

- **Weaving process**
  A diferencia de los esquemas tradicionales de programación que utilizan un compilador para obtener aplicaciones ejecutables, al desarrollar aplicaciones orientadas a aspectos deberá ser utilizado un proceso adicional llamado *weaving process*, el cual permitirá acoplar *aspectos* a las aplicaciones. El proceso de acoplamiento puede ser realizado *estáticamente*, es decir, en tiempo de carga y de compilación, o *dinámicamente* en tiempo de ejecución [18].

- **WSDK**
  Entorno de programación que provee un conjunto de herramientas para el diseño, implementación y ejecución de aplicaciones Java basadas en servicios Web. La funcionalidad de WSDK se basa en especificaciones abiertas como SOAP, WSDL y UDDI [12].

- **WSDL**
  Acrónimo de *Web Services Description Language*, WSDL es un lenguaje XML usado para describir la interfaz de programación de un servicio Web [3].

- **WS-Coordination (WS-C)**
  Soporta, integra y unifica diferentes modelos de coordinación permitiendo a diferentes sistemas interoperar. Provee además mecanismos estándares para crear y registrar servicios usando los protocolos definidos en WS-T [22].

- **WS-Transaction (WS-T)**
  Coordina la ejecución de operaciones distribuidas en el ambiente de servicios

Web a través de la definición de protocolos para transacciones atómicas, de negocios, compensación, etc. [23].

- **XML Scheme**
  Permite verificar sintácticamente la validez o correctitud de cualquier descripción que ha sido escrita en base a un conjunto de etiquetas en documentos XML. En un esquema XML es posible definir: (a) los elementos que pueden aparecer en un documento y sus atributos, los tipos de datos correspondientes y los valores que pueden tomar por defecto, (b) cómo pueden anidarse, es decir si existe un orden de aparición restringido, (c) el número permitido de elementos hijos y por último (d) si un elemento puede ser vacío o no.

Bibliografía

[1] A. M. R. Quintero, "Visión General de la Programación Orientada a Aspectos," Departamento de Lenguajes y Sistemas Informáticos, Facultad de Informática y Estadística, Universidad de Sevilla, Tech. Rep., Diciembre 2000.

[2] F. Curbera, W. A. Nagy, and S. Weerawarana, "Web Services: Why and How," IBM T.J. Watson Research Center, Tech. Rep., August 2001.

[3] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. (2001) "Web Services Description Language (WSDL) 1.1". [Online]. Available: http://www.w3.org/TR/wsdl.

[4] W3C. (2003) "SOAP Version 1.2 Part 1: Messaging Framework". [Online]. Available: http://www.w3.org/TR/soap12-part1/.

[5] OASIS. (2002) "UDDI Version 2.04 API Specification". [Online]. Available: http://uddi.org/pubs/ProgrammersAPI_v2.htm.

[6] J. Yang, "Web Service Componentization: Towards Service Reuse and Specialization," INFOLAB, Tilburg University, Tech. Rep., October 2003.

[7] T. Andrews, F. Curbera, I. Trickovic, S. Weerawarana, and et al. (2003) "Business Process Execution Language for Web Services version 1.1". [Online]. Available: http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/.

[8] (2002) "BPWS4J, A platform for creating and executing BPEL4WS processes.". [Online]. Available: http://www.alphaworks.ibm.com/tech/bpws4j.

[9] "BPEL4J: BPEL for Java," White Paper, BEA Systems and IBM, March 2004.

[10] W3C. (2004) "Web Services Architecture". ws-arch.pdf. [Online]. Available: http://www.w3.org/TR/ws-arch.

[11] E. Armstrong, J. Ball, D. Green, I. Evans, and et al. (2004, December) "The J2EE^{TM} 1.4 Tutorial". [Online]. Available: http://java.sun.com/j2ee/1.4/download.html#tutorial.

[12] IBM. "IBM WebSphere SDK for Web Services (WSDK) Version 5.0.1". [Online]. Available: http://www-106.ibm.com/developerworks/webservices/wsdk/.

[13] Web Service Interoperability Organization. "Web Service Interoperability Organization (WS-I)". [Online]. Available: http://www.ws-i.org/.

[14] Sun Microsystems. "Java Web Services Developer Pack 1.6, Tutorials and Product Documentation". [Online]. Available: http://java.sun.com/webservices/jwsdp/index.jsp.

[15] C. Peltz, "Web Services Orchestration," Hewlett Packard, Coorporation, Tech. Rep., January 2003.

[16] G. Chafle, S. Chandra, V. Mann, and M. G. Nanda, "Decentralized Orchestration of Composite Web Services," IBM, India Research Laboratory New Delhi, India, Tech. Rep., 2004.

[17] C. Peltz. (2004) "Web Services Orchestration and Choreohraphy, a look at WSCI and BPEL4WS". wsoac.pdf. [Online]. Available: http://www.sys-con.com/webservices/.

[18] T. Elrad and R. E. Filman, "Aspect-Oriented Programming," *Special Section Of Communications of the ACM*, vol. 44, no. 10, pp. 29–32, 2001.

[19] G. Kiczales, J. Irwin, J. Lamping, C. V. Lopes, J.-M. Loingtier, and et al., "Aspect-Oriented Programming," Xerox Palo Alto Research Center, Tech. Rep., 1997.

[20] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersen, J. Palm, and W. G. Griswold, "An overview of AspectJ," in *Proceedings European Conference on Object-Oriented Programming*, ser. LNCS, vol. 2072, 2004, pp. 327–353.

[21] R. Khalaf, N. Mukhi, and S. Weerawarana, "Service-Oriented Composition in BPEL4WS," in *Proceedings of the Twelfth International World Wide Web Conference*, Budapest, Hungery, May 2003.

[22] IBM, Arjuna Technologies Ltd., BEA Systems, Hitachi Ltd., and IONA Technologies. (2005) "Web Services Coordination (WSCoordination), Version 1.0". WS-Coordination.pdf. [Online]. Available: http://www.iona.com/devcenter/articles/pub_aug_2005_RC1.1/.

[23] Arjuna Technologies Ltd., BEA Systems, Hitachi Ltd., IBM, and IONA Technologies. (2005) "Web Services Atomic Transaction (WS-AtomicTransaction), Version 1.0". WS-AtomicTransaction.pdf. [Online]. Available: http://www.iona.com/devcenter/articles/pub_aug_2005_RC1.1/.

[24] T. Elrad, R. E. Filman, and G. Kiczales, "Disscussing Aspects of AOP," *Special Section Of Communications of the ACM*, vol. 44, no. 10, pp. 33–38, 2001.

[25] A. Charfi and M. Mezini, "Aspect-Oriented Web Service Composition with AO4BPEL," in *Proceedings of European Conference on Web Services ECOWS 2004*, ser. LNCS, vol. 3250, 2004.

[26] M. A. Cibrán and B. Verheecke, "Dynamic Business Rules for Web Service Composition," Vrije Universiteit Brussel, Tech. Rep., January 2003.

[27] B. Verheecke and M. A. Cibrán, "Modularizing Web Services Management with AOP," System and Software Engineering Lab Vrije Universiteit Brussel, Tech. Rep., 2003.

[28] J. P. Fuente, S. S. Alonso, O. S. Martínez, and L. J. Aguilar, "RAWS: Reflective Engineering for Web Services," Computer Languages and Systems Department, Pontifical University of Salamanca Madrid campus Spain, Tech. Rep., January 2005.

[29] G. R. Andrews, *Concurrent Programming, Principles and Practice.* The Benjamin/Cummings Publishing, Inc., 1991.

[30] Sun Microsystems. "Java Remote Method Invocation (RMI), Tutorials and Product Documentation". [Online]. Available: http://java.sun.com/j2se/1.4.2/docs/guide/rmi/index.html.

[31] "Java Platform, Standard Edition, v 1.3.1, Reflection API Specification". [Online]. Available: http://java.sun.com/j2se/1.3/docs/api/java/lang/reflect/package-summary.html.

[32] (2004) "API JDOM v1.0, API Specification". [Online]. Available: http://www.jdom.org/docs/apidocs/.

[33] N. Cova Suazo and J. O. Olmedo Aguirre, "Aspect-Oriented Web Services Orchestration," in *Proceedings of the 2nd International Conference on Electrical and Electronics Engigeering and of the XI Conference on Electrical Engineering*, Mexico City, Mexico, September 2005.

[34] Sun Microsystems. "JVM Tool Interface Version 1.0, Documentation". [Online]. Available: http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/jvmti.html.

[35] M. Pérez Reséndiz and J. O. Olmedo Aguirre, "Dynamic Invocation of Web Services by using Aspect-Oriented programming," in *Proceedings of the 2nd International Conference on Electrical and Electronics Engigeering and of the XI Conference on Electrical Engineering*, Mexico City, Mexico, September 2005.

[36] M. Pérez Reséndiz, "Selección e integración dinámica de servicios Web orientada a aspectos," Tesis de maestría, CINVESTAV-IPN, Octubre 2005.

Índice

Actividad, 18
AO4BPEL, 26–27
AspectJ, 21–23

Biblioteca de clases bpcl.jar, 58–59
    Etapa de Coordinación, 58
    Etapa de Invocación, 59
    Etapa de Traducción, 59
BPCL, 34–49
BPEL4J, 26
BPEL4WS, 24–25

Estándares, 12

Flujo de control, 18
Flujo de datos, 18

Infraestructura de coordinación, 51–76
Invasividad de código, 20

Modelo de comunicación, 42
    Rendezvous, 46–49
    RPC, 44–46

Orquestación de servicios Web, 18–19

Proceso, 18
Programación orientada a aspectos, 20–23
    Ventajas, 3
Punto de corte, 21
Punto de unión, 21

Redefinición de comportamiento, 21
Reflexión, 28–29
    Estructural, 29
    Por comportamiento, 29
    Por introspección, 29

Servicios Web, 10–12
    Arquitectura, 10
    Características, 11
SOAP, 13–14

Transparencia de localidad de recursos, 5

UDDI, 15–16

Weaving process, 21
WS-Coordination, 25
WS-Transaction, 25
WSDK, 16–18
WSDL, 12–13

XML Scheme, 40–41

"*Nunca consideres el estudio como una obligación, sino como una oportunidad para penetrar en el bello y maravilloso mundo del saber*".

Albert Einstein.