UNIVERSIDADE FEDERAL DE SANTA CATARINA

DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

CURSO DE BACHARELADO EM SISTEMAS DE INFORMAÇÃO

GERENCIAMENTO DE TRANSAÇÕES DISTRIBUÍDAS EM WEB SERVICES

Priscilla Francielle Poleza

Trabalho de conclusão de curso submetido à Universidade Federal de Santa Catarina como parte dos requisitos para obtenção do grau de Bacharel em Sistemas de Informação.

Florianópolis – SC

2007/1

Priscilla Francielle Poleza

GERENCIAMENTO DE TRANSAÇÕES DISTRIBUÍDAS EM WEB SERVICES

Trabalho de conclusão de curso submetido à Universidade Federal de Santa Catarina como parte dos requisitos para obtenção do grau de Bacharel em Sistemas de informação.

Orientador: Prof. Frank Augusto Siqueira, Dr.
Universidade Federal de Santa Catarina
frank@inf.ufsc.br

Banca examinadora

Prof. Ronaldo dos Santos Mello, Dr.
Universidade Federal de Santa Catarina
ronaldo@inf.ufsc.br

Prof. Fernando Augusto da Silva Cruz, Dr.
Universidade Federal de Santa Catarina
cruz@inf.ufsc.br

“Dedico este trabalho aos meus pais, ao meu marido

e aos meus familiares”

AGRADECIMENTOS

Gostaria de agradecer a todos que me auxiliaram, direta ou indiretamente, na conclusão deste trabalho.

Agradeço primeiramente a Deus, por estar sempre presente na minha vida e sem o qual nada é possível.

Agradeço aos meus pais, Algemiro Poleza e Maria Eli da Silva Poleza, pelo amor, apoio e incentivo em toda está longa caminhada.

Agradeço também a força, incentivo, ombro amigo e grande compreensão do meu grande amor, Valmor Ataíde Machado Junior, que me tranqüilizou e acalmou meus anseios nos momentos mais difíceis e decisivos

Obrigada ao meu orientador, pela ótima oportunidade, que me propiciou muitos aprendizados e novos conhecimentos. Obrigada também por ser meu guia e grande incentivador neste trabalho final.

Finalmente, não poderia deixar de agradecer aos meus colegas da faculdade. Em especial a Karla Maria Garcia, o Guilherme Vieira, o Paulo Henrique Michels e o Bruno Cavaler Ghisi, pela luz, grande incentivo e apoio que me deram quando me vi perdida em uma área na qual não tinha muita experiência.

A todos, um sincero obrigada!

Sumário

Lista de Figuras

Lista de Abreviaturas

- **2PC** – Protocolo de Efetivação (*Commit*) em Duas Fases.
- **3PC** – Protocolo de Efetivação (*Commit*) em Três Fases.
- **ACID** – é um acrônimo definido pelas iniciais das propriedades de Atomicidade, Consistência, Isolamento e Durabilidade. As quais conjuntamente garantem a integridade dos dados.
- **API** – *Application Program Interface*: conjunto de rotinas, protocolos e ferramentas para construção de aplicações.
- **AT** – *Atomic Transaction:* Transação atômica.
- **B2B** – *Business-to-business:* Transações entre empresas;
- **B2C/C2B** – *Business-to-consumer/Consumer-to-business:* Transações entre empresas e consumidores.
- **B2G/G2B** – *Business-to-government/Government-to-business:* Transações entre empresas e governo.
- **BA** – *Business Activity:* Transação de negócios.
- **BTP** – Protocolo projetado para suportar aplicações independentes de localização e administração, que necessitam de suporte transacional diferente do suportado pelas transações ACID.
- **C2C** – *Consumer-to-consumer:* Transações entre consumidores finais.
- **CE** – Comércio eletrônico.
- **CORBA** – *Common Object Request Broker Architecture*: arquitetura do *Object Management Group* que permite a comunicação entre objetos em diferentes plataformas.

- **DCE** – *Distributed Computing Environment*: ambiente desenvolvido pelo *The Open Group* para criação de aplicações distribuídas que rodam em diferentes plataformas.
- **DCOM** – *Distributed Component Object Model*: protocolo de objetos remotos, da Microsoft.
- **DTD** – *Document Type Definitio:* define o padrão de subelementos aceitos em um elemento XML.
- **DTP** – Processamento de transações distribuídas.
- **G2C/C2G** – *Government-to-consumer/consumer-to-government:* Transações entre governo e consumidores.
- **G2G** – *Government-to-government:* transações entre entidades do governo.
- **HTML** – *HyperText Markup Language*: linguagem usada para criar documentos na Web.
- **HTTP** – *HyperText Transfer Protocol*: protocolo que define como mensagens são formatadas e transmitidas e quais ações servidores Web e navegadores devem tomar em resposta a vários comandos.
- **IP** – *Internet Protocol*: protocolo Internet para troca de pacotes.
- **Java RMI** – *Java Remote Method Invocation*: conjunto de protocolos desenvolvidos pela Sun que permitem a comunicação remota entre objetos Java.
- **JTA** – *API Java Transaction:* especifica interfaces para a demarcação de transações em aplicações escritas na linguagem Java.
- **OASIS** – *Organization for the Advancement of Structured Information Standards*: é uma organização que foi criada para estabelecer padrões

para Serviços Web. Formada pela Epicentric Inc., Hewlett-Packard, IBM, Macromedia Inc., entre outros.

- **RM** – *Resource Manager:* Gerenciador de recursos.
- **RPC** – *Remote Procedure Call*: Chamada Remota de Procedimento.
- **SOA** – *Service-oriented Architecture*: Arquitetura Orientada a Serviços.
- **SOAP** – Protocolo de transporte de mensagens entre serviços Web.
- **UDDI** – *Universal Description, Discovery and Integration*: Integração, Descobrimento e Descrição Universal. Especificação de descobrimento para serviços Web.
- **URI** – *Uniform Resource Identifier*: Identificador Uniforme de Recursos, termo genérico para todos os tipos de nomes e endereços que se referem a objetos na Web.
- **URL** – *Uniform Resource Locator*: Localizador Uniforme de Recursos, endereço global de documentos e outros recursos na Web.
- **W3C** – *World Wide Web Consortium*: organização internacional cujo objetivo é desenvolver padrões abertos para a Web.
- **WS** – *Web Services*: Serviços Web.
- **WS-C** – *Web Services Coordination:* Protocolo de coordenação de web services.
- **WS-Tx** – *Web Services Transaction:* Protocolo de transações de web services
- **WSDL** – *Web Services Description Language*: Linguagem de Descrição de Serviços Web.
- **WSDLtoJava** – Ferramenta, fornecida pelo framework Apache Axis, que gera o cliente a partir do arquivo WSDL do web service.

- **XML** – *Extensible Markup Language*: linguagem de tags de marcação baseada em texto.
- **XML *Schema*** – Esquema XML: modelo para descrever a estrutura dos dados XML.
- **XSD** – XML *Schema Definition*: é um complemento ao DTD. Ele define uma série de tipos internos, como *string, integer, date, boolean*, etc.

RESUMO

A arquitetura de Serviços Web surgiu com o intuito de facilitar a interligação/integração de sistemas heterogêneos. Ela é baseada em tecnologias padronizadas, em particular XML e http, é fracamente acoplada e garante a interoperabilidade na comunicação entre aplicações, ou seja, é independente de sistema operacional e de linguagem de programação. Porém, por se tratar de uma tecnologia muito nova alguns aspectos ainda precisam ser melhorados, como por exemplo, a realização do controle das transações distribuídas. Este trabalho consiste na realização de uma pesquisa a respeito do funcionamento e utilização da implementação dos protocolos *WS-Coordination* e *WS-AtomicTransaction,* proposta pelo projeto Apache Kandula; e na utilização desta implementação em um cenário de aplicação específico a fim de demonstrar como pode ser feito um controle de transações distribuídas de maneira a garantir a consistência dos dados das aplicações envolvidas. Para isso definiu-se e implementou-se um cenário de aplicação, no qual ocorrem transações atômicas entre uma aplicação e dois serviços web. E utilizou-se o serviço de coordenação, implementado pelo Kandula, para coordenar as transações distribuídas realizadas no cenário implementado e a implementação do JTA, realizada pelo Apache Geronimo, para realizar o controle local das transações pelas aplicações que implementam os serviços web envolvidos no cenário.

Palavras-chave: serviços web, transações distribuídas, transações atômicas, *WS-Coordination*, *WS-AtomicTransaction* e Apache Kandula.

Abstract

Web services architecture was created to facilitate the interconnection and the interaction of heterogeneous systems. Web services are based on standard technologies, particularly XML and http, are weakly connected and guarantee internal communication operations among applications; that is, web services are independent of the operational system and the language programming. However, because it is such a new technology, certain aspects still need to be improved, for example, the achievement of the control of distributed transactions. This monograph entails the development of a research about the working and the use of the implementation of WS-Coordination and WS-Atomic Transaction protocols proposed by the Apache Kandula project; the research contemplates such use within a specific application scenario, in order to demonstrate how to control distributed transactions in a way that may ensure the consistency of the data of the relevant application. For that purpose, an application scenario was defined and implemented, in which atomic transactions take place between one application and two web services. The co-ordination service (implemented by Kandula) was used to co-ordenate the distributed transactions carried out within the implemented scenario, and the implementation of JTA (by Apache Geronimo) was used to carry out the local control of transactions by the applications that implement the web services involved in the scenario.

Key words: *web services, distributed transactions, atomic transactions, WS-Coordination, WS-AtomicTransaction and Apache Kandula.*

1 Introdução

A internet desde o seu surgimento (década de 70), vem evoluindo consideravelmente e popularizando-se cada vez mais como um grande meio de difusão de informações. Sua evolução tem trazido grandes benefícios para a sociedade de forma geral, em especial para as empresas, que passaram a dispor de um meio de comunicação com seus clientes e parceiros de negócio.

Com o intuito de sobreviver e obter sucesso no contexto da economia de mercado atual, as empresas vêm percebendo a necessidade de interligar seus processos de negócio, bem como trocar informações com fornecedores, clientes e parceiros através de processos de negócios externos. No entanto, para que a empresa consiga atender de forma eficaz e eficiente as exigências dos clientes e de estarem atentas às mudanças de mercado e/ou ameaças da concorrência, é necessário que seja possível fazer uma fácil integração entre os diferentes sistemas que implementam os processos utilizados por ela.

A arquitetura de Web Services possibilita a integração de sistemas heterogêneos devido às seguintes características: é baseada em tecnologias padronizadas, em particular XML e HTTP; é fracamente acoplada; e oferece interoperabilidade, ou seja, é independente de sistema operacional e de linguagem de programação. Porém, para que esta tecnologia seja adotada de forma mais efetiva comercialmente é necessário que se tenha uma infra-estrutura confiável para o seu desenvolvimento e implantação, ou seja, é necessário investir ainda na melhoria de alguns pontos.

A realização de consórcios entre grandes empresas como Hewlett-Packard, Oracle, BEA Systems, Microsoft e IBM, tem sido feito com o objetivo

de especificar e padronizar protocolos para solucionar alguns dos problemas ainda existentes na tecnologia de Web Services. Um dos problemas existentes na arquitetura de Web Services são transações distribuídas. Este assunto é abordado neste trabalho.

1.1 Descrição do Problema

Sistemas computacionais geralmente executam operações sobre dados, e podem vir a alterá-los. Em alguns casos, a operação pode não ser realizada da forma correta, e nestas situações não se pode permitir que as alterações resultantes da operação mal sucedida sejam realmente efetivadas. Isto pode tornar inconsistentes os dados envolvidos na operação, que ainda poderão ser utilizados por operações subseqüentes, levando a resultados errôneos.

Um sistema computacional confiável deve evitar qualquer inconsistência ou corrupção de dados, garantindo uma resposta correta ou ao menos uma indicação de erro no sistema ao usuário. A quantidade de erros possíveis em sistemas distribuídos aumenta consideravelmente em virtude de erros de comunicação, falhas de elementos de rede e impossibilidade de determinar um estado possível de falha. Foi neste contexto que o surgiu o termo transação, a qual consiste em uma seqüência de operações que atua sobre dados, possivelmente alterando-os. No caso de alguma operação falhar, todos os dados devem voltar ao seu estado original, assegurando que uma transação mal sucedida não cause qualquer efeito sobre o sistema (ELSMARI; NAVATHE, 2000).

O requisito confiabilidade é imprescindível na maioria dos sistemas computacionais existentes, como por exemplo, sistemas bancários, sistemas de comércio eletrônico, sistemas eleitorais, sistemas de diagnóstico médico e

muitos outros. Várias propriedades e protocolos já foram definidos a fim de garantir que os sistemas sejam confiáveis a nível das transações realizadas. No entanto, a tecnologia de Web Services possui necessidades diferentes das transações tradicionais em virtude das suas características.

Como por exemplo, transações entre serviços Web podem necessitar de um tempo muito maior para serem concluídas do que uma transação tradicional e, portanto, as soluções adotadas para tratar uma transação de curta duração (tradicional), não podem ser usadas para tratar uma transação de longa duração.

Outra característica a ser levada em consideração, quando tratamos de Web Services, é a possibilidade que uma aplicação possui de envolver vários serviços web em uma única atividade. Neste caso, uma transação que envolve serviços disponibilizados por aplicações totalmente independentes deve ser coordenada de modo que a efetivação da transação depende da confirmação da execução correta dos serviços por ela utilizados.

Portanto, a fim de garantir que as transações envolvendo Web Services sejam tratadas de forma adequada, um consórcio de várias empresas especificou os protocolos *Web Services Coordination* (WS-C) (Orchard et al. 2005) e *Web Services Transaction* (WS-Tx) (Cox et al. 2004), que visam propor uma infra-estrutura de suporte a transações distribuídas efetuadas por aplicações que utilizam a tecnologia de Web Services.

Há várias implementações destes protocolos disponíveis no mercado atualmente, sendo uma delas o Projeto Apache Kandula, que implementa os protocolos *WS-Coordination* e *WS-AtomicTransaction* sobre o *framework Apache Axis,* a qual é objeto de estudo neste trabalho. A escolha por esta

implementação deve-se ao fato de tratar-se de uma implementação *de* código aberto, desenvolvida sobre o *framework Apache Axis*, o qual possui ampla aceitação e adoção na construção de WS.

1.2 Objetivos

O principal objetivo deste trabalho consiste no estudo e utilização da implementação dos protocolos *WS-Coordination* e *WS-AtomicTransaction* proposta pelo projeto Kandula em uma aplicação real, a fim de demonstrar a utilização de uma infra-estrutura de suporte a transações distribuídas em um ambiente que utilize a tecnologia de Web Services.

Os seguintes objetivos específicos foram levantados:

- Estudar a arquitetura de Web Services;
- Estudar os padrões *Web Services Coordination* (WS-C) e *Web Services Transaction* (WS-Tx);
- Estudar a infra-estrutura de suporte a transações distribuídas em Web Services implementada pelo projeto Apache Kandula;
- Especificar e implementar uma aplicação utilizando a infra-estrutura proposta pelo Kandula;
- Demonstrar e avaliar o uso da infra-estrutura proposta através da aplicação que será implementada.

1.3 Justificativa

Apesar de ainda ser uma tecnologia recente, os Web Services vêm se tornando uma promessa como o futuro padrão de mercado para interconexão de sistemas. No entanto, para que isto se concretize de fato, muitas pesquisas e especificações estão sendo realizadas por órgãos como o *World Wide Web*

Consortium (W3C), para suprir algumas deficiências desta tecnologia, como por exemplo, questões de segurança, disponibilidade e confiabilidade.

Para que um sistema seja considerado confiável, ele precisa garantir a consistência e integridade dos seus dados, ou seja, ele deve garantir que o usuário irá receber somente respostas corretas ou, no caso de falha, uma indicação de erro do sistema. (ELSMARI; NAVATHE, 2000). Foi devido à necessidade de implementação da confiabilidade nos sistemas computacionais que surgiu o conceito de transação.

Conforme Belli (2005), a idéia que embasa as transações é a mesma que embasa um contrato judicial, no qual várias partes acordam um resultado. Se alguma das partes não puder cumprir uma das partes do acordo, este é invalidado. Propriedades (**A**tomicidade, **C**onsistência, **I**solamento e **D**urabilidade - ACID) Jajodia e Kérschberg (1997), e protocolos (Protocolo de Efetivação em 2 fases (2PC), de Efetivação em 3 fases (3PC), etc.), já foram definidos a fim de garantir sucesso em transações tradicionais. No entanto as transações em Web Services possuem necessidades diferentes que precisam ser tratadas de forma específica.

A idéia de se estudar e demonstrar a utilização de uma infra-estrutura de suporte a transações distribuídas em Web Services é de tentar contribuir para a ampla adoção desta tecnologia. Com base na experiência de implementação de gerência de transações distribuídas em Web services realizada neste trabalho, espera-se que o leitor tome conhecimento de como e de quais ferramentas e tecnologias são necessárias para fazer um controle eficaz de transações distribuídas em cenários que envolvam serviços Web. Adicionalmente, é possível, com a leitura deste trabalho, conhecer como é

realizado o gerenciamento de transações proposto pelos protocolos WS-Coordination e WS-AtomicTransaction e implementado pelo projeto Apache Kandula, e utilizar este conhecimento para compará-los com outras propostas já realizadas.

1.4 Organização do Texto

O primeiro capítulo introduz o assunto abordado neste trabalho, apresentando uma descrição do problema referente ao controle de transações distribuídas em Web Services, assim como a importância de que se tenha uma solução confiável para este problema. Foram apresentados também os motivos que nos levaram a escolher este tema, bem como os objetivos gerais e específicos deste trabalho.

O segundo capítulo consiste em uma revisão da literatura relacionada à tecnologia de Web Services, na qual são apresentados definições, histórico, aplicabilidade e vantagens desta tecnologia. São introduzidos também os conceitos e os principais padrões da arquitetura Web Services, tais como, SOAP, WSDL e UDDI.

O terceiro capítulo também consiste em uma revisão da literatura, no entanto, aborda conceitos e definições referentes a transações, transações distribuídas e transações distribuídas em Web Services. Neste capítulo são descritos também os problemas encontrados no gerenciamento de transações distribuídas em Web Services e algumas das soluções já propostas, bem como a solução escolhida para ser utilizada neste trabalho.

No quarto capítulo, primeiramente faz-se uma descrição do cenário de aplicação que será implementado a fim de demonstrar como pode ser feito o gerenciamento de transações distribuídas em Web Services utilizando-se a

implementação dos protocolos WS-Coordination e WS-AtomicTransaction proposta pelo Apache Kandula. Em seguida, apresentam-se detalhes do funcionamento da implementação dos protocolos utilizados no controle das transações, assim como as ferramentas e tecnologias utilizadas no desenvolvimento deste trabalho.

Finalmente, as conclusões e resultados deste trabalho são apontados no capítulo 5. Neste capítulo também são apresentadas sugestões para trabalhos futuros.

2 Web Services

A criação, a popularização e a evolução da Internet trouxeram muita facilidade e comodidade aos seus usuários. No entanto, à medida que a rede foi crescendo, novas necessidades foram surgindo. Como por exemplo, a necessidade de integração dos sistemas computacionais de uma empresa, os quais implementam os processos de negócio da mesma, bem como a troca de informações com seus fornecedores, clientes e sócios, a fim de que a mesma consiga sobreviver e obter sucesso no contexto da economia de mercado atual que tem exigido que serviços sejam disponibilizados via Web.

Foi neste contexto que surgiu a tecnologia Web Services, a qual visa à integração de sistemas computacionais e de serviços de forma que esta integração seja independente da localização geográfica destes sistemas e serviços, da plataforma sobre a qual os mesmos são executados, da linguagem de programação em que foram implementados, etc.

No final do ano 2000, ano do seu surgimento, as empresas Oracle, HP, Sun, IBM, BEA e Microsoft (maiores fornecedoras de software para TI do mundo), anunciaram as suas intenções de utilizar os padrões Web Services (SOAP, WSDL e UDDI) em seus produtos. Desde então esta tecnologia tem sido alvo de muitas pesquisas e investimentos a fim de que se possa especificar e padronizar protocolos para solucionar alguns dos problemas ainda existentes em sua arquitetura, para que se tenha uma infra-estrutura confiável para o seu desenvolvimento e implantação de modo que ela seja adotada de forma mais efetiva comercialmente.

2.1 Arquitetura Orientada a Serviço (SOA)

Os Web Services foram desenvolvidos com base na Arquitetura Orientada a Serviços (*Service-oriented Architeture – SOA)*. Como o próprio nome já diz, o modelo SOA baseia-se em serviços, os quais segundo Papazoglou (2003), podem ser definidos como funções de negócios implementadas em software e acessíveis através das suas interfaces. A função da interface é fornecer o mecanismo pelo qual os serviços se comunicam com outros serviços e aplicações e apresentar o conjunto de operações disponíveis para invocação dos clientes do serviço.

Segundo Papazoglou (2003), a arquitetura SOA consiste em uma maneira de se reorganizar um conjunto de aplicações de software, previamente desenvolvidos, e de fornecer uma infra-estrutura de suporte a um conjunto de serviços interconectados, os quais são acessíveis através de interfaces padrões e de protocolos de mensagens. Esta arquitetura é aplicável quando múltiplas aplicações são executadas sobre tecnologias e plataformas distintas, e precisam comunicar-se entre si. Portanto, uma vez que os web services consistem em uma implementação da arquitetura SOA sobre a WEB, eles são independentes de plataformas e linguagens de programação.

O W3C conceitua SOA como um conjunto de componentes que podem ser invocados e ter suas descrições de interface publicadas e descobertas. As tecnologias RMI, DCOM, CORBA e DCE são exemplos de sistemas SOA. (HAAS, BROWN, 2004).

O modelo SOA baseia-se na interação entre agentes de software, os quais assumem papéis distintos entre si, tais como: *provider* (provedor), *broker* (mediador ou intermediador) e *requisitor* (consumidor ou invocador). A *Figura 1*

ilustra a interação entre estes agentes, as quais envolvem a publicação de informações sobre os serviços, busca dos serviços disponíveis e a conexão/ligação (*binding*) com estes serviços. A interação entre os agentes é realizada através de troca e mensagens.

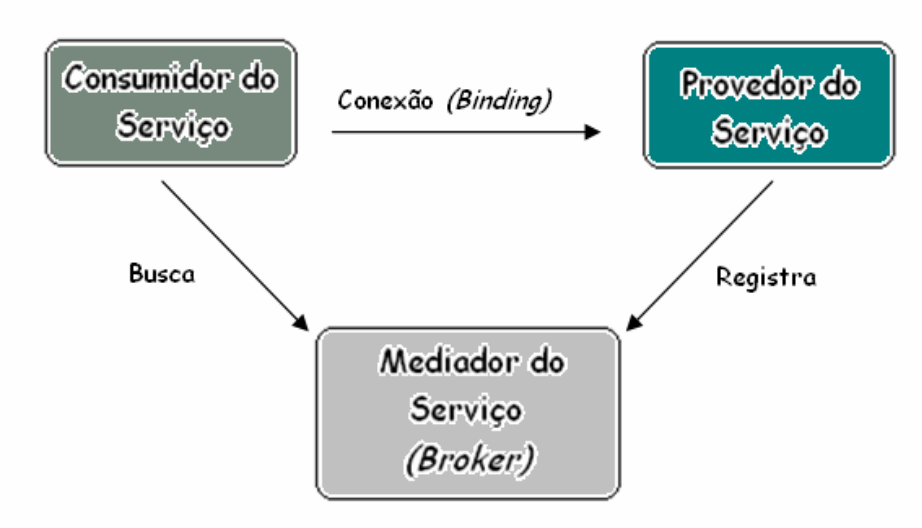

Figura 1 – Arquitetura Orientada a Serviços.

Em cenários típicos de Web Services, um provedor torna seu serviço disponível e publica as informações que descrevem a sua interface através de um registro no *broker.* O consumidor pesquisa no *broker* os serviços que deseja. Em seguida, o *broker* fornece ao consumidor as informações de localização e de contrato do serviço solicitado, as quais são utilizadas para conectar (*bind)* o consumidor ao serviço. (LEANDRO, 2005).

Para que as interações descritas acima possam ocorrer, um sistema SOA deve prover os seguintes componentes de arquitetura. (LEANDRO, 2005):

- **Transporte:** este componente representa os formatos e protocolos usados na comunicação com um serviço. O formato especifica os tipos de dados usados na codificação das mensagens. Os protocolos utilizados são o de transferência, o qual especifica as semânticas de

aplicações que controlam uma transferência, e o protocolo de transporte que realiza a transferência propriamente dita.

- **Descrição:** este componente representa as linguagens que são utilizadas na descrição de um serviço, provendo as informações necessárias para acessá-lo. Uma linguagem de descrição deve prover também um modo de especificar o contrato do serviço, incluindo as operações que ele realiza e quais seus parâmetros.
- **Descobrimento:** este componente representa os mecanismos utilizados para registrar, anunciar e encontrar um serviço e sua descrição. Ele fornece suporte à conexão do consumidor com o serviço oferecido pelo provedor.

Em suma, SOA possui as seguintes características:

- Um serviço que implementa a lógica de negócio e expõem estas lógicas de negócio através de interfaces bem definidas;
- Um mediador que publica os serviços através de interfaces para permitir que os clientes descubram os serviços oferecidos;
- Clientes (incluindo clientes que também podem oferecer serviços) os quais descobrem os serviços usando os mediadores e acessam o serviço diretamente através das interfaces expostas.

Uma importante vantagem de uma arquitetura orientada a serviço é que ela permite o desenvolvimento de aplicações que podem ser distribuídas e acessadas através da rede.

2.2 Definição

A tecnologia Web Service pode ser definida de várias maneiras, os conceitos vão desde os mais simples até os mais técnicos, conforme pode ser verificado a seguir:

Segundo Singh et al. (2004), um Web Service é um software aplicativo, acessível na Web (ou na Intranet de uma empresa) através de uma URL, o qual é acessado por clientes através de protocolos baseados em XML, tal como SOAP, que são enviados sobre protocolos padrões da Internet, tal como HTTP. Os clientes acessam uma aplicação Web Service através de suas interfaces e conexões, os quais são definidos usando artefatos XML, tal como arquivos WSDL.

O W3C, consórcio responsável pela criação dos padrões da Web, define Web service como sendo um sistema de software identificado por uma URI, no qual conexões e interfaces públicas são definidas e descritas em XML. Suas definições podem ser descobertas por outros sistemas de software, os quais poderão interagir com o Web Service com base nas informações descritas na definição deste último, utilizando mensagens baseadas em XML e transportadas por protocolos da internet (W3C, 2006).

Em outras palavras, um Web service pode ser definido como uma peça da lógica do negócio capaz de integrar sistemas e possibilitar a comunicação entre diferentes aplicações, possibilitando desta forma a interação de novas aplicações com aplicações já existentes e a compatibilidade de sistemas desenvolvidos em plataformas diferentes. A interoperabilidade provida pelo Web Service é garantida devido ao fato desta tecnologia permitir que as aplicações enviem e recebam dados no formato XML. Deste modo, cada

aplicação pode ser implementada em qualquer linguagem de programação, que é traduzida para uma linguagem universal, o formato XML.

2.3 Histórico

A Microsoft e a IBM foram as precursoras no desenvolvimento dos Web Services, objetivando prover a comunicação entre aplicativos de software, independentemente das suas plataformas e linguagens de desenvolvimento, a fim de que eles pudessem integrar seus serviços.

A criação da linguagem XML, a qual consiste em um padrão para descrição de dados independente de plataforma, foi o ponto inicial na viabilização da criação dos Web Services. O passo seguinte foi a especificação e padronização de um protocolo de envio de mensagens baseado em XML, o qual consistiu no desenvolvimento do Protocolo SOAP, que foi iniciado pela Microsoft e posteriormente obteve o apoio e esforços da IBM. Paralelamente, a IBM e a Microsoft trabalharam com o intuito de especificarem uma forma de descrever um Web Service. Após algumas discussões elas resolveram fundir suas propostas de protocolos, *Network Accessible Service Specification Language* (proposto pela IBM), *Service Description Language* e SOAP Contract Language (propostas pela Microsoft), criando desse modo a especificação WSDL (*Web Service Description Language*).

Com o desenvolvimento do SOAP e do WSDL as empresas já podiam criar e descrever seus Web Services. No entanto ainda era necessário o desenvolvimento de uma solução que fornecesse meios de se descobrir a disponibilidade de um Web service. Foi nesse contexto, que surgiu, em setembro de 2000, o UDDI (*Universal Description, Discovery and Integration*). Com o estabelecimento do SOAP, WSDL e UDDI, padrões usados na criação

de Web Services, no fim de 2000 empresas como Oracle, HP, Sun, IBM, BEA e Microsoft declaram a sua intenção de suportar e implantar (*deploy*) os padrões Web Services em seus produtos. Os padrões SOAP, WSDL e UDDI, serão abordados com mais detalhes nas seções 2.8, 2.9 e 2.10, respectivamente, deste trabalho.

2.4 Vantagens

Os benefícios fornecidos pela utilização da tecnologia Web Service são os fatores chaves para a popularidade e aceitação dos mesmos. Segundo Singh et al. (2004), dentre estes benefícios fornecidos pelos Web services, os principais são:

- Interoperabilidade em ambientes heterogêneos – consiste no fato do modelo Web Service permitir que serviços distribuídos diferentes rodem sobre uma variedade de arquiteturas e plataformas de software, e que eles sejam escritos em diferentes linguagens de programação.
- Negócios através da Web – os Web services podem ser usados pelas empresas para que as vantagens da Web ajudem a alavancar os seus negócios.
- Integração com sistemas existentes – através dos web services é possível integrar sistemas já existentes, tal como a integração de sistemas gerenciadores de banco de dados e monitores de transações com outras aplicações. Esta característica dos web services pode ser bastante útil em sistema empresarial, por exemplo.

- Liberdade de escolha – os padrões Web Service tem aberto um enorme mercado para as ferramentas, produtos e tecnologias. Estes padrões proporcionam às organizações uma ampla variedade de escolha, permitindo a estas selecionar as configurações que mais se adequam aos requerimentos das suas aplicações.
- Suporte a vários tipos de clientes – com a utilização da tecnologia Web Service é possível estender a utilização de serviços e aplicações para um variado tipo de clientes, como por exemplo, para usuários de plataformas Unix e Windows.

2.5 Aplicabilidade

Segundo Leandro (2005), a tecnologia Web Service tem sido empregada de forma bem variada em diversos campos da computação, tais como:

- Como interface para sistemas legados e sistemas *desktop*, permitindo disponibilizar os processos de negócios de uma empresa para seus parceiros e clientes através da internet sem ter que criar uma nova infra-estrutura de aplicações, bastando para isso apenas a criação de interfaces de comunicação de forma a disponibilizar as aplicações já existentes como Web Services;
- Na integração de *Data Warehouses* e na comunicação entre computadores em grades (*Grids)*;
- Na comercialização de software, na qual uma empresa pode disponibilizar suas aplicações na Web, através da tecnologia Web Service, para que seus clientes usufruam dos seus serviços. A fim de exemplificarmos, suponhamos que uma empresa tenha

desenvolvido um aplicativo que provê o serviço de análise de crédito; no caso de uma pessoa ou empresa necessitar deste serviço ao invés de desenvolver uma solução, ela pode simplesmente acessar o Web service para obter essa funcionalidade.

- Na combinação de serviços mais simples para a implementação de um serviço mais complexo, utilizando os serviços como componentes de software, os quais auxiliam no desenvolvimento de aplicações distribuídas. Podemos citar como exemplo uma empresa de turismo que vende pacotes de férias. Para atender as necessidades dos seus clientes a empresa deve oferecer serviços de reserva de hotéis, aluguel de carro, reserva de vôo, etc. No entanto, para oferecer estes serviços a empresa de turismo pode combinar os serviços oferecidos por hotéis, locadoras de carros e empresas aéreas, usando a tecnologia Web Services.

2.6 Arquitetura Web Services

A tecnologia Web Services consiste basicamente na implementação da Arquitetura Orientada a Serviço (SOA), abordada no item 2.1 deste trabalho. Como citado anteriormente, o modelo SOA, e conseqüentemente os Web Services, baseiam-se na interação de três agentes de software: o provedor do serviço, o mediador (*broker*) do serviço e o consumidor do serviço. O objetivo principal desta interação é fazer com que um consumidor consiga buscar e utilizar um serviço fornecido por um provedor através de um mediador (*broker*).

No item 2.1 deste trabalho vimos que para que às interações entre os agentes de software possam ocorrer, o sistema SOA, neste caso os Web

Services, deve prover em sua arquitetura os seguintes componentes: de transporte, de descrição e de descobrimento. E é exatamente isto que os Web Services fazem através dos padrões SOAP, WSDL e UDDI.

Segundo Singh et al. (2004), padrões consistem em coleções de especificações, regras e orientações formuladas e aceitas pelas principais empresas do mercado sem a especificação dos detalhes de implementação. O grande sucesso dos Web Services está na adoção de padrões, os quais permitem que um serviço seja requisitado por qualquer cliente de forma independente de plataforma e de linguagem de programação, e do mesmo modo que permite que um cliente localize e utilize um serviço sem precisar se preocupar com os detalhes de implementação deste serviço.

Analisando a *Figura 2*, a qual ilustra a arquitetura Web Service, pode-se observar que os componentes de transporte, descrição e descobrimento, citados anteriormente, são implementados respectivamente pelos padrões SOAP, WSDL, e UDDI. O padrão WSDL é utilizado na **descrição** das interfaces dos serviços web fornecidos pelo provedor de serviço. Através do padrão UDDI é possível realizar tanto a publicação da descrição das interfaces para o mediador (*broker*) de serviços, quanto a **descoberta** dos serviços requisitados pelos consumidores. O padrão UDDI é o responsável também pela especificação do local no qual o serviço requerido pode ser encontrado bem como qual deve ser o formato das mensagens trocadas. Já o padrão SOAP é utilizado no envio e recebimento, ou seja, **transporte** das mensagens trocadas entre o consumidor, mediador e o provedor de serviços.

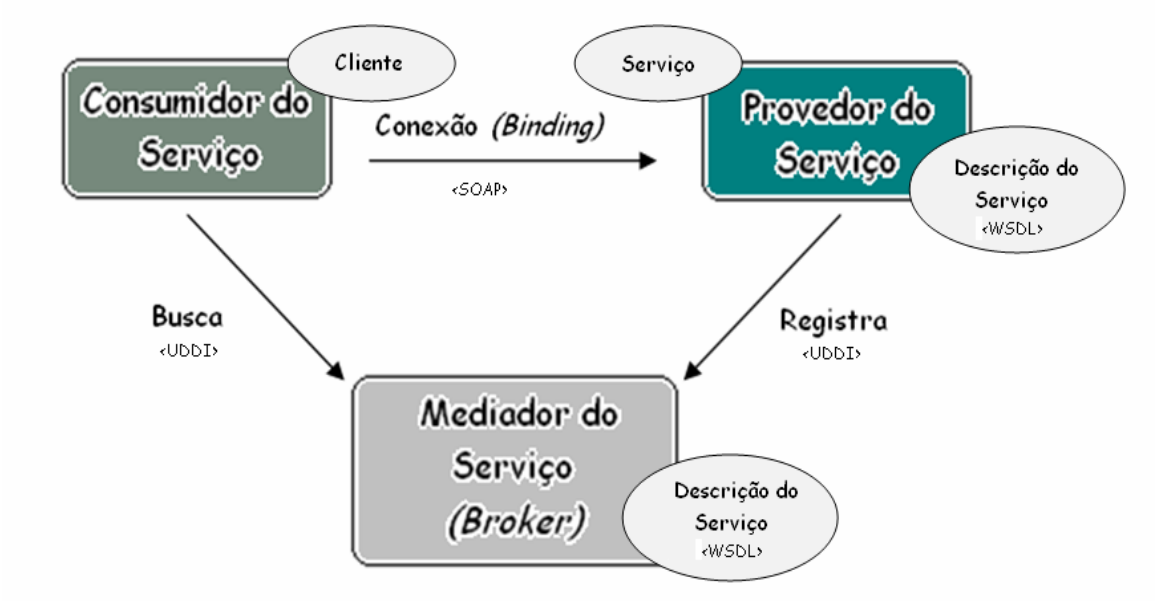

Figura 2 – Arquitetura do Web Service.

2.7 XML – Extensible Mark-up Language

A XML (*Extensible Mark-up Language*) consiste em uma linguagem extensível de marcação de dados baseada em texto, definida e recomendada pela W3C. (BRAY et al., 2006). Segundo Silberschatz, Korth e Sudarshan (2006) o termo marcação refere-se a qualquer elemento em um documento que não sirva como parte da saída impressa. Uma funcionalidade importante da marcação é que ela pode ajudar a registrar o que cada parte do texto representa semanticamente e, ao mesmo tempo, ajuda a automatizar a extração das principais partes dos documentos. No processamento eletrônico de documentos, uma linguagem de marcação, tal como XML, pode ser utilizada para descrever formalmente qual parte do documento é conteúdo, qual parte é marcação e o que significa a marcação.

A marcação na linguagem XML, assim como nas linguagens HTML e SGML, é realizada através da utilização de *tags* (marcadores) que são delimitadas pelos sinais < e >. As *tags* são usadas aos pares, com <tag> e

</tag> delimitando o inicio e o final da parte do documento à qual a *tag* se refere. O que difere as linguagens HTML e XML é o fato de, ao contrário da linguagem HTML, a XML não delimitar o conjunto de *tags* permitidas, sendo que o conjunto de *tags* pode ser escolhido conforme a necessidade de cada aplicação. É esta flexibilidade da linguagem XML que permite que ela seja utilizada na representação e troca de dados, enquanto HTML é utilizada principalmente na formatação de documentos.

XML é um padrão muito bem aceito pela indústria, pois ela viabiliza a comunicação, ou seja, a troca de dados entre aplicações de forma independente de plataforma ou de tecnologia e suas mensagens podem ser enviadas pela internet através do protocolo HTTP. Segundo Vaughan-Nichols (2002) XML é o padrão Web Service mais importante e é também a base dos demais padrões WS.

Segundo Silberschatz, Korth e Sudarshan (2006), a linguagem XML apresenta as seguintes vantagens:

- Auto-descritiva – a presença das *tags* facilita o entendimento das mensagens, tornando-as auto-explicáveis, ou seja, não é necessário consultar um esquema para se entender o significado do texto;
- Flexível – o formato do documento XML não segue uma estrutura rígida, permitindo, por exemplo, que *tags* que especificam atributos de um elemento apareçam em algumas partes do documento e em outras não. Como por exemplo, a *tag* <*cor*> do exemplo da *Figura 3*, a qual é utilizada no elemento *item* com identificador *0456* e não é utilizada no elemento *item* com identificador *0123.* Desse modo, a

XML permite a representação tanto de dados estruturados quanto de dados semi-estruturados.

- Padrão aberto – por ser independente de plataformas e de sistemas operacionais XML pode ser utilizado na integração de sistemas heterogêneos.
- Extensível - A possibilidade de criação de *tags* de um modo arbitrário permite que a estrutura de um documento XML seja adaptada a praticamente qualquer domínio de problema.

2.7.1 Estrutura de Dados XML

Um documento XML é formado basicamente por elementos, valores dos dados e opcionalmente por atributos das *tags*. Um elemento é composto por um par de *tags* que delimitam o seu inicio e fim e pelo texto compreendido entre este par de *tags*. Segundo Silberschatz, Korth e Sudarshan (2006), em um documento XML é necessário que se tenha um único elemento raiz, o qual compreende todos os outros elementos no documento. No exemplo da *Figura 3*, o elemento <*ordem_compra*> forma o elemento Raiz. Além disso, os elementos precisam ser aninhados corretamente. Conforme Silberschatz, Korth e Sudarshan (2006) o texto aparece no contexto de um elemento se ele aparecer entre a *tag* de inicio e *tag* de fim desse elemento. As *tags* são aninhadas corretamente se cada *tag* de inicio tiver uma única *tag* de fim correspondente que esteja no contexto do mesmo elemento pai.

Exemplo de aninhamento correto:

<item> .... <id>... </id> .... </item>

Exemplo de aninhamento errado:

<item> .... <id>... </item> .... </id>

Os valores dos dados são especificados entre as *tags* de inicio e fim de um elemento, como por exemplo, o texto Priscilla Francielle especificado entre o elemento <fornecedor> no exemplo da *Figura 3*.

Os atributos em XML são usados na descrição dos elementos ou no fornecimento de uma informação adicional sobre os mesmos. Eles aparecem como pares *nome=valor*, antes do sinal de fechamento de uma *tag* (>). Um exemplo de atributo pode ser observado na *Figura 3* no elemento *<preço>*, no qual é especificado o atributo *moeda* que fornece informação adicional ao elemento *<preço>*.

Um elemento que não contém subelementos ou texto pode ser abreviado como *<elemento/>* ao invés de assumir a forma *<elemento> </elemento>*.

```
<?xml version="1.0"?>
<ordem_compra>
    <id> OrdemCompra_1</id>
    <comprador>
        <nome> Valmor Ataide Machado Junior </nome>
        <endereço> Rua: João Manoel, 1569</endereço>
    </comprador>
    <fornecedor>
        <nome> Priscilla Francielle </nome>
        <endereço> Rua: Lauro Linhares, 426</endereço>
    </fornecedor>
    <lista-itens>
        <item>
            <id> 0123 </id>
            <descrição>Notebook 15" Pentium 1.7M 512MB HD40 DVD-RW </descrição>
            <quantidade> 1 </quantidade>
            <preço moeda=real> 2.699,00 <</preço>
        </item>
        <item>
            <id> 0456 </id>
            <descrição> Monitor LCD 15" 540N - Prata e Preto</descrição>
            <quantidade> 1 </quantidade>
            <cor> Prata </cor>
            <preço moeda=real> 599,99 </preço>
        </item>
    </lista-itens>
    <custo_total moeda=real> 3.298,99</custo_total>
</ordem_compra>
```

Figura 3 – Exemplo de Documento XML.

2.7.2 Esquema do Documento XML

Documentos XML podem ser criados sem que se tenha que seguir qualquer estrutura previamente definida, ou seja, qualquer elemento pode ter qualquer subelemento ou atributo. Esta característica pode não ser muito útil quando um documento XML tiver que ser processado automaticamente como parte de uma aplicação. Para solucionar este problema, de validação de documentos XML, foram criados os esquemas DTD e XSD.

Document Type Definition – DTD

Segundo Silberschatz, Korth e Sudarshan (2006), a finalidade principal uma DTD consiste em restringir as informações e os tipos de informações presentes no documento XML, ou seja, ela descreve a estrutura de um documento XML. A DTD não restringe os tipos no sentido dos valores de dados que eles podem assumir como inteiro, cadeia de caracteres ou real; ela só restringe o surgimento de subelementos e atributos dentro de um elemento.

Uma DTD contém as informações sobre as *tags* que o documento XML, correspondente ao DTD, pode conter, bem como as regras que indicam o padrão de subelementos que podem aparecer dentro de um elemento. Desse modo, aplicações que utilizam XML para intercambiar dados podem validar os documentos recebidos através do DTD antes de processá-los.

XML Schema Definition - XSD

Surgiu para reparar as deficiências apresentadas pelo esquema DTD. Conforme Silberschatz, Korth e Sudarshan (2006), a XSD, também conhecida como XML Schema, define uma série de tipos internos, como *string*, *integer*,

decimal, *date* e *boolean*. Além disso, permite o uso de tipos definidos pelo usuário, que podem ser simples, com restrições adicionais, ou tipos complexos.

Silberschatz, Korth e Sudarshan (2006) destaca os seguintes benefícios da XSD em relação à DTD, dentre outros:

- Permite restrição a tipos específicos, como tipos numéricos, etc.
- Permite a criação de tipos definidos pelos usuários.
- Permite restrições de exclusividade e de chave estrangeira.

2.8 SOAP

Inicialmente, SOAP era usado como acrônimo de *Simple Object Access Protocol*. No entanto, como o protocolo perdeu a sua simplicidade original e passou a trabalhar com várias formas de estrutura de dados (não só com objetos), os fornecedores da última versão do padrão decidiram por referenciá-lo apenas como SOAP - deixando de ser considerado um acrônimo.

O W3C define o padrão SOAP versão 1.2 (SOAP) como “um protocolo leve cujo objetivo consiste em prover uma infra-estrutura que permita a troca de informação em ambientes distribuídos, descentralizados”. (MITRA, 2003). A especificação SOAP define um *framework* de transferência de mensagens para a troca de dados no formato XML sobre a internet. Este *framework* de transferência de mensagens é completamente independente de sistema operacional, linguagem de programação, ou plataforma de computação distribuída.

Em outras palavras, SOAP fornece um mecanismo de comunicação pelo qual um componente de software pode invocar as habilidades de outro através da transmissão de mensagens XML pela Web. Em termos práticos, SOAP determina como deve ser feita a construção do documento XML que representa

a mensagem que carrega a requisição e a resposta da arquitetura Web Service.

De acordo com Muschamp (2004), o protocolo SOAP engloba o seguinte:

- Um envelope que define um *framework* para escrever o que há na mensagem e como processá-la.
- Um conjunto e regras de codificação para enviar instâncias de tipos de dados dentro de uma mensagem.
- Uma convenção para representar a maneira pela qual os procedimentos (ou métodos) dos componentes de software podem ser chamados, e suas respostas.
- Uma conexão (*binding*) para trocar mensagens usando um protocolo de comunicação.

SOAP baseia-se no mecanismo de *request-response* (requisição-resposta), no qual um componente de software faz uma requisição para outro componente de software que pode ou não fornecer a resposta. Segundo Newcomer (2002), SOAP é um tipo de extensão do HTTP que suporta transferência de mensagens XML, sendo que o envio da mensagem XML pelo protocolo SOAP é feito via um HTTP *request* e o recebimento de uma resposta, se houver, é feita via HTTP *response*.

Conforme Singh et al. (2004), a fim de possibilitar a troca de mensagens, o protocolo SOAP define um **envelope**, estrutura composta por dois elementos: o corpo (*body*) SOAP, dentro do qual as mensagens que serão enviadas ao destinatário são incluídas, e opcionalmente o cabeçalho (*header*) SOAP. Este envelope, o qual representa uma mensagem SOAP, consiste em um

documento XML formado pelo elemento raiz *envelope*, que por sua vez contém o elemento *header* (opcional) e o elemento *body* (obrigatório).

De acordo com Leandro (2005), um elemento *header* possibilita a adição de diferentes funcionalidades ao protocolo SOAP. Cada subelemento do elemento *header* é chamado de *header block* (bloco de cabeçalho). Estes blocos de cabeçalho não são definidos pelo protocolo SOAP, no entanto outras especificações, tais como de segurança e de transações, os definem com informações relacionadas às suas funcionalidades. Atributos são definidos pelo protocolo SOAP a fim de indicar quem deve tratar um bloco de cabeçalho e se o processamento do bloco é obrigatório ou opcional.

O modelo de processamento SOAP baseia-se na interação entre nós SOAP (SOAP *node*), os quais se referem aos componentes de software que enviam e recebem mensagens SOAP. Neste modelo, cada nó participante da interação recebe uma denominação especifica. O nó que inicia a transmissão da mensagem é denominado *original sender*. O nó de destino da mensagem, o qual processa a mensagem, é denominado *ultimate sender*. Todos os nós que interagem com a mensagem entre os nós *original sender* e *ultimate sender* são denominados nós intermediários. O conjunto de nós pelos quais uma mensagem passa é denominado caminho da mensagem (*message path*).

Segundo Leandro (2005), a especificação SOAP define dois papéis: (*roles*), *next e ultimateReceiver*. Estes são associados aos nós SOAP do caminho da mensagem com o intuito de identificá-los. O papel *ultimateReceiver* é associado somente ao nó destino da mensagem enquanto os demais nós são associados ao papel *next*. Cada papel SOAP é uma categoria que associa um nome URI a uma funcionalidade abstrata (*caching*, validação, autorização,etc.).

O processamento do corpo da mensagem SOAP é sempre destinado ao *ultimateReceiver*, no entanto o cabeçalho da mensagem pode ser destinado tanto aos nós intermediários (*next*) como ao nó destino (*ultimateReceiver)*. Com o intuito de identificar qual nó deve processar determinado bloco de cabeçalho e como deve ser feito este processamento, o protocolo SOAP definiu três atributos para o elemento *header block*: *role, relay* e *mustUnderstand.*

A função do atributo *role* é indicar a qual nó o bloco de cabeçalho é endereçado. Já o atributo *mustUnderstand* indica que o cabeçalho não pode ser ignorado e portanto deve ser encaminhado de um nó para outro até chegar ao seu nó destino, ou seja, ao nó identificado pelo atributo *role*. Por fim, o atributo *relay* indica se um bloco de cabeçalho deve ser encaminhado ou descartado pelo nó.

A *Figura 4* apresenta um exemplo de mensagem SOAP. As primeiras 4 linhas do exemplo referem-se ao encapsulamento HTTP, uma vez que está mensagem SOAP foi enviada via HTTP. Nesta mensagem SOAP podemos identificar os dois elementos que compõem um envelope SOAP: o cabeçalho (*header*) e o corpo (*body*). Ao analisarmos a mensagem podemos verificar que o elemento *header* é composto por um bloco de cabeçalho *priority*, o qual tem por finalidade indicar a prioridade da mensagem SOAP.

No conteúdo do elemento *body* está definida a chamado a um método *translate,* ao qual são passados 3 parâmetros. O elemento *text* fornece o texto que deverá ser traduzido. O elemento *from* indica a língua na qual o texto a ser traduzido foi escrito e o elemento *to* indica para qual língua o texto deve ser traduzido.

```
POST /axis/services/WSATranslator HTTP/1.0
Content-Type: text/xml; charset="utf-8"
Content-Length: xxxx
SOAPAction: "urn:translate"
<SOAP:Envelope  xmlns:SOAP="http://schemas.xmlsoap.org/soap/envelope/"
                xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                SOAP:encodingStyle="http://www.w3.org/2003/05/soap-encoding"
                xmlns:ns1="http://translator.example/translator">
    <soapenv:Header>
        <soapenv:priority
            soapenv:mustUnderstand="0" xsi:type="xsd:string">high
        </soapenv:priority>
    </soapenv:Header>
    <SOAP:Body>
        <ns1:translate>
            <ns1:text xsi:type="xsd:string">hello</ns1:text>
            <ns1:from xsi:type="xsd:string">en</ns1:from>
            <ns1:to xsi:type="xsd:string">pt</ns1:to>
        </ns1:translate>
    </SOAP:Body>
</SOAP:Envelope>
```

Figura 4 – Exemplo de mensagem SOAP enviada via http. (LEANDRO, 2005).

2.9 WSDL

Web Services Descripition Language (*WSDL*) é a linguagem que define uma forma padrão para que os detalhes de um web service sejam especificados (CHRISTENSENT et al., 2001). Esta linguagem pode ser utilizada para descrever a interface de um Web Service, para especificar as conexões (*bindings*) realizadas entre os componentes de software e para especificar outros detalhes de desenvolvimento dos web services. Segundo Muschamp (2004), um arquivo WSDL consiste em um documento no formato XML que fornece as seguintes informações referentes ao provedor de serviço:

- Informações sobre todos os métodos disponíveis, incluindo seus parâmetros de chamada.
- Informações a respeito dos tipos de dados das mensagens XML, incluindo a especificação de valores.

- Informações sobre as conexões (*bindings*) e sobre o protocolo de transporte a ser utilizado.
- Endereço onde encontrar o serviço especificado.

De acordo com Leandro (2005), o WSDL é composto por três partes: por uma interface abstrata, a qual descreve os serviços que serão oferecidos pelo Web Service; por informações dependentes de protocolo, que devem ser usadas no acesso ao serviço juntamente com a especificação do protocolo de comunicação utilizado; e pela localização do serviço. O padrão WSDL possibilita que cada uma destas partes sejam especificadas em documentos WSDL diferentes a fim de prover maior flexibilidade e reusabilidade. No caso de as partes serem especificadas em documentos diferentes, se faz necessária a importação de cada uma das partes.

A primeira parte do documento WSDL consiste na descrição do que o serviço provê. Esta parte é composta pelo conjunto de operações que o serviço realiza, sendo que para cada operação são definidas as mensagens de entrada e saída, o formato de cada mensagem e o tipo de dados de cada elemento da mensagem (LEANDRO, 2005). Em termos de elementos que compõem o documento WSDL, a primeira parte engloba o seguinte: o elemento <*types*>, o qual define os tipos de dados contidos nas mensagens transferidas como parte do serviço; o elemento <*message*>, o qual especifica as mensagens que são aceitas pelo Web Service; e o elemento <*portType*>, o qual define de forma abstrata o conjunto de operações que são oferecidas pelo Web service.

Conforme Leandro (2005), a segunda parte do documento WSDL define a conexão (*binding*) da interface abstrata, definida na primeira parte, a um conjunto concreto de protocolos. O elemento <*binding*> do documento WSDL

implementa está parte. Ele é o responsável por especificar o estilo de codificação da mensagem, a necessidade ou não de um esquema XML, o protocolo XML usado para na construção do envelope, os cabeçalhos que devem ser incluídos na mensagem e o protocolo de transferência que deve ser usado.

A terceira e última parte do arquivo WSDL consiste na descrição de um serviço. Esta parte é implementada pelo elemento <*service*> do arquivo WSDL, o qual é composto por um ou mais subelementos <*port*>. Cada elemento <*port*> implementa uma conexão (*binding*) específica de uma interface abstrata, indicando o ponto de acesso a um *endpoint* de um serviço. Um provedor de serviço pode oferecer vários pontos de acesso para o mesmo serviço, cada um implementando uma conexão (*binding*) diferente. (LEANDRO, 2005).

A *Figura 5* apresenta um exemplo de documento WSDL especificado para um serviço meteorológico. Neste exemplo os elementos <*message*> definem as mensagens usadas como parâmetros de entrada (o nome da cidade) e de saída (o valor da temperatura). O elemento <*portType*> define as operações oferecidas pelo serviço de meteorologia, neste caso a operação *getWeather.* O elemento <*binding*> especifica as regras de codificação para as entradas e saídas do *port type*, conectando-o a operação. O elemento <*service*> especifica o serviço e fornece o endereço de localização do serviço através do subelemento <*port*>.

```
<?xml version="1.0" encoding="UTF-8"?>
    <definitions name="WeatherService"
            targetNamespace="http://www.townweather.com/wsdl/WeatherService.wsdl"
            xmlns="http://schemas.xmlsoap.org/wsdl/"
            xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
            xmlns:tns="http://www.townweather.com/wsdl/WeatherService.wsdl"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema">
        <message name="getWeatherRequest">
            <part name="town" type="xsd:string"/>
        </message>
        <message name="getWeatherResponse">
            <part name="temperature" type="xsd:int"/>
        </message>
        <portType name="Weather_PortType">
            <operation name="getWeather">
                <input message="tns:getWeatherRequest"/>
                <output message="tns:getWeatherResponse"/>
            </operation>
        </portType>
        <binding name="Weather_Binding" type="tns:Weather_PortType">
            <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
            <operation name="getWeather"> <soap:operation soapAction=""/>
                <input> <soap:body
                        encodingStyle="http://schemas.xmlsoap.org/soap.encoding/"
                        namespace="urn:examples:weatherservice" use="encoded"/>
                </input>
                <output> <soap:body
                        encodingStyle="http://schemas.xmlsoap.org/soap.encoding/"
                        namespace="urn:examples:weatherservice" use="encoded"/>
                </output>
            </operation>
        </binding>
        <service name="WeatherService">
            <documentation>WSDL File for Weather Service</documentation>
        <port binding="tns:Weather_Binding" name="Weather_Port">
            <soap:address location="http://localhost:8080/soap.servlet.rpcrouter"/>
        </port>
        </service>
</definitions>
```

Figura 5 – Exemplo de documento WSDL. (MUSCHAMP, 2004).

2.10 UDDI

A especificação *Universal Description, Discovery and Integration (UDDI)* fornece um mecanismo de registro, remoção e pesquisa de Web Services (CLEMENT et al., 2004). UDDI fornece um repositório centralizado, o qual pode ser de acesso público ou privado, no qual os Web Services são disponibilizados para consulta de usuários que procuram por um serviço específico.

De acordo com Leandro (2005), o protocolo UDDI define um Web Service para registro de serviços, acessado através de mensagens SOAP, o qual gerencia as informações referentes aos tipos, implementações e provedores dos serviços que ele registra. Através deste mecanismo, definido pelo UDDI, os provedores de serviços podem disponibilizar os serviços que oferecem e os consumidores podem pesquisar os serviços desejados e obter os metadados necessários para utilizá-los.

Conforme Singh et al. (2004), para que um provedor de serviço possa registrar um Web Service através do protocolo UDDI, ele precisa fornecer as seguintes informações: a definição do seu negócio, a definição dos seus serviços, as conexões (*bindings*) e as informações técnicas sobre o serviço.

Estas informações são descritas como entidades, as quais são divididas em três categorias:

- Páginas brancas (*white pages*) – fornecem informações sobre o provedor do serviço, tais como: nome, endereço, web site da empresa, número de telefone, etc.
- Páginas amarelas (*yellow pages*) - classificam os provedores em categorias de acordo como seu ramo de negócio.

- Páginas verdes (*green pages*) – fornecem uma lista dos serviços, das conexões (*bindings*) e informações técnicas sobre os serviços que a empresa oferece. No caso de haver um ponteiro para o arquivo WSDL, ele pode ser disponibilizado nesta categoria.

Segundo Leandro (2005), a estrutura dos dados UDDI possui quatro elementos: <*businessEntity*>, <*businessService*>, <*bindingTemplate*> e <*tModel*>.

O elemento <*businessEntity*> descreve o negócio ou outras entidades para os quais as informações foram registradas. O elemento <*businessService*>, subelemento do <*businessEntity*>*, f*ornece o nome, identificador e a descrição do serviço publicado. O elemento <*bindingTemplate*> fornece informações sobre o serviço, incluindo o endereço de acesso ao serviço.

De acordo com Newcomer (2002), o elemento <*tModel*> consiste em um mecanismo utilizado para trocar metadados sobre um Web Service, assim como a descrição de um Web Service, ou um ponteiro para um arquivo WSDL.

A especificação UDDI fornece duas categorias de API's de acesso aos serviços UDDI através de aplicações:

- Localização de serviços (*Inquiry API*): permite a pesquisa e acesso às informações os negócios e serviços registraddos.
- Publicação de serviços (*Publish API*): permite que aplicações publiquem, atualizem e apaguem os registros dos seus serviços através do *registry.*

A *Figura 6* exemplifica a estrutura UDDI de um Web Service. Analisando o exemplo, é possível identificar claramente todos os elementos que compõem uma especificação UDDI, conforme listado anteriormente. Este exemplo ilustra um registro UDDI para uma empresa que fornece o serviço de tradução. Através do elemento <*businessEntity*> conseguimos obter informações referentes ao provedor do serviço, neste caso a *Babelfish*. O elemento

<*businessServices*> descreve os serviços oferecidos pela *Babelfish*. Neste caso podemos verificar que a empresa só oferece o serviço de tradução. No caso do provedor de serviço oferecer mais serviços, teremos mais e uma ocorrência do elemento <*businessService*>.

O endereço de acesso ao serviço fornecido pela *Babelfisch* é especificado pelo elemento <*bindingTemplate*>, mais especificamente pelo seu subelemento <*accessPoint*>.

```
sinessEntity businessKey="ba744ed0-3aaf-11d5-80dc-002035229c64">
  <name>Translator</name>
  <description>The Babel fish is probably the oddest thing in the Universe...</description>
  <contacts>
      <contact useType="Geral">
          <personName>Babelfish</personName>
          <phone>00 55 42 42 42</phone>
          <email>contact@hg.co.uk</email>
          <address>
              <addressLine>Rua 10, 120</addressLine>
              <addressLine>Florianópolis, SC</addressLine>
          </address>
      </contact>
  </contacts>
  <businessServices>
      <businessService serviceKey="d5921160-3e16-11d5-98bf-002035229c64"
          businessKey="ba744ed0-3aaf-11d5-80dc-002035229c64">
          <name>Translate</name>
          <bindingTemplates>
          <bindingTemplate bindingKey="d594a970-3e16-11d5-98bf-002035229c64"
               serviceKey="d5921160-3e16-11d5-98bf-002035229c64">
          <description>SOAP binding to translate a text</description>
          <accessPoint URLType="http">
  http://translator.example/translator</accessPoint>
          <tModelInstanceDetails>
          <tModelInstanceInfo
                tModelKey="uuid:0e727db0-3e14-11d5-98bf-002035229c64" />
           </tModelInstanceDetails>
           </bindingTemplate>
           </bindingTemplates>
      </businessService>
  </businessServices>
usinessEntity>
```

Figura 6 – Exemplo da Estrutura UDDI. (LEANDRO, 2005)..

2.11 Considerações Finais

Com base nas características e vantagens da tecnologia Web Service, apresentadas neste capítulo podemos identificar o grande potencial que esta tecnologia apresenta em se tornar o futuro padrão de mercado para integração de sistemas. No entanto, para que isto se concretize de fato, muitas pesquisas

e especificações estão sendo realizadas por órgãos como o W3C (*World Wide Web Consortium*) e o OASIS (*Organization for the Advancement of Structured Information Standards*), para suprir algumas deficiências desta tecnologia, como por exemplo, questões de segurança, disponibilidade e confiabilidade.

De acordo com Elsmari e Navathe (2000), para que um sistema seja considerado confiável ele precisa garantir a consistência e integridade dos seus dados, e é neste contexto que se insere o conceito de transação, assunto este que é abordado no próximo capítulo deste trabalho.

3 Transações

Eventualmente, operações que efetuam alterações sobre um conjunto de dados inter-relacionados são executadas por sistemas computacionais. Porém, é comum que uma ou mais partes desta operação não sejam concluídas de maneira correta, ou do modo esperado. Neste caso, torna-se indesejável que estas operações se concretizem de fato, alterando dados que possam ser utilizados por outras aplicações. Seria inaceitável, por exemplo, se em uma operação de transferência bancária de uma conta-corrente para uma conta poupança, a conta-corrente fosse debitada e a poupança não fosse creditada. A situação se agrava ainda mais quando vários usuários executam operações simultâneas sobre os mesmos dados, pois é possível que estes dados estejam inconsistentes durante a execução de uma determinada operação, levando outras operações a resultados errôneos.

Segundo Silberschatz, Korth e Sudarshan (2006), um conjunto de operações que formam uma única unidade lógica de trabalho é chamado de transação. Para que um sistema seja considerado confiável é necessário que ele garanta a execução apropriada de transações, apesar da possibilidade de falhas. Deste modo, evita-se que os dados sejam corrompidos ou tornem-se inconsistentes, garantindo ao usuário uma resposta correta, ou ao menos uma mensagem de indicação de erro do sistema.

3.1 Conceitos de Transações

O conceito de transação surgiu da necessidade de implementação de confiabilidade em sistemas computacionais. Sua idéia é a mesma que rege um contrato judicial, no qual várias partes acordam um resultado, sendo que se

alguma das partes não cumprir sua parte no acordo ele é invalidado. (BELLI, 2005).

De acordo com Belli (2005), no mundo computacional uma transação consiste em uma unidade lógica de trabalho, geralmente envolvendo uma seqüência de operações que agem sobre recursos, possivelmente alterando-os. Neste caso, se alguma destas operações falharem, todos os recursos envolvidos na transação devem voltar ao seu estado original, assegurando deste modo que uma transação mal-sucedida não prejudique o funcionamento do sistema.

No exemplo da transferência bancária de uma conta-corrente para uma conta-poupança, conforme citado anteriormente, temos uma transação com no mínimo duas operações: uma para poder retirar o dinheiro da conta-corrente, e outra para depositá-lo na conta-poupança. Neste caso, o sistema deve garantir que ambas as operações serão realizadas com sucesso, caso contrário nenhuma delas deve ser realizada. No entanto, para que o sistema possa tomar conhecimento de que ambas as operações fazem parte de uma mesma transação, é necessário que isto seja especificado.

O escopo de uma transação geralmente é definido pelas instruções (ou chamadas de função) ***begin transaction, commit transaction*** *e* ***rollback transaction***. A instrução ***begin transaction*** indica o início de uma transação e é seguida pelas operações que compõem a transação. As instruções ***commit transaction*** e ***rollback transaction*** indicam o final da transação. A instrução ***commit transaction*** é utilizada para a efetivação da transação no caso de todas as operações terem ocorrido com sucesso. Já a instrução ***rollback transaction*** é utilizada para desfazer as operações realizadas pela transação

em questão, no caso de ocorrer alguma falha em uma das operações realizadas.

Quando uma transação não consegue completar a sua execução com sucesso, ela é considerada uma transação **abortada**. Desse modo, quaisquer alterações realizadas por ela no banco de dados devem ser desfeitas. Quando as alterações realizadas por uma transação abortada são desfeitas, dizemos que a transação foi **recuperada** (*rolled back*), sendo responsabilidade do esquema de recuperação gerenciar transações revertidas. (SILBERSCHATZ; KORTH; SUDARSHAN, 2006).

Transações bem sucedidas são chamadas de transações **efetivadas** (*commited*). As alterações no banco de dados realizadas por transações efetivadas precisam persistir mesmo que haja uma falha no sistema. Os efeitos causados por uma transação efetivada somente podem ser desfeitos através de uma transação de compensação, assunto este que será abordado mais adiante.

A fim de determinar mais precisamente o que significa o término bem-sucedido de uma transação, Silberschatz, Korth e Sudarshan (2006) afirmam que uma transação precisa estar em um dos seguintes estados:

- **Ativa** – durante sua execução a transação permanece nesse estado;
- **Parcialmente Confirmada** – a transação entra neste estado quando sua última instrução é executada;
- **Falha** – a transação entra no estado de falha após a descoberta de que a execução normal não pode prosseguir;

- **Abortada** – a transação pode ser considerada abortada depois que ela for revertida e que o banco de dados for restaurado ao seu estado anterior ao início da execução da transação. Após a transação entrar neste estado o sistema pode seguir um dos seguintes caminhos: ele pode **reiniciar** a transação, esta opção é escolhida somente no caso da transação ter sido abortada em decorrência de alguma falha de hardware ou software que não tenha sido criada por causa da lógica interna da transação; ou ele pode **matar** a transação, o que ocorre normalmente quando a transação foi abortada devido a algum erro lógico interno que só pode ser corrigido com a reescrita da aplicação, ou porque a entrada foi defeituosa, ou porque os dados desejados não foram encontrados no banco de dados; e
- **Confirmada** – a transação é confirmada após o término da sua execução bem sucedida.

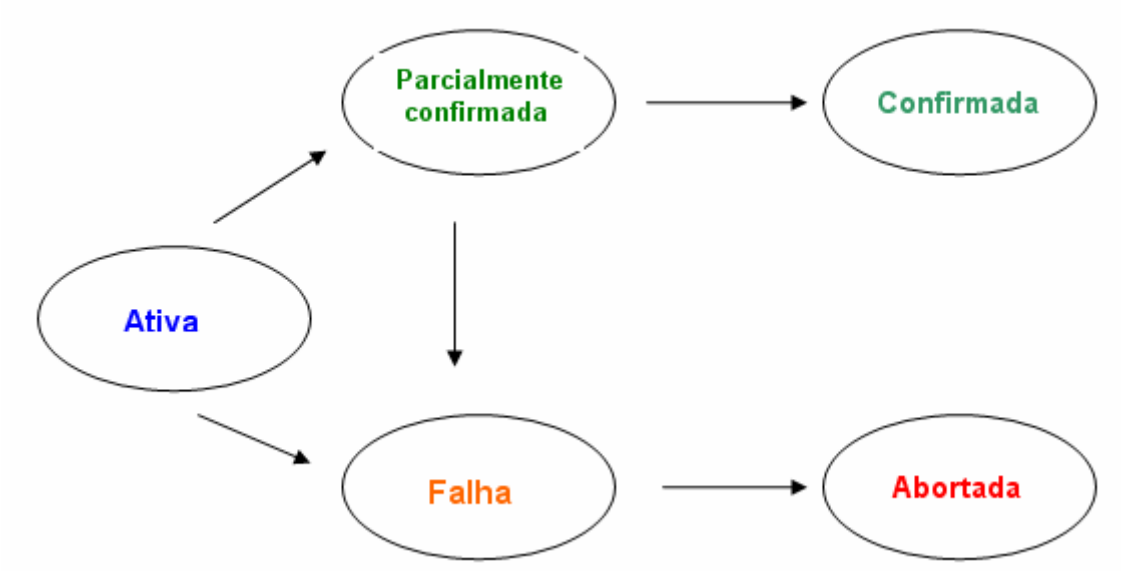

Figura 7 – Diagrama de estados de uma transação. (SILBERSCHATZ; KORTH; SUDARSHAN, 2006).

Uma transação somente é considerada terminada após entrar no estado confirmada ou abortada. O diagrama da *Figura 7* apresenta os estados de uma transação.

3.2 Transações Atômicas e Propriedades ACID

O conceito de transações atômicas surgiu da necessidade de que, mesmo nos casos em que falhas ocorram, a consistência das aplicações deve ser garantida. Uma transação atômica pode ser definida como uma transação indivisível, ou seja, ou ela é totalmente executada ou nenhuma de suas operações são efetivadas.

A transação atômica é considerada uma técnica de tolerância a falhas muito útil, principalmente quando múltiplos recursos transacionais são envolvidos. (CARVALHO, 2000, p. 10, *apud* PHILIP et al., 1987).

De acordo com Silberschatz, Korth e Sudarshan (2006), a fim de que a integridade dos dados seja garantida torna-se necessário garantir as seguintes propriedades:

- **Atomicidade** – nesta propriedade é necessário garantir a atomicidade de uma transação, ou seja, ou todas as operações de uma transação são efetivadas com sucesso ou nenhuma delas é efetivada;
- **Consistência** – uma transação mal-sucedida não deve causar inconsistência no banco de dados. Caso haja uma falha em uma operação da transação, todas as modificações realizadas por esta operação e pelas demais operações da transação em questão devem ser revertidas ao estado original. Ou seja, as transações devem preservar a consistência do banco de dados. Uma maneira

de garantir a consistência do banco de dados é a execução da transação isolada, ou seja, sem qualquer outra transação ocorrendo simultaneamente;

- **Isolamento** – nesta propriedade é necessário garantir que as transações estejam isoladas uma das outras, no sentido de que modificações realizadas por uma transação T1, não se tornem visíveis para qualquer outra transação, até que T1 execute com sucesso uma operação *commit*, ou seja, até que a transação T1 seja finalizada com sucesso. A operação *commit* faz com que as modificações realizadas por uma transação tornem-se visíveis para as outras transações;
- **Durabilidade** – para que uma transação esteja de acordo com esta propriedade é necessário garantir que ela é durável (persistente), ou seja, após uma transação executar uma operação *commit* é necessário que todas as atualizações realizadas por ela sejam aplicadas ao banco de dados, mesmo nos casos de ocorrência de falha do sistema em algum instante. Portanto, alterações efetuadas por uma transação bem-sucedida não podem ser perdidas e devem estar visíveis para transações posteriores.

Essas propriedades são conhecidas como propriedades ACID, acrônimo este derivado da junção da letra inicial de cada propriedade.

3.3 Recuperação de Transações

Segundo Carvalho (2000, *apud* BJORK, 1973, p. 12), a recuperação de uma transação é considerada uma das tarefas mais importantes no gerenciamento de transações. Ela consiste na recuperação do banco de dados

envolvido na transação, ou seja, consiste na restauração do banco de dados para um estado consistente logo após uma falha levá-lo a um estado inconsistente. Desse modo, o mecanismo de recuperação é uma forma de se garantir a propriedade de **consistência**, uma das propriedades ACID conforme visto anteriormente, em uma transação.

Considerando o fato de que uma transação pode e geralmente é composta por mais de uma operação, torna-se necessário que não seja permitido, nestes casos, que uma ou mais operações deixem de serem executadas, ou que sejam executadas com falha. Isto tornaria o banco de dados inconsistente, fato este indesejável para sistemas confiáveis.

Garantir que todas as operações pertencentes a uma transação sejam executadas da forma desejada seria o ideal para garantir a consistência das transações, no entanto isto é algo impossível de ser alcançado. Os sistemas de gerenciamento de transações, com o intuito de suprir essa deficiência, fornecem mecanismos de recuperação, os quais no caso de ocorrência de falhas durante a execução de uma transação, desfazem todas as modificações realizadas pela transação em questão. Dessa forma garante-se a propriedade de atomicidade de uma transação, na qual a transação ou é executada totalmente ou deve ser cancelada integralmente como se nunca tivesse sido realizada. (CARVALHO, 2000).

O componente de sistema responsável por garantir a atomicidade de uma transação é o gerenciador de transações, também conhecido como monitor de processamento de transação (monitor TP). O funcionamento do mecanismo de recuperação baseia-se na utilização das operações *commit* e *rollback*.

Através da operação *commit* o gerenciador da transação toma conhecimento do término bem sucedido de transação, e portanto, deve validar, ou seja, tornar permanente todas as modificações realizadas por esta unidade lógica de trabalho. Já a operação *rollback* indica o término de uma transação mal sucedida, sinalizando ao gerenciador que todas as modificações realizadas por esta transação devem ser desfeitas.

Uma questão importante a ser salientada é que em sistemas reais, no caso de ocorrência de falhas durante a execução de uma transação, além da recuperação do banco de dados, uma mensagem de aviso de erro deve ser enviada ao usuário.

As modificações realizadas por uma transação podem ser desfeitas das seguintes maneiras:

- **Recuperação baseada em *Log*** – o sistema mantém um *log* no qual são registrados detalhes de todas as modificações realizadas por uma transação, bem como a identificação da transação e o dado modificado por ela, além do seu valor antigo e do valor novo. Desse modo, no caso de uma transação mal sucedida, o sistema utiliza as entradas de *log* correspondentes à transação em questão a fim de restaurar o estado do banco de dados antes do início desta transação.
- **Recuperação baseada em *Checkpoint*** – a diferença entre este tipo de recuperação e a baseada em *log* é que neste caso *checkpoints* (marcas) são criados no arquivo de *log* para indicar que todas as operações definidas antes dele já foram efetivadas

em disco. Portanto, em caso de falha, somente as operações definidas depois do último *checkpoint* devem ser refeitas.

- **Recuperação usando Cópia *Shadow* (Sombra)** – esse esquema baseia-se na realização de cópias do banco de dados, chamadas cópia sombra (*shadow*). Um ponteiro, chamado *db-pointer*, é mantido e aponta para a cópia atual do banco de dados. Quando uma transação deseja realizar atualizações no banco de dados, ela cria uma cópia do banco de dados, na qual são realizadas todas as suas atualizações, mantendo o banco de dados original intacto. No caso da transação ser abortada devido a alguma falha, a cópia do banco de dados criada por ela é simplesmente excluída. No entanto, se a transação for concluída com sucesso, procede-se da seguinte maneira: primeiramente o sistema operacional certifica-se de que todas as páginas da cópia do banco de dados foram gravadas em disco; depois o ponteiro *db-pointer* é atualizado para que ele aponte para a cópia sombra, criada e modificada pela transação; e por último a cópia do banco de dados para qual o *db-pointer* apontava é excluída.

3.4 Transações Distribuídas

Em um ambiente de banco de dados distribuídos, uma transação pode acessar dados gravados em mais de um local (*site*), sendo neste caso denominada uma **transação distribuída**. Cada transação é dividida em sub-transações, uma para cada *site* no qual estão armazenados os dados que a transação acessa. (BELL; GRIMSON, 1992).

O nível de complexidade envolvido no gerenciamento de transações distribuídas é muito superior ao gerenciamento de **transações locais**, as quais acessam e atualizam dados apenas em um banco de dados local. Este aumento de complexidade deve-se ao fato de que os gerenciadores de transações distribuídas precisam gerenciar recursos de rede e também de máquinas remotas.

Outra característica particular às transações distribuídas é que o seu desempenho é muito prejudicado se suas operações forem executadas de forma seqüencial nos casos em que as operações não dependem logicamente entre si e são executadas em *sites* diferentes. (JAJODIA; KERSCHBERG, 1997).

Segundo Silberschatz, Korth e Sudarshan (2006), os sistemas de transações distribuídas são compostos por dois componentes:

- **Gerenciador de transações** – cada *site* possui um. Ele é responsável por controlar as transações (ou sub-transações) que acessam os dados localizados naquele *site* local. Dentre as suas responsabilidades estão: manter um *log* para fins de recuperação e participar de um esquema de controle de concorrência a fim de controlar a execução concorrente das transações executadas em um determinado *site*.
- **Coordenador de transação** – coordena a execução de várias transações (locais e globais) iniciadas em um determinado *site*. Suas responsabilidades são: iniciar a execução de uma transação, dividir a transação em várias sub-transações, distribuí-las aos *sites* apropriados para execução e coordenar o término da transação, a

qual poderá ser confirmada em todos os *sites* ou abortada em todos os *sites*.

Além de poder sofrer com os mesmo tipos de falhas que ocorrem em um sistema centralizado (como por exemplo, erros de software, erros de hardware ou falhas de disco), sistemas distribuídos podem sofrer com outros tipos de falhas básicos: falha de um site, perda de mensagens, falha de um enlace de comunicação e partição da rede. (SILBERSCHATZ; KORTH; SUDARSHAN, 2006).

Nesta seção são apresentados mecanismos de gerenciamento de transações peculiares a ambientes distribuídos.

3.4.1 Efetivação de Transações Distribuídas

A efetivação ou aborto de uma transação distribuída torna-se muito mais difícil se comparada à ação equivalente em ambiente centralizado devido à descentralização de um sistema distribuído, no qual podem ocorrer problemas de rede e falhas de máquinas. Garantir a atomicidade de uma transação distribuída pode não ser uma tarefa fácil, uma vez que as transações (sub-transações) devem ser confirmadas ou abortadas em todos os sites envolvidos na transação. Portanto, para garantir essa propriedade, um coordenador de transação precisa de um protocolo de efetivação (protocolo *commit*) que garanta a sincronia do estado da transação de maneira consistente. (SILBERSCHATZ; KORTH; SUDARSHAN, 2006).

3.4.2 Protocolos de Efetivação Distribuídos

O problema dos protocolos de efetivação consiste basicamente em chegar a um consenso. Vários protocolos já foram propostos a fim de resolver

este problema. A seguir serão descritos os Protocolos de Efetivação (*Commit*) em Duas Fases (2PC) e em Três Fases (3PC).

3.4.2.1 Protocolo *Commit* de Duas Fases (2PC)

Dentre os protocolos de efetivação propostos, o de duas fases (2PC) é um dos mais simples e mais utilizados. Como o próprio nome sugere, este protocolo é composto por duas fases: uma fase de **votação** e uma de **decisão**. A idéia principal deste protocolo é que todos os participantes, gerenciadores de transações (sub-transações) locais, sejam consultados pelo coordenador da transação a fim de verificar se eles estão preparados ou não para efetivarem (*commited*) suas transações. Se um participante votar pelo aborto da transação, ou não responder dentro do período de *timeout* determinado, então o coordenador deve instruir que todos os participantes abortem suas sub-transações. Se todos votarem pela efetivação, então todos os participantes serão ordenados a efetivarem suas sub-transações. Este protocolo assume que cada *site* tem seu próprio *log* local e, portanto pode de forma confiável efetivar ou desfazer uma transação.

Segundo Bell e Grimson (1992) as regras de votação são as seguintes:

- Cada participante tem direito a um voto que pode ser a favor da efetivação (*commit*), ou a favor do aborto (*abort*);
- Depois de votar, um participante não tem direito a mudar seu voto;
- Se um participante votar pelo aborto da transação, então ele está livre para abortar a sua sub-transação imediatamente;
- Se um participante votar pela efetivação da transação, então ele deve esperar pela mensagem broadcast *global-commit* ou *abort-*

commit que será enviada pelo coordenador, para então realizar a ação ordenada;

- Se todos os participantes votarem pela efetivação da transação, a decisão global do coordenador deverá ser pela efetivação da transação;
- A decisão global deve ser acatada por todos os participantes.

Segundo Silberschatz, Korth e Sudarshan (2006), para entendermos o funcionamento do protocolo 2PC consideremos uma transação *T* iniciada em um *site Si*, no qual o coordenador é *Ci*. O protocolo 2PC é iniciado por *Ci*, quando todos os *sites* que executaram *T* informam a *Ci* que *T* foi completado.

Primeira fase: o coordenador *Ci* grava um registro <*prepare T*> ao *log* e depois envia uma mensagem *prepare T* para todos os participantes de *T*. Quando os gerenciadores de transação desses sites recebem tal mensagem eles devem determinar se desejam confirmar ou não as suas partes de *T*. Caso a resposta seja não, eles devem gravar um registro <*no T*> ao *log* e depois enviarem uma mensagem *abort T* para *Ci*. Se a resposta for sim, eles devem gravar um registro <*ready T*> ao *log* e enviarem uma mensagem *ready T* para *Ci*.

Segunda fase: quando *Ci* receber a resposta *prepare T* de todos os participantes, ou quando um intervalo de tempo pré-determinado tiver passado desde que a mensagem *prepare T* foi enviada, *Ci* determina se *T* deve ser confirmada ou abortada. Se todos os *sites* participantes responderam com uma mensagem *ready T* a *Ci* a transação *T* pode ser confirmada. Caso contrário a transação *T* precisa ser abortada. Dependendo da decisão final do coordenador *Ci* um registro <*commit T*> ou um registro <*abort T*> é gravado no

log e uma mensagem *commit T* ou uma mensagem *abort T* é enviada para todos os *sites*. Após receber essa mensagem, o *site* registra a mensagem no *log* e confirma ou aborta a sua parte de *T*, conforme a decisão de *Ci*.

Durante a execução deste protocolo diversos tipos de falhas podem ocorrer. Para Silberschatz, Korth e Sudarshan (2006), as principais falhas são seguintes:

- **Falha em um site participante** – se a falha ocorrer antes do *site* responder a mensagem *prepare T* enviada pelo coordenador, o coordenador assumirá a resposta como sendo uma mensagem *abort T.* Caso a falha ocorra após a resposta, então a efetivação da transação prosseguirá normalmente. Quando o *site* conseguir recuperar-se ele deverá consultar o seu *log* a fim de verificar o status das transações que estavam sendo executadas no momento da sua falha. Dependendo do registro de controle (*commit, abort, ready*) encontrado pelo *site* em seu *log* medidas distintas deverão ser realizadas, conforme as especificadas a seguir:
    1. Registro ***<commit T>*** - ao deparar-se com este registro, o *site* participante deverá executar uma operação *redo(T)*, ou seja, refazer T;
    2. Registro ***<abort T>*** - neste caso a operação *undo(T)*, ou seja, desfazer T, deverá ser executada;
    3. Registro ***<ready T>*** - neste caso é necessário consultar o coordenador (*Ci*) para saber qual foi a decisão tomada como destino da transação. Se *Ci* estiver ativo, ele

responderá e o site deverá executar a operação *redo(T)* caso a resposta seja de efetivação e a operação *undo(T)* caso a resposta seja de cancelamento da transação *T*; Caso o coordenador esteja inativo, o site deverá mandar uma mensagem *query-status* aos demais *sites* participantes para que eles verifiquem os seus *logs* e informá-lo qual foi o destino de *T*;

4. Nenhum registro – esse caso indica que a falha ocorreu antes da resposta à mensagem prepare *T* e, portanto, o site deverá cancelar a transação, ou seja, deverá executar a operação *undo*(T).

- **Falha do coordenador** – caso o coordenador falhe durante a efetivação do protocolo de duas fases, teremos as seguintes situações:

    1. Se um dos sites (ativo) participantes da transação T contiver gravado em seu *log* um registro <*commit T*> ou um registro <*abort T*>, então os participantes deverão efetivar ou cancelar a transação, respectivamente;
    2. Se nenhum site ativo contiver em seu *log* o registro de controle <*ready T*>*,* a transação T deve ser abortada;
    3. Caso todos os *sites* ativos contenham um registro <*ready T*> em seus *logs,* mas não contenham nenhum registro <*commit T*> ou <*abort T*>, então é necessário esperar o retorno do coordenador *Ci*. Essa situação não é nem um pouco desejável, uma vez que todos os recursos alocados

pela transação *T* permanecerão bloqueados até que o coordenador *Ci* se recupere, e isso pode durar desde segundos até semanas.

- **Particionamento da rede** – caso haja uma divisão na rede onde a transação está sendo executada, haverá duas situações:
    1. Se todos os *sites* participantes e o coordenador permanecerem na mesma partição não haverá nenhum problema e a efetivação da transação poderá discorrer de forma normal;
    2. Caso haja uma divisão dos *sites* pertencentes à execução da transação, a partição em que o coordenador fizer parte seguirá o protocolo normalmente, supondo que os outros *sites* falharam. Os sites que ficaram na outra partição devem executar o protocolo tratando da falha do coordenador.

3.4.2.2 Protocolo *Commit* de Três Fases (3PC)

Este protocolo é uma extensão do protocolo *commit* de duas fases (2PC), que busca minimizar a possibilidade de incerteza em caso de falha do coordenador. É tolerante a falhas de sites e falhas de comunicação. Porém a complexidade da sua implementação é bem maior que a do 2PC, e ele proporciona um *overhead* maior na rede.

Para garantir que o coordenador, ao falhar, não bloqueie o sistema, foi adicionada uma terceira fase. Essa fase depende da segunda, pois só é executada se a decisão tomada na fase anterior for de pré-efetivação. Caso o coordenador falhe, um novo coordenador é escolhido, através de um protocolo

de eleição, e os sites podem trocar informações como o novo coordenador. Como as informações deste protocolo são mais completas, as chances de um novo coordenador tomar uma decisão correta são bem maiores do que no protocolo 2PC.

3.4.3 Recuperação e Compensação de Transações Distribuídas

As alterações de estado realizadas por cada transação em execução devem ser armazenadas para que possam ser posteriormente efetivadas em caso de sucesso, canceladas em caso de aborto ou recuperadas em caso de falhas. O armazenamento destas alterações é realizado através de arquivos de recuperação. As principais abordagens utilizadas são as de recuperação baseada em *log* e a baseada em cópias *shadow* (sombra), já abordadas no item 3.3 deste trabalho.

Uma das características dos sistemas distribuídos consiste na utilização de recursos de vários domínios administrativos. Como não se pode prever a duração de um processo especifico, a utilização de seções críticas e mecanismos de sincronização a fim de garantir a propriedade de isolamento entre transações pode acarretar em grandes atrasos. Isto se deve ao fato de que um recurso pode ser alocado por uma transação e levar muito tempo para ser liberado, sendo que as transações que precisam utilizar este recurso precisarão ficar aguardando a sua liberação.

Para resolver este problema pode-se implementar um modelo de transação baseado em compensação. Nesse modelo, as alterações do estado do sistema decorrentes das operações realizadas por uma transação tornam-se imediatamente visíveis para as outras transações, não necessitando de sincronização. Apesar da utilização desse modelo acarretar na perda da

propriedade isolamento, ela melhora consideravelmente o desempenho do sistema como um todo. Quando uma transação T é abortada, todas as transações que utilizaram os recursos alterados por ela deverão executar operações para compensar os efeitos indevidos causados, a fim de garantir a propriedade de consistência. Essa compensação nem sempre é possível; nesse caso o coordenador da transação deve informar o usuário para que possa tomar as medidas necessárias. (BELLI, 2005).

3.5 Transações em Web Services

Transações em Web Services possuem necessidades diferentes das necessidades previstas pelas transações tradicionais. Transações tradicionais ou transações básicas consistem em transações que implementam as propriedades de atomicidade, consistência, isolamento e durabilidade (ACID). Já as transações complexas, tais como as transações em Web Services, consistem em transações prolongadas e relaxadas. Transações prolongadas permitem o agrupamento de suas operações dentro de estruturas hierárquicas (sub-transações), enquanto que transações relaxadas indicam que um modelo de transação pode deixar de implementar alguma das propriedades ACID.

Segundo Limthanmaphon e Zhang (2004), as transações em Web Services não podem ser gerenciadas da mesma forma que as transações em sistemas distribuídos devido às seguintes características:

- Transações em web services geralmente são conduzidas além dos limites organizacionais. Isto significa que os participantes da transação podem ser independentes e estarem distribuídos pela Internet. Devido às limitações dos protocolos padrão de

gerenciamento de transações, não é possível dar suporte a este tipo de requisito.

- Transações em web services podem ser de longa duração, ou seja, podem durar horas, dias e até mesmo meses. A utilização de um controle rígido por *timeout* torna-se inviável diante de tal cenário, uma vez que seria impossível determinar o valor ideal do *timeout*. Além disso, se uma transação obtiver acesso exclusivo a um recurso por um longo período de tempo, outras transações não poderão acessar este recurso que estará bloqueado, resultando em uma redução na disponibilidade do sistema. Portanto, em transações em web services deve-se tomar o cuidado para que os recursos não fiquem bloqueados por um longo período de tempo. No entanto, a garantia deste quesito torna-se impossível quando se pretende seguir as propriedades ACID, pois ACID indica que os recursos utilizados por uma transação devem ser bloqueados até que a execução da transação termine, para que as propriedades de isolamento e consistência sejam preservadas.

Devido à necessidade de que estes problemas sejam resolvidos, alguns protocolos de suporte a transações em Web Services já foram propostos. A seguir, são abordados os protocolos *OASIS Business Transaction Protocol* (BTP) e *Web Service Coordination/Web Services Transactions*.

3.5.1.1 OASIS Business Transaction Protocol (BTP)

O protocolo BTP foi projetado para suportar aplicações independentes de localização e administração, que necessitam de suporte transacional diferente do suportado pelas transações ACID. Seu objetivo é tornar-se um protocolo

base que oferece suporte transacional em termos de coordenação distribuída para múltiplas funcionalidades de negócio independentes, na forma de serviço. Segundo Limthanmaphon e Zhang (*2004, apud BTP, Potts et, al. 2002, p. 173),* os objetivos da especificação BTP resumidamente são os seguintes:

- Fornecer um modelo de transações sobre a Internet;
- Integrar sistemas confiáveis sobre canais de comunicação não confiáveis;
- Gerenciar o ciclo de vida das transações e suportar as propriedades ACID;
- Suportar comunicações assíncronas entre sistemas fracamente acoplados;
- Fornecer suporte para transações de longa duração;
- Coordenar relações múltiplas e independentes entre transações e sub-transações.

Seu funcionamento baseia-se em grupos de consenso. Um grupo de consenso consiste em um grupo de participantes de uma transação que entram em acordo a respeito do resultado da transação. Participantes da mesma transação podem ser membros de grupos de consenso diferentes, o que permite que eles tomem decisões diferentes, infringindo deste modo uma das propriedades ACID. (BELLI, 2005)

BTP baseia-se no protocolo *commit* de duas fases para interações de curta duração conhecidas com *atom*, as quais podem ser combinadas e juntas formarem uma grande transação não ACID, conhecidas como *cohesion*. (LIMTHANMAPHON; ZHANG, 2004). O protocolo BTP trabalha com esses dois tipos de transações:

- ***Atom*** – neste tipo de transação tem-se a garantia de que o resultado será atômico, ou seja, todos os participantes entrarão em consenso sobre o mesmo resultado. No entanto, a semântica para este tipo de transação (isolamento, durabilidade, etc.), não é definida da mesma forma que na transação atômica;
- ***Cohesion*** – consiste em uma transação com atomicidade relaxada. Neste tipo de transação podemos ter diferentes resultados para cada participante de uma transação, de modo que a lógica de negócio poderá decidir pela efetivação da transação por parte de alguns participantes e pelo cancelamento por parte dos outros participantes.

Segundo Little e Freund (2003), o protocolo BTP não foi desenvolvido especificamente para transações em Web Services e a intenção é que ele possa ser utilizado em outros ambientes. Já os protocolos Web Services Coordination (WS-C) e Web Service Transactions (WS-TX) foram projetados especificamente para Web Services e, portanto, implementam as definições básicas da infra-estrutura de Web Services.

3.5.1.2 Web Services Coordination (WS-C) e Web Service Transactions (WS-TX)

WS-C e WS-Tx foram propostos por um consórcio de várias empresas, dentre elas a Microsoft, a IBM e a BEA System. Ambos possuem o propósito de definirem semânticas para as transações. (BELLI, 2005).

Segundo Little e Freund (2003), a especificação *Web Services Coordination* (WS-C) define um *framework* genérico de coordenação que pode suportar vários tipos de protocolos de coordenação. Por exemplo, um coordenador que execute um protocolo *commit* de 3 fases pode ser plugado à

implementação do WS-C, aprimorando-o. No entanto, independente do protocolo de coordenação utilizado, a especificação WS-C prevê as seguintes regras:

- Instanciação (ou ativação) de um novo coordenador para um protocolo de coordenação específico;
- Registro dos participantes da transação;
- Propagação das informações do contexto entre os Web Services envolvidos na aplicação;
- Uma entidade para gerenciar o protocolo de coordenação até sua finalização.

Os três primeiros requisitos estão diretamente relacionados à especificação WS-C enquanto que o quarto requisito é definido pela especificação WS-Tx, sendo que geralmente é a aplicação cliente que controla a aplicação como um todo. A *Figura 8* ilustra estas quatro regras e suas interações.

WS-C dispõem de um serviço de ativação (*Ativation Service*) o qual dá suporte à criação de coordenadores para protocolos específicos e para seus contextos. O processo de invocação é assíncrono. Desse modo, tanto a interface do serviço de ativação quanto a interface de invocação do serviço são definidas pela especificação WS-C.

Uma vez que o coordenador tenha sido instanciado e um contexto correspondente criado pelo serviço de ativação, um serviço de registro (*Registration Service*) precisa ser criado e disponibilizado. Este serviço permite que os participantes se registrem para receberem o protocolo de mensagem associado a um coordenador particular. Semelhante ao serviço de ativação, o

serviço de registro utiliza comunicação assíncrona, e, por isso, um arquivo WSDL para o serviço de registro e outro para a requisição do registro são especificados (LITTLE; FREUND, 2003).

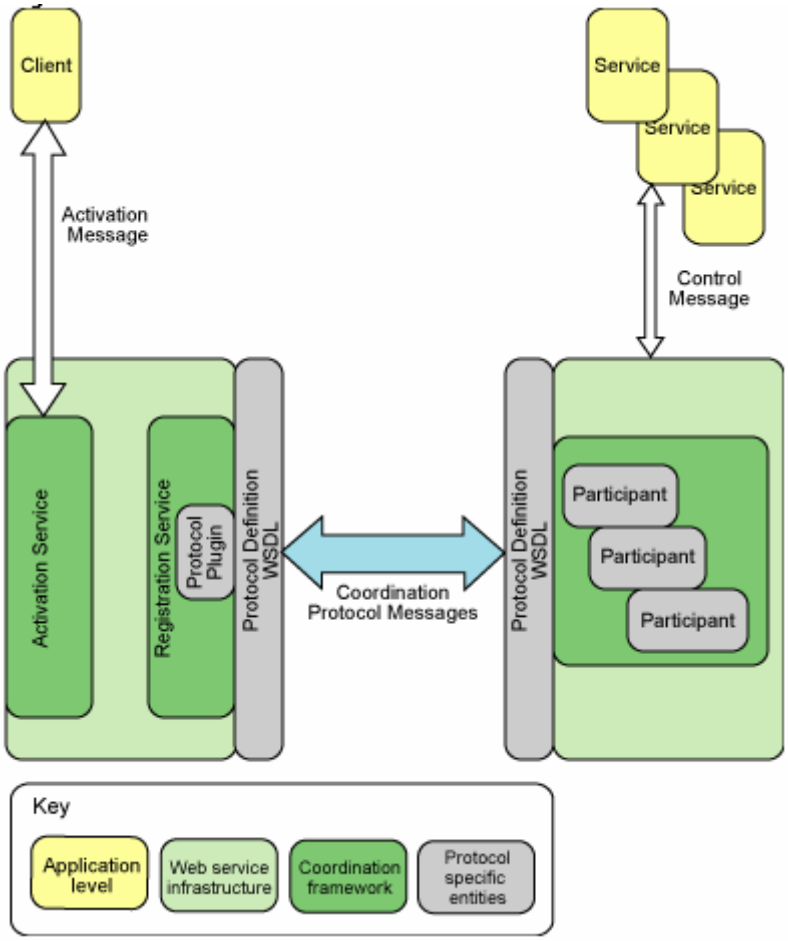

Figura 8 – Infra-estrutura da especificação WS-Coordination. (LITTLE e FREUND, 2003).

De acordo com Little e Freund (2003), o contexto possui uma função muito importante na coordenação das transações, pois ele é o responsável por dar suporte à conexão das aplicações pertencentes aos Web Services em uma única aplicação coordenada. O formato de um contexto deve conter os seguintes aspectos:

- Uma identificação única para cada coordenador, na forma de URI;
- Um endereço de serviço de registro, através do qual os participantes recebem o contexto e podem realizar o seu registro;

- Um valor *time-to-live* que indique por quanto tempo um contexto deve ser considerado válido;
- Informações a respeito das possíveis extensões do protocolo de coordenação atual suportado pelo coordenador.

O Ws-C é responsável por coordenar a transação, incluindo a criação, o registro de participantes e a finalização. As especificações WS-C e WS-Tx interagem para executarem as tarefas do WS-C. (BELLI, 2005).

De acordo com Limthanmaphon e Zhang (2004), a especificação *Web Service Transaction* (WS-Tx) define como os web services coordenam sua atividades, em seqüência, a fim de assegurar a integridade das operações básicas dos bancos de dados. Dois modelos de transação são definidos pela especificação WS-Tx, de modo que cada uma suporta uma semântica diferente de interação *business-to-business* (negócio): transações atômicas e transações de negócios.

O modelo de transação atômica (AT) é utilizado para coordenar atividades de curta duração executadas dentro de domínios com alto nível de proteção. Transações atômicas seguem as propriedades ACID e garantem que todos os participantes terão o mesmo resultado (atômico): todos efetivam a transação ou todos abortam a transação (propriedade “tudo ou nada”). (LIMTHANMAPHON e ZHANG, 2004).

O modelo AT é baseado no protocolo de efetivação (*commit*) de duas fases, abordado na seção 3.4.2.1 deste trabalho, o qual funciona da seguinte maneira: cada participante aloca (bloqueia) os registros do banco de dados envolvidos na transação para impedir que qualquer mudança seja feita nos dados durante o processamento da mesma (isolamento); somente quando

todos os participantes da transação indicam ao coordenador que a transação pode ser efetivada, este os instrui a realizarem as modificações. Se algum dos participantes abortar a transação ou falhar ao responder, o coordenador instrui todos os participantes a abortarem a transação e descartarem as modificações realizadas pela transação que estava sendo executada. (LIMTHANMAPHON e ZHANG, 2004).

O problema desta proposta é que cada banco de dados envolvido na transação deve controlar os bloqueios aos seus registros para que eles não durem muito tempo, pois isto acarretará em uma alta indisponibilidade dos dados aos outros usuários. Enquanto que transações sobre redes internas geralmente são de curta duração, transações em Web Services podem ser de longa duração (questão de horas, dias ou até mesmo semanas) e isto pode ser um grande problema no gerenciamento de recursos, uma vez que os recursos alocados para a transação em questão podem ficar indisponíveis por muito tempo.

De acordo com Little e Freund (2003), o modelo de transação de negócios (BA – *Business Activity*) fornece flexibilidade às propriedades das transações e foi projetado especificamente para interações de longa duração, nas quais o controle de recursos torna-se muitas vezes impossível ou impraticável.

Para Limthanmaphon e Zhang (2004), geralmente uma BA é definida como uma atividade que consiste em uma seqüência de tarefas, onde cada tarefa satisfaz uma restrição de uma transação atômica. Sua principal característica consiste na compensação de transações. Conforme Little e Freund (2003), neste modelo, requisições de serviços, como por exemplo, uma reserva de hotel, são feitas ao provedor de serviços, o qual executa o serviço

de forma que ele pode ser desfeito caso haja alguma falha, através do mecanismo de compensação. Desse modo, se o BA decidir que por causa de alguma falha o serviço precisa ser cancelado, ele instrui o provedor do serviço a desfazer o serviço prestado.

Apesar de não conseguir garantir todas as propriedades ACID, o modelo de transações de negócios (BA) consegue garantir a propriedade consistência através de ações de compensação.

De acordo com Cabrera et al. (2005), o modelo de transação de negócios (BA) possui as seguintes características:

- Transações de negócios podem utilizar vários recursos por um longo período de tempo;
- Uma quantidade significativa de transações atômicas podem estar envolvidas;
- O resultado de tarefas individuais pertencentes a uma transação de negócio podem ser vistas antes da conclusão da transação;
- A resposta de uma requisição pode levar um longo período de tempo para ser emitida, pois uma aprovação, reunião, produção ou entrega humana podem ter que ocorrer antes que a resposta seja enviada;
- No caso em que uma exceção do negócio requer que uma transação seja desfeita logicamente, um aborto da mesma geralmente não é suficiente. Os mecanismos de tratamento de exceção podem precisar da lógica do negócio para realizarem um tratamento, por exemplo, um mecanismo de compensação. Para

reverter os efeitos de uma tarefa completada previamente precisa conhecer a lógica do negócio;

- Participantes de uma transação de negócio podem estar dentro de domínios de confiança diferentes, onde todas as relações de confiança são estabelecidas explicitamente.

Com base nestas características, verificou-se que os seguintes aspectos devem ser abordados pela especificação WS-Tx:

- Todas as transições de estado devem ser gravadas de forma confiável, incluindo metadados do estado da aplicação e da coordenação da transação;
- Todas as notificações devem ser reconhecidas pelo protocolo, a fim de assegurar uma visão consistente do estado entre o coordenador e os participantes;
- Cada notificação é definida como uma mensagem individual (CABRERA et al., 2005).

3.6 Considerações Finais

Este capítulo descreveu o contexto no qual as transações em Web Services estão inseridas, bem como os problemas enfrentados pelos sistemas que pretendem dar suporte a elas.

Conforme citado anteriormente, o objetivo deste trabalho consiste no estudo e utilização de uma infra-estrutura que dê suporte a transações distribuídas em Web Services. Após uma pesquisa e avaliação das infra-estruturas existentes optamos por utilizar a infra-estrutura proposta projeto *Apache Kandula*, a qual implementa os protocolos *WS-Coordination e WS-AtomicTransaction.* A escolha desta implementação deve-se ao fato de tratar-

se de uma implementação código aberto, desenvolvida sobre o *framework Apache Axis*, o qual possui ampla aceitação e adoção na construção de WS.

O capítulo 4, a seguir, explica de forma mais detalhada o funcionamento dos protocolos *WS-Coordination e WS-AtomicTransaction* e a maneira como eles interagem, a fim de proporcionar um gerenciamento de transações eficiente.

4 Cenário de Aplicação

O cenário de aplicação implementado neste trabalho a fim de demonstrar a utilização de uma infra-estrutura de suporte a transações distribuídas em Web Services é o de comércio eletrônico, mais precisamente uma livraria virtual. A escolha deste cenário deve-se ao fato da popularidade do comércio eletrônico nos dias atuais, bem como a sua importância para a sobrevivência das empresas no mercado atual, o qual é vorazmente competitivo. A implantação do comércio eletrônico tem se tornado um grande diferencial entre empresas concorrentes, uma vez que visa comodidade para o cliente e redução de custos para as empresas.

Segundo Albertin (2000), comércio eletrônico (CE) é a realização de todo o processo de negócio em um ambiente eletrônico, através da utilização de tecnologias de comunicação e de informação, que visa atender os objetivos do negócio. As aplicações de CE podem ser classificadas da seguinte forma:

- *Business-to-business* (B2B) – transações entre empresas;
- *Business-to-consumer/Consumer-to-business* (B2C/C2B) – transações entre empresas e consumidores;
- *Business-to-government/Government-to-business* (B2G/G2B) – transações entre empresas e governo;
- *Consumer-to-consumer* (C2C) – transações entre consumidores fiinais;
- *Government-to-consumer/consumer-to-government* (G2C/C2G) – transações entre governo e consumidores; e

- *Government-to-government* (G2G) – transações entre entidades do governo.

O cenário é composto por três aplicações uma Livraria, uma Operadora de cartões de crédito e um Fornecedor de produtos (livros). A operadora de cartões e o fornecedor de produtos desempenharão o papel do agente de software *provedor de serviços* da arquitetura SOA, disponibilizando serviços web (Web Service) que serão posteriormente consumidos pela Livraria, que assumirá o papel do agente de software *consumidor de serviços*. A fim de simplificar a implementação deste cenário, não faremos a persistência dos dados, ou seja, os dados serão manípulados diretamente na memória do computador.

O serviço disponibilizado pela Operadora de Cartões consistirá na autorização ou não de uma compra por determinado cliente com base no saldo do seu cartão e no valor da sua compra. Por se tratar de um cenário hipotético, somente para fins didáticos, a única forma de pagamento que está Livraria suportará será através do cartão de crédito. Já o serviço disponibilizado pelo Fornecedor de Produtos informará à Livraria se determinada quantidade de um certo produto está disponível ou não para venda. Estas três aplicações foram implementadas utilizando-se a linguagem de programação Java, com o auxílio da IDE Eclipse versão 3.2.

No cenário proposto podemos identificar aplicações de comércio eletrônico do tipo *B2B*, representado pela interação entre a livraria e a operadora de cartões de créditos e pela interação entre a livraria e o fornecedor dos produtos, e do tipo *B2C/C2B*, representado pela interação entre a livraria e o cliente.

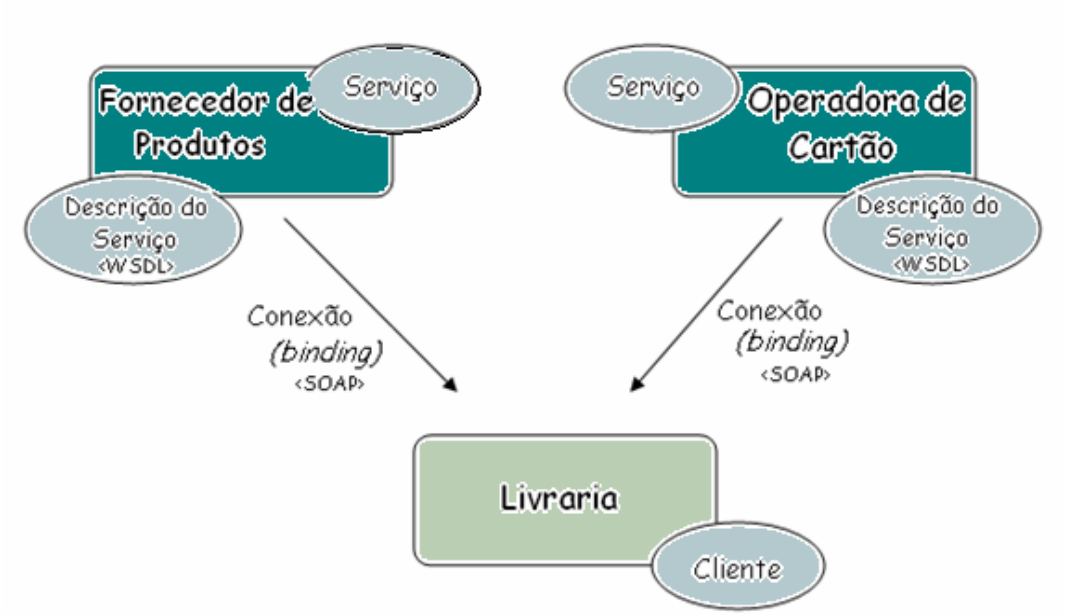

Figura 9 – Arquitetura do cenário proposto.

O objetivo deste trabalho, conforme já apresentado anteriormente, consiste em utilizar uma proposta de suporte a transações distribuídas em web services. Para tanto será utilizado a implementação dos protocolos *WS-Coordination e WS-AtomicTransaction* disponibilizadas pelo projeto *Apache Kandula.* (KANDULA, 2007a).

O projeto *Apache Kandula* implementa os protocolos *WS-Coordination e WS-AtomicTransaction* sobre o Apache Axis. Ele possui duas versões: o Kandula1 que roda sobre o Apache Axis 1.x e o Kandula2 que roda sobre o Apache Kandula 2.x.. Neste trabalho será utilizada a ramificação Kandula1, cujo código fonte está disponível para download via SVN em https://svn.apache.org/repos/asf/webservices/kandula/branches/Kandula_1/, pois ela apresenta atualmente um suporte maior ao usuário no quesito documentação. Outro aspecto que nos motivou a utilizar esta versão foi o fato da mesma rodar sobre o Apache Axis, que por se tratar de uma versão um

pouco mais antiga do que o Apache Axis2, é mais estável e também mais amigável.

O Apache Axis é um framework *open source*, implementado atualmente na linguagem de programação Java e baseado no padrão XML. Ele é utilizado na construção de Web Services baseados no padrão SOAP. A versão utilizada na implementação deste trabalho foi a versão 1.4, disponível para download em http://ws.apache.org/axis/.

A implementação do Web Service envolveu ainda a utilização de outras tecnologias como o Apache TomCat versão 5.5.23, disponível para *download* em http://tomcat.apache.org/download-55.cgi, e o parser XML *Xerces Java Parser* versão 2.9.0, disponível para *download* em http://archive.apache.org/dist/xml/xerces-j/. O Apache Tomcat é um contêiner de Servlets, o qual será utilizado para disponibilizarmos os serviços Web da Operadora de Cartões e do Fornecedor de Produtos.

A instalação do Apache Kandula requer a instalação do Apache Maven e do Apache Ant, as versões utilizadas neste trabalho foram respectivamente a versão 1.0.2, disponível para *download* em http://archive.apache.org/dist/maven/binaries/, e a versão 1.7.0 do Apache Ant, disponível para download em http://ant.apache.org/. O Apache Maven é utilizado para construção do Kandula, e através dele é feito o *download* dos pacotes (arquivos .JAR) necessários para o funcionamento do Kandula, inclusive do Apache Geronimo, que será descrito a seguir.

A *Figura 10*, a seguir, apresenta a arquitetura em camadas do cenário proposto. Como podemos observar, os Serviços da Operadora de Cartões de Crédito e do Fornecedor de Produtos rodam sobre o Apache Axis e Apache

Geronimo, que por sua vez rodam sobre a Máquina Virtual do Java (JVM). Já o controle global das transações, realizado pela aplicação Livraria, roda sobre o Apache Kandula, que por sua vez roda sobre o Apache Axis, que conforme mencionado anteriormente roda sobre a JVM.

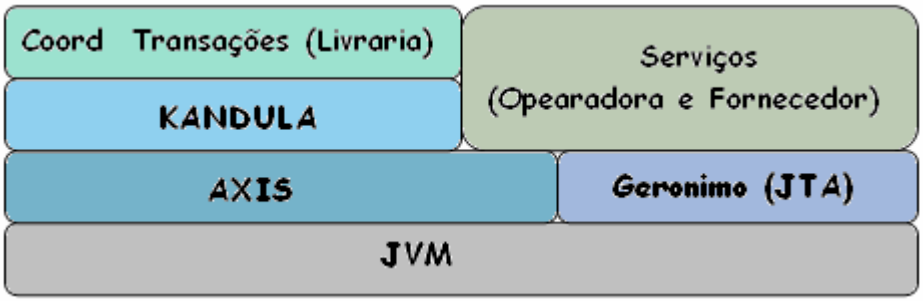

Figura 10 – Arquitetura em Camadas.

A arquitetura do apache Kandula, apresentada na *Figura 11*, funciona da seguinte maneira: Um serviço de coordenação é responsável por coordenar atividades. Na arquitetura Kandula, o termo coordenador é usado para referenciar componentes individuais em tempo de execução que coordenam atividades, e o termo serviço de coordenação é utilizado para referenciar o conjunto de todos os serviços que compõem esta arquitetura, tais como, ativação, registro, etc.

Quando uma requisição é recebida por um *endpoint,* ela pode ser tratada de duas formas diferentes. Quando se trata de uma requisição de criação de atividades, um novo coordenador é criado para a nova atividade. Quando se trata de requisições normais, as informações referentes à requisição são passadas para o coordenador responsável pela atividade de interesse, o qual deve tratar a operação requerida. À medida que as atividades terminam de ser executadas, os coordenadores responsáveis por elas são destruídos. (KANDULA, 2007B).

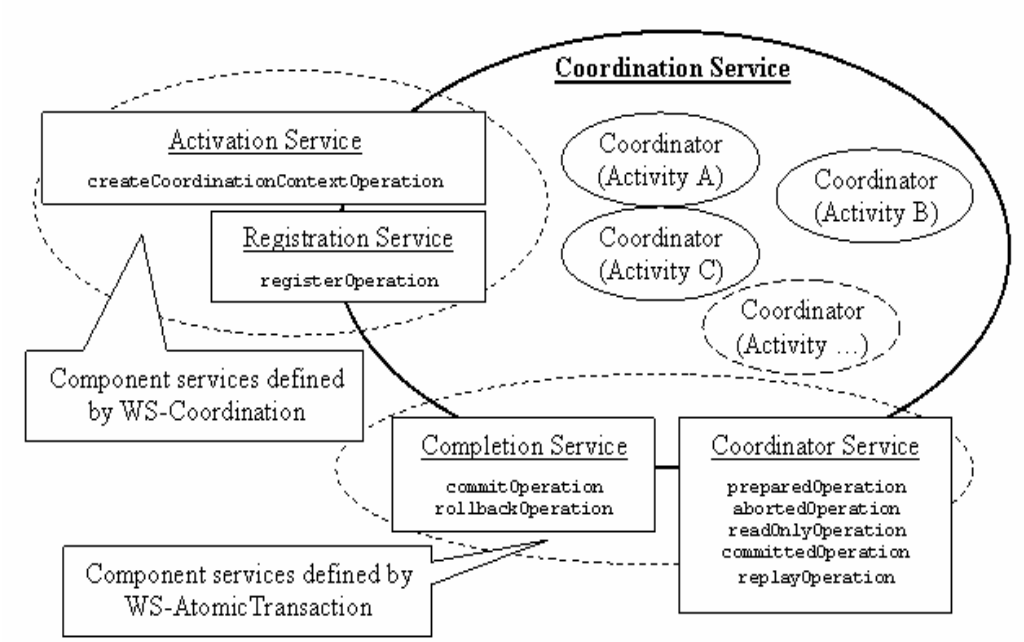

Figura 11 – Arquitetura do Serviço de Coordenação. (KANDULA, 2007b).

Conforme Kandula (2007b) um repositório central contendo todos os coordenadores ativos é mantido pelo serviço coordenador. Quando uma nova requisição chega a qualquer *endpoint*, o identificador de atividade é utilizado para localizar o coordenador responsável por aquela atividade específica para despachar a requisição. A especificação do protocolo WS-*Coordination* define os *endpoints* de ativação e de registro. Os demais *endpoints* são especificados por protocolos específicos tais como *WS-AtomicTransaction, WS-BusinessActivation*, etc.

O *endpoint* de ativação permite que novas atividades sejam criadas e o *endpoint* de registro permite que participantes sejam registrados em qualquer atividade pelo serviço de coordenação. O objetivo de todos os outros *endpoints* é facilitar a comunicação entre os participantes e o serviço de coordenação baseado em um protocolo de coordenação especifico. (KANDULA, 2007B).

Quando uma nova atividade é criada, o serviço de coordenação primeiramente determina o seu tipo, como por exemplo, *AtomicTransaction*.

Então ele cria um coordenador do tipo apropriado para coordenar a atividade. Todos os coordenadores devem implementar as operações definidas pelo *WS-Coordination*, como por exemplo, a criação de um contexto de coordenação para a atividade, o registro de participantes, etc. Adicionalmente, cada coordenador precisa suportar todas as operações exigidas pelo tipo de atividade que ele coordena.

O serviço de coordenação é totalmente implementado pelo projeto Apache Kandula, e consiste em um conjunto de serviços web, disponibilizados pela Apache no pacote de instalação do Kandula. Estes serviços precisam ser disponibilizados (*deployment*) pelo desenvolvedor que irá utilizar a implementação do Kandula - as instruções para a publicação destes serviços estão disponíveis no site do projeto. As classes que compõem o serviço de coordenação bem como o seu funcionamento serão descritos na seção 4.3 deste trabalho.

Apesar do *framework* de coordenação de web services ser independente de plataforma, os serviços participantes inevitavelmente precisam utilizar tecnologias de plataformas específicas para realizar o controle transacional local. Por exemplo, aplicações implementadas em Java EE devem utilizar a API Java Transaction (JTA).

JTA é uma API pertencente à plataforma Java EE, que especifica interfaces para a demarcação de transações em aplicações escritas na linguagem Java. Através da interface JTA o desenvolvedor interage com o monitor de transação para determinar as fronteiras de uma aplicação, isto é, através da interface JTA ele define o inicio da transação e determina se ela será confirmada (*commit*) ou não (*rollback*).

Segundo Cheung, Matena (2007), JTA especifica interfaces Java locais entre um gerenciador de transação e as partes envolvidas em um sistema de transação distribuído: a aplicação, o gerenciador de recursos e o servidor de aplicação. A API JTA é composta por:

- Uma interface de aplicação de alto nível, que permite a uma aplicação transacional demarcar os limites de uma transação;
- Um mapeamento Java do protocolo padrão da indústria X/Open XA, que permite a um gerenciador de recursos transacional participar de uma transação global controlada por um gerenciador de transação externo;
- Uma interface de gerenciamento de transações de alto nível, que permite a um servidor de aplicação controlar a demarcação dos limites da transação para uma aplicação sendo gerenciada pelo servidor de aplicação.

Dentro do contexto de coordenação de atividades, o JTA é requerido para, juntamente com um coordenador externo, decidir se e quando uma atividade deve ser confirmada. A implementação Kandula intercala a implementação local JTA e um coordenador externo para permitir a propagação das transações de um domínio Java EE para um domínio de serviços web e vice-versa.

Portanto, dentro do cenário proposto neste trabalho, será necessário utilizar um controle de transação local baseado na API JTA tanto na Operadora de cartões de créditos como no Fornecedor de produtos. No entanto, JTA consiste apenas em uma especificação de como o controle de transações deve ser realizado. Desse modo, para utilizarmos este recurso será necessário

implementar as interfaces especificadas pela API JTA ou utilizar alguma implementação pronta que esteja disponível.

Atualmente Kandula1 suporta somente o controle de transações especificado pela API JTA e implementado pelo Apache Geronimo v. 1.1. Portanto, adotaremos este controle de transações tanto na Operadora de cartões de crédito quanto no Fornecedor de produtos. A seguir será especificado e demonstrado como foi feito o controle de transações local nestas duas aplicações.

4.1 Aplicação Operadora de Cartões de Crédito

A *Figura 12* apresenta o diagrama de classes da aplicação Operadora de Cartão de Crédito. Os métodos *setters* e *getters* não estão representados no diagrama para não poluí-los visualmente, no entanto deve-se considerar que todos os atributos, apresentados no diagrama, possuem os seus respectivos métodos *set* e *get*. Como pode ser observado no diagrama de classes, esta aplicação é composta por dez classes. As classes *Cartao, Fatura, ItemFatura, Cliente e Endereco* representam entidades que fazem parte da lógica do problema e possuem somente atributos e os respectivos métodos *set* e *get*.

As classes *GerenciadorCartoes* e *GerenciadorClientes* são classes de controle que gerenciam respectivamente as entidades *Cartao* e *Cliente*. São elas que controlam a criação, inserção e remoção destes objetos durante a execução da aplicação.

A classe *OperadoraDBMS* implementa a *interface org.apache.geronimo.transaction.manager.NamedXAResource*, a qual por sua vez estende a *interface javax.transaction.xa.XAResource* da API JTA, adicionando apenas a assinatura de um método, *getName()*. Portanto, a classe

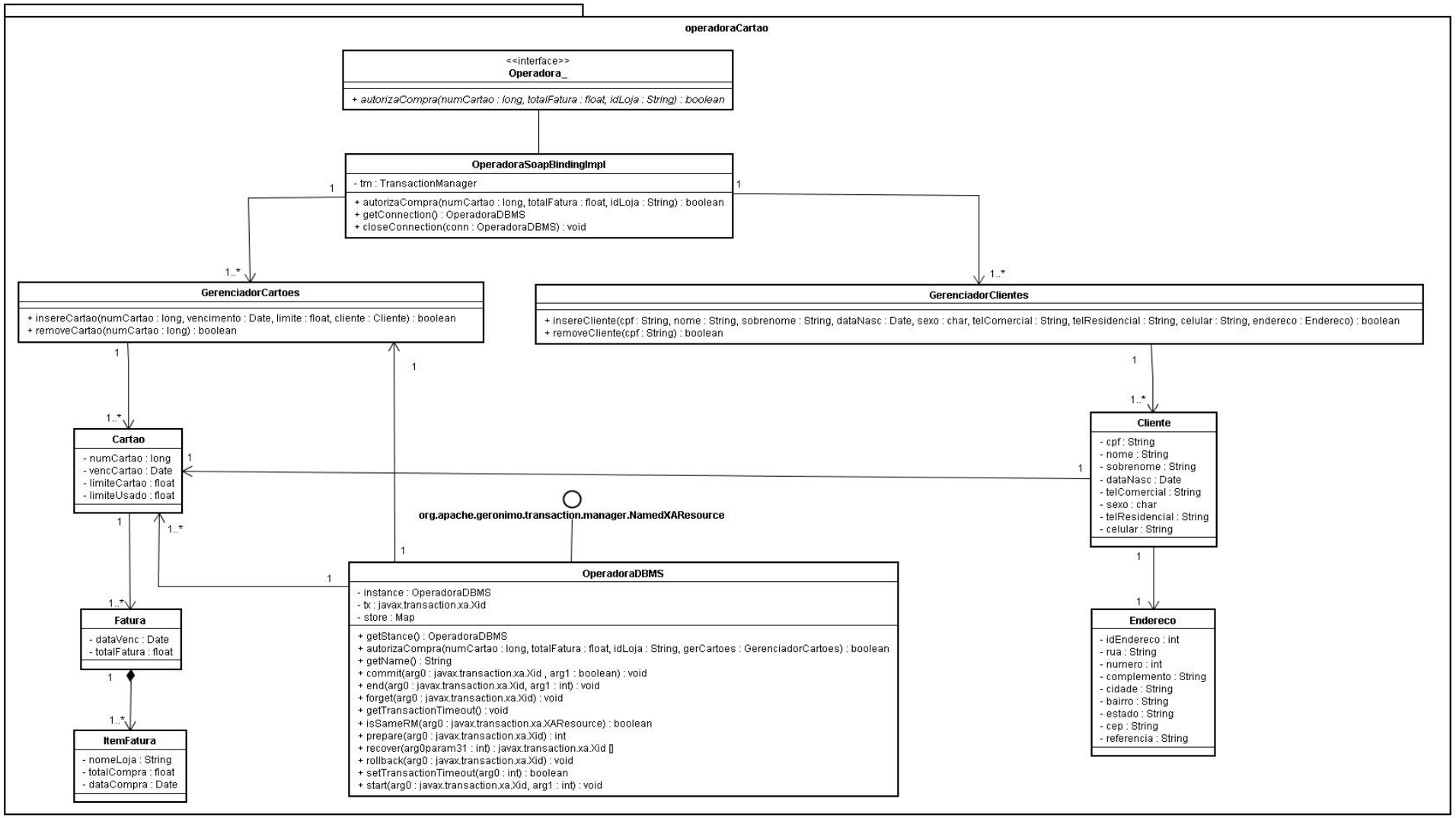

Figura 12 – Diagrama de classes da aplicação Operadora de Cartões de Crédito.

OperaDBMS implementará os métodos definidos pela *interface javax.transaction.xa.XAResource* e o método *getName()* definido pela *interface org.apache.geronimo.transaction.manager.NamedXAResource.*

Conforme Cheung, Matena (2007), a *interface javax.transaction.xa.XAResource* define o contrato entre um gerenciador de recursos e um gerenciador de transações em um ambiente de processamento de transações distribuídas (DTP). A classe *OperadoraDBMS* desempenhará a função de um gerenciador de recursos que implementa a interface *XAResource* a fim de suportar a associação de um recurso transacional. Neste caso especifico, a operadora de cartões, a uma transação global. Uma transação global consiste em uma unidade de trabalho que pode ser executada por um ou mais gerenciadores de recursos (RM) em um ambiente de processamento de transações distribuídas (DTP).

Em uma transação global, um gerenciador de transações obtém um *XAResource* para cada recurso transacional participante. O método *start()* é utilizado por ele para associar um recurso a transação global e o método *end()* é utilizado para desassociar à transação do recurso. O gerenciador de recursos é responsável por associar todas as ações executadas sobre os dados à transação global entre a invocação dos métodos *start()* e *end().* (CHEUNG, MATENA, 2007).

A classe *OperadoraDBMS* possui um atributo do tipo *javax.transaction.xa.Xid,* que consiste em um mapeamento Java das transações *X/Open*. Esta *interface* implementa a infra-estrutura de identificação de transações do padrão *XA.* Ela define meios para acesso do id (identificador) da transação que é utilizado pelo gerenciador de transações e de recursos.

A classe *OperadoraSoapBindingImpl* implementa a *interface* *operadoraCartao.Operadora*, que nada mais é do que uma *interface* que especifica o método *autorizaCompra()*, o qual será disponibilizado como um serviço web. Essa classe possui o atributo *tm* que é do tipo *org.apache.geronimo.transaction.manager.TransactionManagerImpl* que implementa a *interface javax.transaction.TransactionManager* da API JTA. Segundo Cheung, Matena (2007), esta *interface* permite que o servidor de aplicação controle as fronteiras transacionais em nome da aplicação que está sendo gerenciada. O *container*, no nosso caso o Apache Tomcat, utiliza a *interface TransactionManager* principalmente para demarcar as fronteiras transacionais onde operações afetam o contexto transacional das *threads* chamadas.

O gerenciador de transações mantém um contexto transacional associado com *threads* como parte de uma estrutura de dados interna. O contexto transacional da *thread* pode ser *null* ou referenciar uma transação global especifica. Várias *threads* podem concorrentemente serem associadas à mesma transação global. Cada contexto transacional é encapsulado por um objeto *Transaction*, o qual pode ser utilizado para executar operações que são especificas para transação alvo, sem considerar o contexto transacional da *thread* chamada. (CHEUNG, MATENA, 2007).

Conforme citado anteriormente, a aplicação Operadora de Cartões de Crédito disponibilizará um serviço web, que ao ser acessado por um cliente, que informará o número do cartão de crédito do cliente que está efetuando a compra juntamente com o valor total da compra, enviará uma resposta a ele autorizando ou não a compra especificada. Este serviço web tem a sua lógica implementada pelo método *autorizaCompra(**long** numeroCartao, **float** totalFatura, **String** idLoja)*

definido pela *interface Operadora* e implementado pela classe *OperadoraSoapBindingImpl* . Portanto, ao acessar o serviço web disponibilizado pela Operadora de Cartões de Crédito, o cliente estará invocando o método *autorizaCompra.*

A *Figura 13* apresenta o trecho de código no qual o método *autorizaCompra* é implementado. O método *getConnection(),* também implementado pela classe *OperadoraSoapBindingImpl,* é invocado durante a execução do método *autorizaCompra* e caso seja executado com sucesso retornará um objeto do tipo *OperadoraDBMS* que implementa a *interface javax.transaction.xa.XAResource* conforme visto anteriormente. O objetivo do método *getConnetion()* consiste em instanciar um gerenciador de recursos, neste caso específico o objeto *OperadoraDBMS* que é do tipo *XAResource*, e registrá-lo. Este registro é realizado através da execução do método *enlistResource*, que é realizada pelo objeto *tm*, que consiste em uma instância da implementação da *interface javax.transaction.TransactionManager*.

O método *enlistResource()* é implementado pela classe *org.apache.geronimo.transaction.manager.TransactionImpl* que implementa a *interface javax.transaction.Transaction* da API JTA. Segundo Cheung, Matena (2007), a *interface Transaction* permite que operações sejam executadas sobre transações associadas a objetos alvos. Todas as transações globais são associadas a um objeto *Transaction* quando a transação é criada. O objeto *Transaction* pode ser utilizado para:

- Registrar os recursos transacionais que estão sendo utilizados pela aplicação;
- Registrar *callbacks* de sincronização para transações;

- Realizar o *commit* (confirmação) ou *rollback* (não confirmação) de uma transação;
- Obter o *status* da transação;

```
public boolean autorizaCompra(long numeroCartao, float totalFatura, String idLoja)
throws RemoteException {

    try {
        OperadoraDBMS conn = getConnection();
        boolean autorizacao = conn.autorizaCompra(numeroCartao, totalFatura, idLoja,
                gerCartoes);
            closeConnection(conn);
            return autorizacao;
        } catch (Exception e) {
            throw new RemoteException(e.getMessage());
        }
}

private OperadoraDBMS getConnection() throws IllegalStateException, RollbackException,
SystemException {
    OperadoraDBMS conn = OperadoraDBMS.getInstance();
    tm.getTransaction().enlistResource(conn);
    return conn;
}

private void closeConnection(OperadoraDBMS conn)throws IllegalStateException,
SystemException {
    tm.getTransaction().delistResource(conn, XAResource.TMSUSPEND);
}
```

Figura 13 – Trecho de código da classe OperadoraSoapBindingImpl.

Em tempo de execução, um servidor de aplicação fornece à aplicação uma infra-estrutura que inclui o gerenciamento dos recursos transacionais. No entanto, para que um gerenciador de transações externo, no nosso caso o implementado pelo projeto Apache Kandula, possa coordenar as ações transacionais executadas pelo gerenciador de recursos, é necessário que o servidor de aplicações *enlist* (registre) e *delist* os recursos utilizados na transação.

O registro (*enlistment*) de recursos realizado pelo servidor de aplicações possui dois propósitos:

- Informar ao Gerenciador de Transação a respeito da instância do gerenciador de recurso participante da transação global;
- Habilitar o Gerenciador de Transações a agrupar os tipos de recursos em uso por cada transação. O agrupamento de recursos permite que

o Gerenciador de transação conduza o protocolo transacional *commit* de duas fases entre o Gerenciador de Transações e os Gerenciadores de Recursos como definido pela especificação X/Open XA.

Para cada recurso utilizado pela aplicação, o servidor de aplicação invoca o método *enlistResource()* e especifica o objeto *XAResource* que identifica o recurso em uso. A requisição *enlistResource* resulta no Gerenciador de Transação informando ao gerenciador de recursos para iniciar a associação da transação com as ações executadas através do recurso correspondente. Está associação é feita através da invocação do método *start()*, que no nosso caso é implementado pela classe *OperadoraDBMS.*

O método *start* tem como função associar transações globais a um recurso transacional, e o método *end* tem como função desassociá-los.

O gerenciador de transações pode intercalar múltiplos contextos transacionais, que utilizam o mesmo recurso, através da invocação apropriada dos métodos *start* e *end* para cada troca de contexto transacional. Cada vez que um recurso precisar ser associado com uma transação diferente, é necessário que o método *end* tenha sido invocado pela última transação associada com o recurso em questão, e que o método *start* seja invocado pelo contexto transacional atual. O gerenciador de recursos *XAResource* não suporta transações aninhadas. Ou seja, ocorrerá um erro na invocação do método *start* se a instância do *XAResource* passada como parâmetro na chamada do método *start* já estiver associada a uma outra transação. (CHEUNG, MATENA, 2007).

```
public void start(Xid arg0, int arg1) throws XAException {
    System.out.println("[OperadoraDBMS] start");
    if (locked)
        throw new XAException();
    this.tx = arg0;
    cache = (Cartao) store.get(arg0);
    if (cache == null) {
        cache = new Cartao();
        this.cache = cartao;
        store.put(arg0, cache);
    }
}

public void end(Xid arg0, int arg1) throws XAException {
    System.out.println("[OperadoraDBMS] end");
    cache = null;
}
```

Figura 14 – Implementação dos métodos start() e end() pela classe OperadoraDBMS.

De acordo com o trecho de código apresentado na *Figura 13* após a execução do método *enlistResource(),* que retorna uma instância do gerenciador de recurso *XAResource*, o objeto retornado executa o método *autorizaCompra()* implementado pela classe *OperadoraDBMS*. Esse método primeiramente verifica se o número do cartão, passado como parâmetro na sua chamada, é válido e se o total da compra, também passado como parâmetro, é inferior ou igual ao limite disponível do cartão. Caso as duas verificações sejam positivas, o gerenciador de cartões inclui esta compra na fatura do cliente e retorna como resposta o valor *true*. Caso contrário, ele retorna como resposta o valor *false,* indicando que a venda não deve ser realizada porque ou o cartão é inexistente ou o valor da compra ultrapassou o limite disponível para compras do cartão em questão.

Ainda de acordo com a *Figura 13*, na seqüência da execução do método *autorizaCompra*, o método *closeConnection()* é invocado, o qual por sua vez invoca o método *delistResource* implementado pela classe *org.apache.geronimo.transaction.manager.TransactionImpl* que implementa a *interface javax.transaction.Transaction* da API JTA. O método *delistResource* é usado para desassociar um recurso especifico do contexto transacional no qual

estava inserido. A invocação deste método exige a passagem de dois parâmetros:

- Um objeto *XAResource* que representa o recurso;
- Uma *flag* que indica que o *delistment* está ocorrendo devido ao fato de que:
    - A transação foi suspensa (TMSUSPEND);
    - Houve alguma falha (TMFAIL);
    - Um recurso foi normalmente liberado por uma aplicação (TMSUCCESS).

A requisição *delist* resulta no gerenciador de transações informando ao gerenciador de recursos que ele deve finalizar a associação da transação com o recurso representado pela instância *XAResource.* Essa dissociação, conforme visto anteriormente, ocorrerá pela invocação do método *end()*, implementado na Classe *OperadoraDBMS* e apresentado na *Figura 14* .

4.2 Aplicação Fornecedor de Produtos

A *Figura 15* apresenta o diagrama de classes da aplicação Fornecedor de Produtos (novamante, os métodos *setters* e *getters* não estão representados no diagrama, mas todos os atributos apresentados no diagrama possuem os seus respectivos métodos *set* e *get*). Como pode ser observado no diagrama de classes, esta aplicação é composta por dez classes. As classes *Pedido, ItemPedido, ItemEstoque e Produto* representam entidades que fazem parte da lógica do problema e possuem somente atributos e os respectivos métodos *set* e *get*.

As classes *GerenciadorEstoque, GerenciadorPedidos e GerenciadorProdutos* são classes de controle que gerenciam respectivamente as

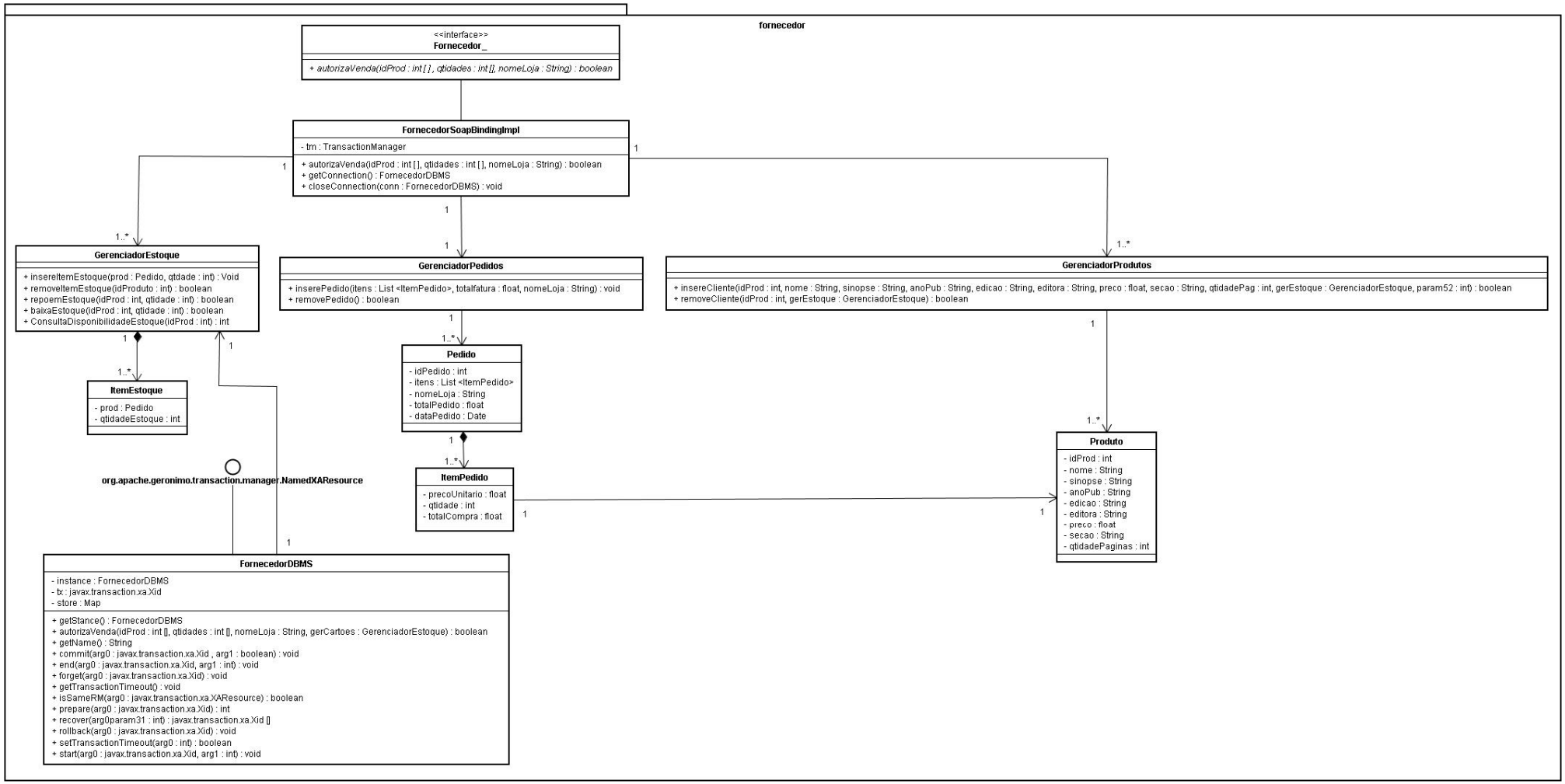

Figura 15 – Diagrama de classes da aplicação Fornecedor de Produtos.

entidades *ItemEstoque, Pedido* e *Produto*. São elas que controlam a criação, inserção e remoção destes objetos durante a execução da aplicação.

A classe *FornecedorDBMS* possui a mesma função da classe OperadoraDBMS, explicada no item anterior, ou seja, ele implementa o gerenciador de recursos *XAResource*.

A classe *FornecedorSoapBindingImpl* implementa a *interface fornecedor.Forenecedor*, que nada mais é do que uma *interface* que especifica o método *autorizaVenda()*, o qual será disponibilizado como um serviço web. Essa classe é muito similar à classe *OperadoraSoapBindingImpl,* abordada no item anterior, possuindo o mesmo atributo *tm* e os mesmo métodos *getConnection()* e *closeConnection()*.

Conforme citado anteriormente, a aplicação *Fornecedor de Produtos* disponibilizará um serviço web de autorização de vendas. Este serviço tem como objetivo fornecer ao seu cliente um mecanismo de consulta da disponibilidade ou não de uma certa quantidade de um determinado produto.

O funcionamento deste serviço ocorrerá da seguinte forma: para acessar o serviço o cliente, que no nosso caso especifico será a aplicação Livraria, deverá invocar o método *autorizaVenda(**int[]** idProd, **int[]** qtidades, **String** nomeLoja)* definido pela *interface Fornecedor* e implementado pela classe *FornecedorSoapBindingImpl.* Ao invocar este método o cliente deverá passar como parâmetro um *array* com os identificadores dos produtos, outro array com as respectivas quantidades dos produtos desejadas e nome que o identifica perante o fornecedor de produtos. Em resposta à requisição *autorizaVenda*, o cliente receberá o valor *booleano true,* caso todos os produtos estejam

disponíveis na quantidade desejada, ou o valor *booleano false* caso algum dos produtos esteja faltando.

A *Figura 16* apresenta o trecho de código no qual o método *autorizaVenda* é implementado. O funcionamento é basicamente o mesmo do método *autorizaCompra()* implementado pela classe *OperadoraSoapBindingImpl*, explicado no item anterior. Portanto, como explicado anteriormente, primeiramente o método *getConnection()* é invocado e retorna um objeto do tipo XAResource. Este na seqüência invoca o método *autorizaVenda()*, implementado pela classe *FornecedorDBMS.* Este método implementa a lógica propriamente dita do serviço *autorizaCompra,* ou seja, ele busca em seus registros os produtos informados como parâmetro na sua chamada. Verifica se a quantidade desejada pelo cliente é inferior ou igual à que ele possui em seu estoque e, em caso afirmativo, efetua uma baixa em seu estoque equivalente à quantidade solicitada, registra o pedido para o cliente para posterior entrega, e retorna ao cliente uma resposta positiva. Caso contrário ele retorna uma resposta negativa.

```
public boolean autorizaVenda(int[] idProd, int[] qtdades, String nomeLoja)
throws RemoteException {

        try {
            FornecedorDBMS conn = getConnection();
            boolean autorizacao = conn.autorizaVenda(idProd, qtdades, nomeLoja);
                closeConnection(conn);
                return autorizacao;
            } catch (Exception e) {
                throw new RemoteException(e.getMessage());
            }
}

private FornecedorDBMS getConnection() throws IllegalStateException, RollbackException,
SystemException {
    FornecedorDBMS conn = FornecedorDBMS.getInstance();
    tm.getTransaction().enlistResource(conn);
    return conn;
}

private void closeConnection(FornecedorDBMS conn)throws IllegalStateException,
SystemException {
    tm.getTransaction().delistResource(conn, XAResource.TMSUSPEND);
}
```

Figura 16 – Trecho de código da classe FornecedorSoapBindingImpl.

4.3 Aplicação Livraria

A *Figura 17* apresenta o diagrama de classes da aplicação Livraria. Como nos demais diagramas, o métodos *setters* e *getters* não estão representados no diagrama, no entanto deve-se considerar que todos os atributos apresentados no diagrama possuem os seus respectivos métodos *set* e *get*.

Esta aplicação é composta por quatro *packages* (pacotes). Os *packages livrariaEntidade* e *livrariaControle* são compostos pelas classes que implementam a lógica da aplicação Livraria propriamente dita. Já os *packages operadoraCartao* e *fornecedor* são compostos pelas classes geradas através da ferramenta *WSDLtoJava,* oferecida pelo *framework* Apache Axis, a partir dos arquivos WSDL do serviço web disponibilizado pela Aplicação Operadora de Cartões de Crédito e do serviço web disponibilizado pela Aplicação Fornecedor de Produtos, respectivamente.

O pacote *livrariaEntidade* é composto exclusivamente por classes que representam as entidades que fazem parte do domínio do problema da Livraria. Estas classes possuem somente os atributos, apresentados na *Figura 17*, e os respectivos métodos *get* e *set.*

As classes *GerenciadorClientes, GerenciadorProdutos, GerenciadorEnderecos, GerenciadorPagamentos* e *GerenciadorVendas*, que compõem o pacote *livrariaControle*, são classes de controle que gerenciam respectivamente as entidades *Cliente, Produto, Endereço, Pagamento* e *Venda*. São elas que controlam a criação, inserção, manipulação e remoção destes objetos durante a execução da aplicação.

Conforme citado anteriormente, a Aplicação Livraria acessará o serviço web *autorizaCompra*, disponibilizado pela aplicação descrita no item 4.1 deste trabalho, a fim de efetuar a cobrança da compra efetuada pelo seu cliente através

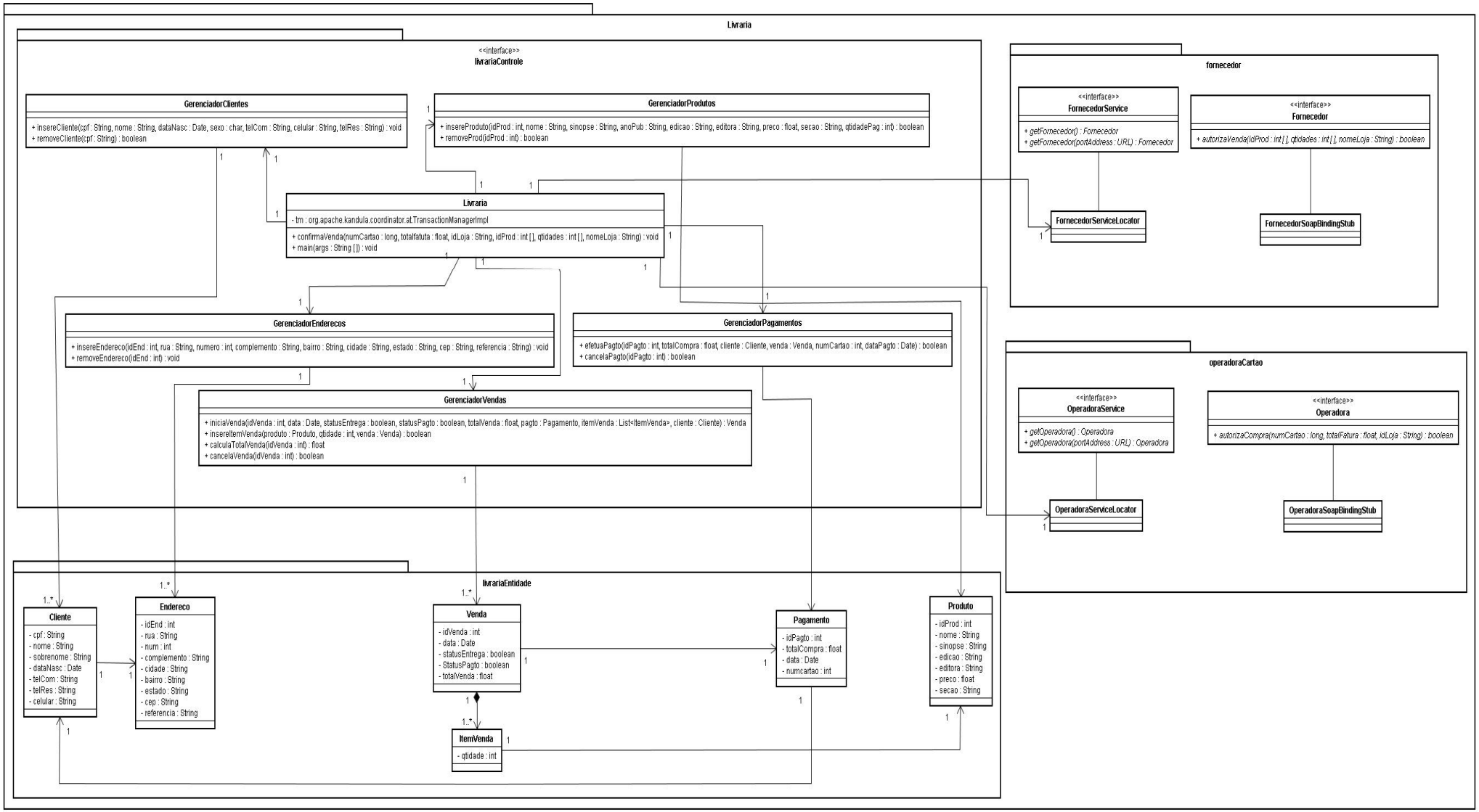

Figura 17 – Diagrama de classes da aplicação Livraria.

do seu cartão de crédito. No entanto, por se tratar de um serviço remoto, situado na Web, faz-se necessário a utilização de um mecanismo RPC (chamada remota de procedimentos) para o seu acesso. Desse modo, utilizaremos o mecanismo *Java RMI* (Remote Method Invocation), o qual é implementado pelas classes que compõem o pacote *operadoraCartao.*

Dentre os componentes do pacote *operadoraCartao* temos a *interface Operadora* que estende *java.rmi.Remote*, o que a caracteriza como uma interface remota que tem a finalidade de possibilitar o acesso ao serviço remoto. A classe *OperadoraSoapBindingStub*, que implementa a *interface Operadora,* consiste na implementação de um objeto *proxy,* e foi gerada através da ferramenta *WSDLtoJava*, mencionada anteriormente, e á responsável por realizar todas as tarefas necessárias para viabilizar a comunicação na rede.

A aplicação Livraria, dentro da sua lógica de funcionamento, também acessará o serviço web *autorizaVenda,* disponibilizado pela aplicação descrita no item 4.2 deste trabalho, a fim de verificar se há disponibilidade dos produtos requeridos pelo cliente, durante a sua compra, no seu fornecedor. O acesso a este serviço será realizado da mesma forma que será feito o acesso ao serviço *autorizaCompra*, explicado acima. O pacote *fornecedor* contém as classes que possibilitam o acesso a este serviço.

A Livraria funcionará do seguinte modo: qualquer usuário poderá acessar os produtos disponíveis e consultar seus respectivos preços. Para realizar uma compra o usuário deverá realizar um cadastro, no qual informará seus dados pessoais, *login* e senha, a fim de que possa ser identificado pelo sistema, e posteriormente logar-se no sistema. Ao realizar o cadastro, o usuário será mantido no sistema como um cliente da Livraria. Após efetuar seu *login* no sistema, o cliente poderá efetuar um

pedido de compra, informando ao sistema os produtos desejados e as respectivas quantidades. Quando o cliente finaliza sua compra, o sistema precisa processá-la a fim de encaminhar os produtos pedidos para a empresa encarregada de realizar a entrega dos mesmos ao cliente em questão.

Para processar a compra realizada pelo seu cliente, a aplicação Livraria precisará acessar tanto o serviço *autorizaVenda* da Aplicação Fornecedor de Produtos a fim de verificar se a mesma possui disponível em seu estoque os produtos e respectivas quantidades solicitadas pelo cliente; quanto o serviço *autorizaCompra* da Aplicação Operadora de Cartões de Créditos a fim de garantir que o pagamento pela realização da venda será feito pelo referido cliente através do seu cartão de crédito. Este processamento é implementado pelo método *confirmaVenda*, *Figura 18*, da classe Livraria, que compõe o pacote *LivrariaControle* e detém o controle inicial da aplicação Livraria.

Note que o ato de efetuar uma venda envolve a execução de transações distribuídas em serviços web distintos, as quais deverão ser efetivadas de forma atômica, ou seja, ou as duas são efetivadas ou ambas são desfeitas. Este controle deverá ser feito pela aplicação Livraria, a qual deverá utilizar algum recurso que coordene estas transações a fim de alcançar a atomicidade desejada.

Conforme visto anteriormente, os protocolos *WS-Coordination e WS-AtomicTransaction* especificam uma forma de gerenciar estas transações a fim de garantir a consistência dos dados envolvidos e consequentemente o provimento de um sistema confiável. Na aplicação Livraria será utilizada a implementação destes protocolos, proposta pelo projeto Apache Kandula, a fim de garantir o gerenciamento adequado das transações.

No inicio deste capítulo foi abordado o funcionamento da arquitetura Apache Kandula e mencionado que o serviço de coordenação é feito através de serviços

web. A *Figura 19* apresenta as classes e seus relacionamentos que compõem o serviço de coordenação de transações.

```
public void confirmaVenda(long numeroCartao, float totalFatura, String idLoja,
        int[] idProd, int[] qtdades, String nomeLoja){

    boolean autorizaCompra;
    boolean autorizaVenda;

    TransactionManagerImpl tm = TransactionManagerImpl.getInstance();

    OperadoraService operadora = new OperadoraServiceLocator();
    Operadora op;
    try {
        op = operadora.getOperadora();

        FornecedorService fornecedor = new FornecedorServiceLocator();
        Fornecedor forn = fornecedor.getFornecedorSoapBindingImpl();

        tm.begin();
        autorizaCompra = op.autorizaCompra(numeroCartao, totalFatura, idLoja);
        autorizaVenda = forn.autorizaVenda(idProd, qtdades, nomeLoja);

        if (!autorizaCompra || !autorizaVenda){
            tm.rollback();
        }else{
            tm.commit();
        }
    }catch (RemoteException e1) {
        e1.printStackTrace();
        try {
            tm.rollback();
        } catch (RemoteException e2) {
            e1.printStackTrace();
        }

     } catch (ServiceException e1) {
         e1.printStackTrace();
         try {
             tm.rollback();
         } catch (RemoteException e2) {
             e1.printStackTrace();
         }
     }catch (Exception e) {
         e.printStackTrace();
         try {
             tm.rollback();
         } catch (RemoteException e1) {
             e1.printStackTrace();
         }
     }
}
```

Figura 18 – Trecho de código da classe Livraria.

Neste serviço, todas as instâncias coordenadoras devem implementar a *interface WSCoordinator.* Esta *interface* define os requisitos básicos esperados de qualquer tipo de coordenador, como por exemplo, criação de um contexto de coordenação, registro de participantes, identificação das atividades por ele coordenadas, etc. As *Interfaces* de coordenadores de tipos específicos, como por exemplo, *WSAtomicTransactionCoodinator* estendem a *interface WSCoordinator* e definem os requisitos particulares ao seu tipo de coordenação. (KANDULA, 2007B).

WSCoordinatorImpl é uma classe abstrata que também implementa *WSCoordinator* e fornece funções tais como *locking* (bloqueio) que são requeridas por todos os tipos de coordenadores. Todas as classes concretas que implementam um tipo de coordenador especifico, como por exemplo *WSAtomicTransactionCoordinationImpl,* ,implementam esta classe. Ela funciona como um adaptador par as subclasses.

Segundo Kandula (2007B), a classe *CoordinatorService* é a responsável pela criação de um coordenador apropriado para as atividades novas ou importadas. É ela também que permite que transações locais (JTA) sejam importadas para dentro do serviço de coordenação e seja coordenada como uma atividade. Para cada atividade do tipo transação atômica, o serviço de coordenação cria uma nova instância da classe *WSAtomicTransactionCoordinationImpl.* Para cada transação local importada ele cria uma instância do tipo *SurrogateWSAtomicTransactionCoordinationImpl.*

O *SurrogateWSAtomicTransactionCoordinationImpl* registra-se como um participante da transação local passada como parâmetro no seu construtor e também implementa a *interface XAResource,* do JTA. Quando uma transação local é confirmada (*commited*) ou desfeita (*rolled back*) a classe simplesmente invoca o método correspondente da super classe *WSAtomicTransactionCoordinationImpl* de

dentro da implementação *XAResourse* para poder estender o protocolo *commit* de duas fases feito sobre os recursos locais para os participantes da transação atômica. A classe sobrescreve os métodos *commit* e *rollback* da classe *WSAtomicTransactionCoordinationImpl.* Os métodos sobrescritos simplesmente invocam os métodos *commit* e *rollback* da transação local para serem executados.

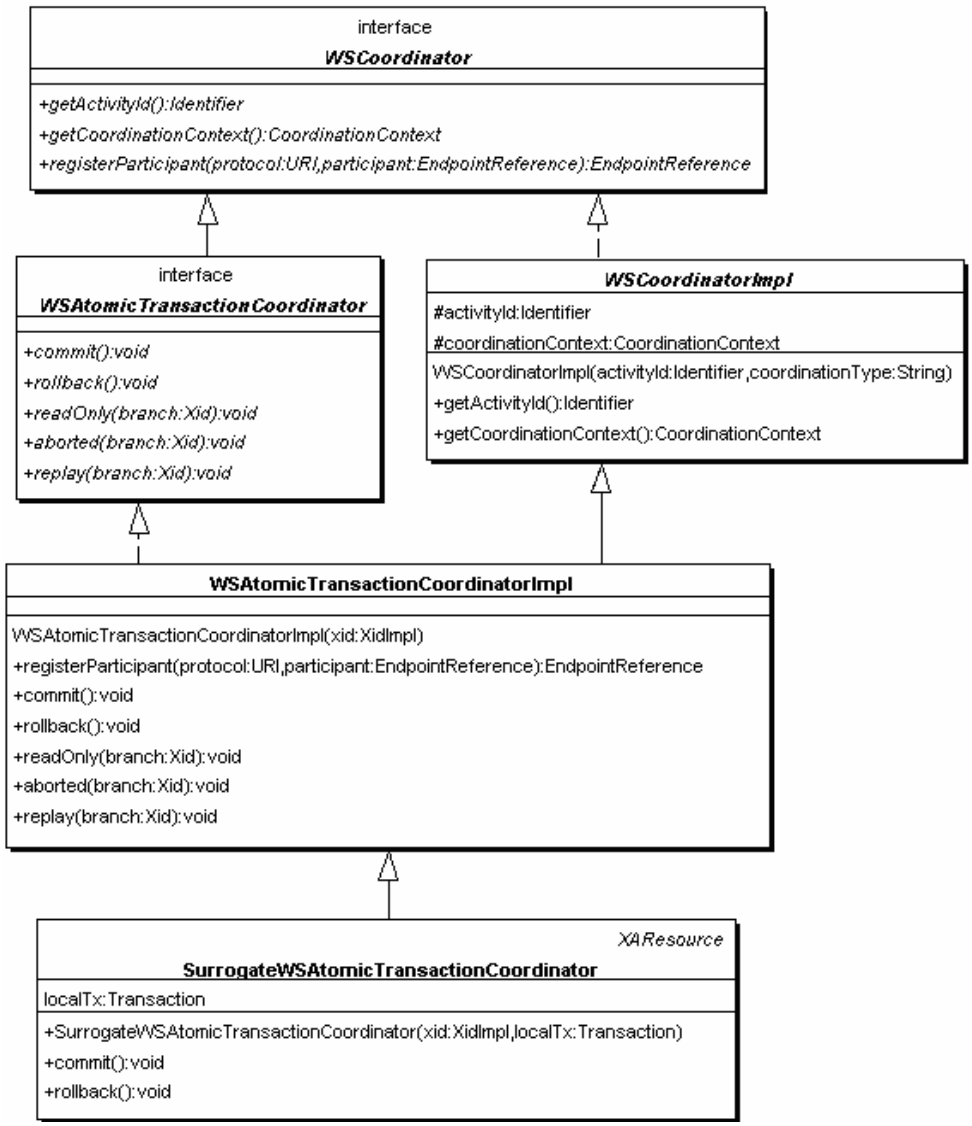

Figura 19 – Diagrama de Classes do Serviço de Coordenação . (KANDULA, 2007B).

A classe *CoordinatorService* é um *singleton.* Além da criação de novos coordenadores, ela também é utilizada para encaminhar as requisições para os respectivos coordenadores através de *endpoints* de serviço diferentes. A classe matém um *hash map* com todos os coordenadores de atividades ativos, identificados pelo seu identificador de atividade. Quando uma nova requisição é

recebida, o *endpoint* utiliza as propriedades de referência para localizar o identificador de atividade correspondente à requisição recebida. Então o método *getCoordinator*, da classe *CoordinatorService*, é utilizado para obter uma referência para o coordenador da atividade.

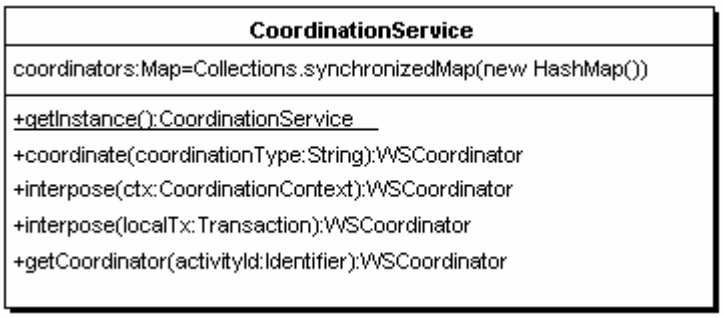

Figura 20 – Classe CoordinatorService. (KANDULA, 2007B).

A *Figura 21* apresenta o diagrama de seqüência que demonstra o funcionamento do gerenciamento global das transações distribuídas envolvidas no cenário proposto nesse trabalho. Como podemos observar, a Livraria assume o papel de cliente dos serviços de coordenação (*autorizaVenda* e *autorizaCompra)* e possui um gerenciador de transações, representado pelo objeto do tipo *org.apache.kandula.coordinator.at.TransactionManagerImpl,* o qual possui como função demarcar as fronteiras transacionais onde as operações afetam o contexto transacional das *threads* chamadas, conforme já abordado no item 4.1 deste trabalho. Esta demarcação é iniciada pelo método *begin()*.

De acordo com o diagrama de seqüência, um objeto *TransactionManager* é instanciado e excuta o método *begin()*, o qual por sua vez acessa o serviço de coordenação e cria um contexto de coordenação através do *endpoint* de ativação. Após a criação do contexto, ele registra-se acessando o *endpoint* de Registro do serviço de coordenação. Na seqüência, o serviço *autorizaVenda* é acessado através do seu *endpoint* pela Livraria. Nesse momento, o controle de transacional local implementado na aplicação Fornecedor, abordado no item 4.2 deste trabalho, é

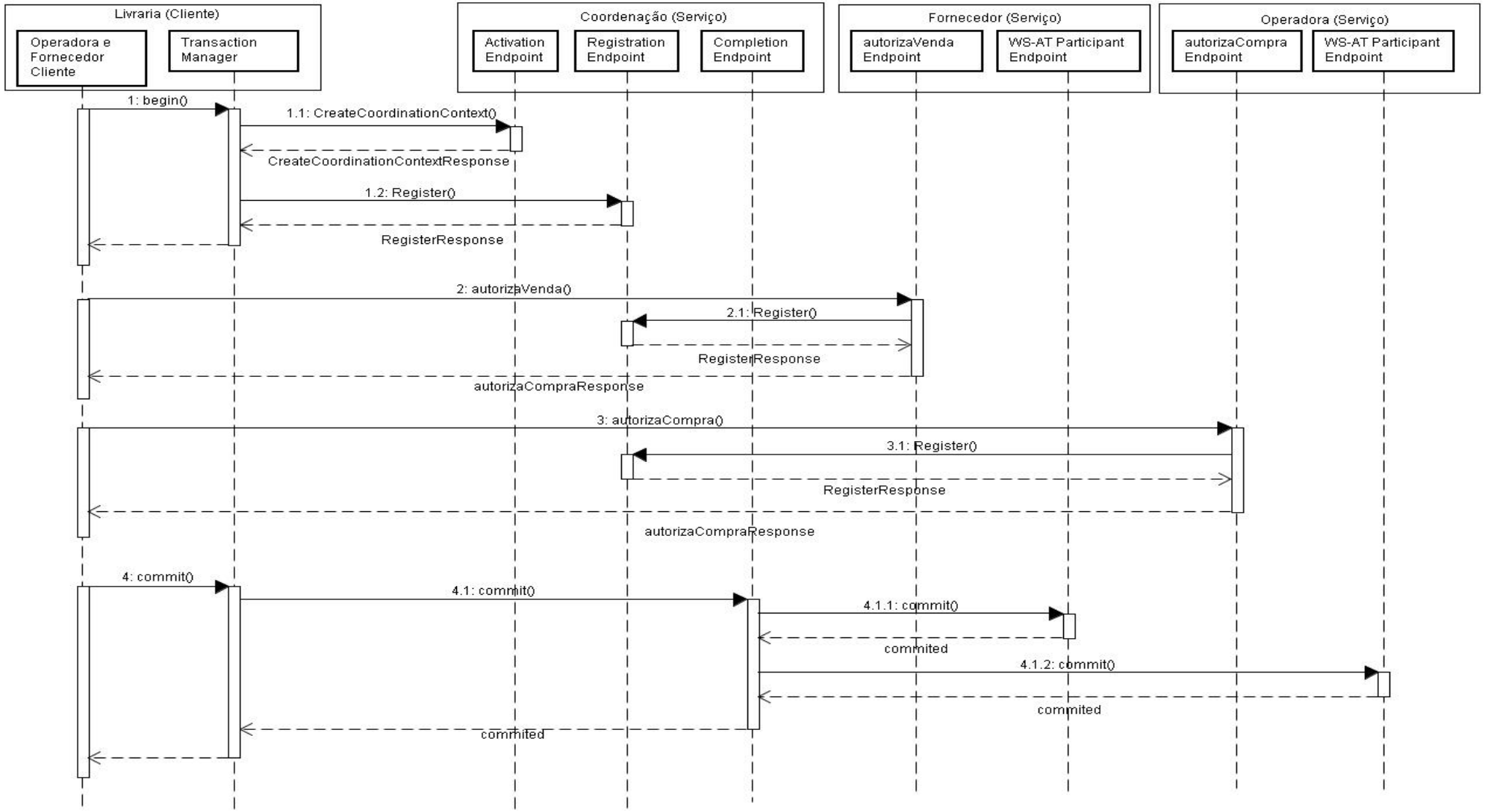

Figura 21 – Diagrama de Seqüência do gerenciamento de transações global no caso de efetivação de todas as transações locais envolvidas .

executado. Durante esta execução, ocorre o registro dessa transação como participante da transação global, através do *endpoint* de registro do serviço de coordenação. Após o registro da transação, o serviço *autorizaVenda* propriamente dito é executado, e após sua execução retorna como resposta para a aplicação Livraria se possui ou não em seu estoque os produtos e as respectivas quantidades desejadas.

Na seqüência o serviço *autorizaCompra* é acessado através do seu *endpoint* pela Livraria, e ocorre o mesmo fluxo de execução ocorrido no acesso ao serviço *autorizaVenda.* Se a resposta obtida no acesso a ambos os serviços forem positivas, a aplicação Livraria informa ao objeto *TransactionManager* que as transações participantes devem ser efetivadas, ou seja, o método *commit()* é invocado. Caso uma das respostas seja negativa, o objeto *TransactionManager* é informado que ambas as transações participantes devem ser desfeitas, ou seja, o método *rollback* deve ser invocado.

Em ambas as situações, quando os métodos *commit* ou *rollback* da instância *TransactionManager* são invocados, o *endpoint Completion* do serviço de coordenação é acessado e ele invoca o método correspondente ao *commit* ou *rollback* implementado pelo controle de transações local da aplicação Fornecedor e da aplicação Operadora, garantindo assim a consistência dos dados envolvidos na transação global ‘confirma venda’ da aplicação Livraria. O diagrama de seqüência apresentado na *Figura 21* mostra apenas o fluxo de execução no caso em que as transações participantes devem ser efetivadas (*commited*). Um diagrama muito semelhante representaria o fluxo de execução no caso em que as transações precisassem ser desfeitas. A única coisa que mudaria é que a invocação de todos os métodos *commit* seria substituída pela invocação do método *rollback*.

A implementação do controle transacional global, descrito acima, realizado pela Livraria é representado pelo método *confirmaVenda,* demonstrado na *Figura 18*. Como podemos observar, a implementação desse controle global torna-se muito simples com a utilização do serviço de coordenação implementado pelo projeto Kandula. Ela resume-se a instanciar um objeto do tipo *org.apache.kandula.coordinator.at.TransactionManagerImpl*, invocar o seu método *begin,* depois invocar os serviços desejados, nesse caso *autorizaVenda* e *autorizaCompra,* e invocar o método *commit* ou *rollback* dependendo das respostas dos serviços invocados. Toda a complexidade do gerenciamento global das transações distribuídas dos serviços web é encapsulada pelo serviço de coordenação do projeto Kandula.

Como pode ser observado na *Figura 18*, ambos os serviços *autorizaVenda* e *autorizaCompra* são invocados e suas respostas armazenadas em variáveis *booleanas*. Na seqüência, suas respostas são verificadas, e caso ambas sejam afirmativas (*true*) as transações locais realizadas tanto na Operadora de Cartões de Credito como no Fornecedor de Produtos são efetivadas. Caso uma delas seja negativa (*false*), ambas são desfeitas. Isso garante a consistência dos dados e consequentemente um sistema confiável. Note também que caso ocorra alguma exceção, como por exemplo, de conexão, durante o acesso a estes serviços, ambas as transações são desfeitas.

5 Conclusão e Trabalhos Futuros

5.1 Conclusão

A importância do gerenciamento transacional em um sistema computacional é incontestável e, portanto, qualquer sistema deve realizar este gerenciamento de forma eficaz a fim de evitar que a sua execução cause inconsistências nos dados manipulados por ele. No decorrer deste trabalho vimos que protocolos foram especificados e implementados a fim de proporcionar um gerenciamento de transações de forma eficaz e eficiente. No entanto, estes protocolos foram desenvolvidos com o intuito de satisfazerem as necessidades de transações tradicionais.

Com o surgimento dos Web Services, que se tornam cada vez mais populares devido as suas atrativas características, bem como interoperabilidade, flexibilidade, redução de custos operacionais e de desenvolvimento, reutilização de código e melhora no relacionamento entre parceiros e clientes, surgiu também a necessidade da especificação e implementação de novos protocolos de gerenciamento de transações. Isto se deve ao fato de que as transações realizadas em serviços web possuem características diferentes das transações tradicionais, como por exemplo, a possibilidade de transpor os limites organizacionais e o seu tempo de duração, que em alguns casos pode levar dias, dentre outros.

Neste trabalho foram apresentadas algumas soluções já propostas para o gerenciamento de transações distribuídas em web services. Dentre elas a implementação realizada pelo Apache, através do projeto Kandula, dos protocolos *WS-Coordination* e *WS-AtomicTransaction*, qual foi utilizada para gerenciar

transações atômicas de serviços web distribuídos utilizados por um sistema de Livraria Virtual.

A utilização da implementação proposta pelo projeto Kandula nos permite realizar o gerenciamento global de transações distribuídas de forma bastante simplificada, uma vez que toda a complexidade deste gerenciamento é encapsulada pelo serviço de coordenação disponibilizado pelo projeto. No entanto, o projeto Kandula, por ainda estar em fase de desenvolvimento, é bastante instável e possui algumas deficiências, listadas a seguir, que dificultam a sua utilização.

- Quase total falta de documentação e suporte na distribuição do Kandula e a total falta de documentação na distribuição Kandula 2. Para poder utilizar a implementação proposta pelo projeto foi necessário analisar o seu código fonte;
- Fornecimento de informações contraditórias dependendo da parte do site do projeto acessada. Por exemplo, em determinados locais é informado que o projeto implementa o protocolo *WS-BusinnessActivity* e em outro local já informa que atualmente não há suporte para este protocolo, o que confunde o usuário desenvolvedor.

Um outro fator negativo da implementação do projeto Kandula é o fato de que atualmente ela suporta somente o controle de transações local JTA implementado Apache Geronimo, o que acarreta na perda de interoperabilidade entre sistemas heterogêneos, uma importante vantagem dos Web Services. Isto implica que, para utilizar o Kandula, todos os serviços web terão que ser implementados em Java e utilizar o controle de transações implementado pelo apache Geronimo.

5.2 Trabalhos Futuros

Como possíveis trabalhos futuros, pode-se apontar:

- Uma extensão do cenário proposto com a inserção de um outro serviço web que envolva transações de longa duração que poderão ser coordenadas pelo protocolo Ws-B*usinnesActivity* .
- Migração das tecnologias utilizadas para implementação do cenário proposto neste trabalho do Axis para o Axis2 e consequentemente do Kandula para Kandula2.
- Experimentar outras implementações no controle de transações distribuídas e fazer um comparativo com a experiência realizada neste trabalho com a implementação proposta pelo Apache kandula.
- Devido a grande dificuldade encontrada na utilização da infra-estrutura proposta pelo projeto Apache Kandula seria interessante que fosse proposto uma maneira mais simples de se utilizar os protocolos WS-C e WS-Tx no controle de transações distribuídas em WS. Uma saída seria o desenvolvimento de um framework.

6 Referências

[Albertin, 2000] ALBERTIN, Alberto Luiz. **Comércio eletrônico: modelo, aspectos e contribuições de sua aplicação**. 2 ed. São Paulo: Atlas, 2000. 242p.

[Bell, Grimson, 1992] BELL, David; GRIMSON, Jane. **Distributed Database Systems,** England:Addison-Wesley Publishers, 1992. 410p.

[Belli, 2005] BELLI, Djan. Transações Distribuídas. Florianópolis, 10 p. Trabalho não publicado.

[Bray et al., 2006] BRAY, Tim; PAOLI, Jean; SPERBERG-MCQUEEN, C. M.; MALER, Eve; YERGEAU, François. **Extensible Markup Language (XML) 1.0 (Fourth Edition)**. Apresenta a especificação da W3C para a tecnologia XML. Ago. 2006. Disponível em: <http://www.w3.org/TR/2006/REC-xml-20060816/>. Acesso em: 16 nov. 2006.

[Cabrera et al., 2005] CABRERA, Luis Felipe; COPELAND, George; FEINGOLD, Max; FREUND, Robert W; FREUND, Tom; JOYCE, Sean; KLEIN, Johannes; LANGWORTHY, LITTLE, Mark; LEYMANN, Frank; NEWCOMER, Eric; ORCHARD, David; ROBINSON, Ian; STOREY, Tony, THATTE, Satish. ***Web Services Business Activity Framework (WS-BusinessActivity)***, 2005. Disponível em: < ftp://www6.software.ibm.com/software/developer/library/WS-BusinessActivity.pdf>. Acesso em 10 Dez. 2006.

[Cheung, Matena, 2007] CHEUNG, Susan; MATENA, Vlada. **Java Transaction API (JTA)**, 2007. Disponível em: < http://sdlc-esd.sun.com/ESD31/JSCDL/jta/1.1/jta-1_1-spec.pdf?AuthParam=1181576773_733437da8a50b97dd8f07947c1b0feaa&TUrl=an1npDpbKod7kSYrROhENTonIec1W0D1Lc4nXz+pGFFranixdCdgxDTPbW4=&TicketId=dVF4NgBGMOo+8w==&GroupName=SDLC&BHost=sdlc6g.sun.com&FilePath=/ESD31/JSCDL/jta/1.1/jta-1_1-spec.pdf&File=jta-1_1-spec.pdf>. Acesso em 10 maio 2007.

[Carvalho, 2000] CARVALHO, Isabela Stanzio Lessa de. **Um Estudo sobre Modelos de Transações para a Internet**, 2000. 164f. Dissertação de

Mestrado (Ciências da Computação) – Universidade federal de Santa Catarina, Florianópolis, 2005.

[Chappell, Jewell, 2002] CHAPPELL, David A.; JEWELL, Tyler. **Java Web Services**. United States: O'Reilly, 2002. 262p.

[Christensent et al, 2001] CHRISTENSENT, Erik; CURBERA, Francisco; MEREDITH, Greg; WEERAWARANA, Sanjiva. **Web Services Description Language (WSDL) 1.1.** Apresenta a especificação da W3C para a tecnologia WSDL. Mar. 2001. Disponível em: <http://www.w3.org/TR/wsdl>. Acesso em: 15 out. 2006.

[Clement et al., 2004] CLEMENT, Luc; HATELY, Andrew; RIEGEN, Claus von; ROGERS, Tony. **UDDI Version 3.0.2.** Apresenta a especificação da OASIS para a tecnologia UDDI. Out. 2004. Disponível em: <http://uddi.org/pubs/uddi_v3.htm>. Acesso em: 15 out. 2006.

[Cox et al., 2004] COX, CABRERA, Felipe; COPELAND, George; FREUND, Tom; KLEIN, Johannes; *STOREY,* Tony; THATTE, Satish . ***Web Services Transaction (WS-Transaction),*** 2004. Disponível em: < http://dev2dev.bea.com/pub/a/2004/01/ws-transaction.html>. Acesso em: 19 Ago. 2006.

[DatAware, 2006] DatAware. **Serviço de transação**. Disponível em: http://dataware.nce.ufrj.br:8080/dataware/fisico/cursos/middleware/slides/transacao.pdf. Acesso em: 11 de maio de 2007.

[Elsmari, Navathe, 2000] ELSMARI. Ramez; NAVATHE, Shamkant B. **Sistema de Banco de Dados – Fundamentos e Aplicações.** Rio de Janeiro: LTC Editora, 2000. 795p.

[Haas, Brown, 2004] HAAS, Hugo; BROWN, Allen. **W3C Glossary.** W3C Working Group Note. Fev. 2004. Disponível em: <http://www.w3.org/TR/ws-gloss/>. Acesso em: 15 out. 2006.

[Jajodia, Kerschberg, 1997] JAJODIA, Sushil; KÉRSCHBERG, Larry. **Advanced Transaction Model and Architectures.** London: Kluwer Academic Publishers, 1997. 400p.

[Kandula, 2007a] KANDULA. **Apache Kandula.** Disponível em < http://ws.apache.org/kandula/>. Acesso em: 9 maio 2006.

[Kandula, 2007b] KANDULA. **Architetural Design.** Disponível em < http://ws.apache.org/kandula/1/architecture-guide.html>. Acesso em: 22 maio 2007.

[Leandro, 2005] LEANDRO, Sabrina da Silva. **Balanceamento de Carga em Web Services**, 2005. 81 f. Trabalho de Conclusão de Curso (Bacharelado em Sistemas de Informação) – Universidade Federal de Santa Catarina, Florianópolis, 2005.

[Lima, 2003] LIMA, Rodrigo Pedroso. **Análise Estatística do Desempenho de Algoritmos de Replicação na Execução da Operação *WRITE***, 2003. 88 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) – Universidade de Passo Fundo, Passo Fundo, 2003.

[Limthanmaphon, Zhang, 2004] LIMTHANMAPHON, Benchaphon; ZHANG, Yanchun. Web Service Composition Transaction Management. In: Fifteenth Australasian Database Conference (ADC2004), 50, 2003. **Research and Practice in Information Technology.** Dunedin: Australian Computer Society, 2003. V. 27. p. 171-179.

[Little, Freund, 2003] LITTLE, Mark; FREUND, Thomas; **A comparison of Web services transaction protocols** - A comparative analysis of WS-C/WS-Tx and OASIS BTP. Disponível em: <http://www-128.ibm.com/developerworks/webservices/library/ws-comproto/>. Acesso em: 05 Dez. 2006.

[Mitra, 2003] MITRA, Nilo. **SOAP Version 1.2 Part 0: Primer.** Apresenta a especificação da W3C para a tecnologia SOAP. Jun. 2003. Disponível em: < http://www.w3.org/TR/2003/REC-soap12-part0-20030624/>. Acesso em: 16 nov. 2006.

[Muschamp, 2004] MUSCHAMP, Paul. **An Introduction to Web Services**. In: BT Technology Journal, Vol. 22, No. 1, Janeiro 2004.

[Newcomer, 2002] NEWCOMER, Eric. **Understanding Web Services – XML, WSDL, SOAP, and UDDI.** United States: Addison-Wesley, 2002. 332p.

[Orchard et al., 2005] ORCHARD, David; CABRERA, Felipe; COPELAND, George; FREUND, Tom; KLEIN, Johannes; LANGWORTHY, David; SHEWCHUK, John; *STOREY,* Tony . ***Web Services Coordination (WS-Coordination),*** 2005. Disponível em: < ftp://www6.software.ibm.com/software/developer/library/WS-Coordination.pdf>. Acesso em: 07 Jul. 2006.

[Papazoglou, 2003] PAPAZOGLOU, Mike P. Service-Oriented Computing: Concepts, Characteristics and Directions. In: Web Information Systems Engineering Workshops, 2003 (WISE' 03), 04, 2003. **Proceedings Fourth International Conference on Web Information Systems Engineering.** IEEE, 2003. p. 3-12.

[Silberschatz, Korth, Sudarshan, 2006] SILBERSCHATZ, Abraham; KORTH, Henry F.; SUDARSHAN, S. **Sistema de Banco de Dados.** São Paulo: Elsevier Editora, 2006. 781p.

[Singh, et al., 2004] SINGH, Inderjeet; BRYDON, Sean; MURRAY, Greg; RAMACHANDRAM, Vijay, VIOLLEAU, Thierry; STEARNS, Beth. **Designing Web Services with the J2EE 1.4 Plataform JAX-RPC, SOAP, and XML Technologies.** United States: Addison-Wesley, 2004. 418p.

[Vaughan-Nichols, 2002] VAUGHAN-NICHOLS, Steven J. Web Services: Beyond the Hype. **Computer Innovative Technology for Computing Professionals**, v. 35, n. 2, p. 18-21, fev. 2002.

[W3C, 2006] W3C. **Web Services Activity**. Disponível em: <http://www.w3.org/2002/ws/>. Acesso em: 02 nov. 2006.

7 APÊNDICES

7.1 APÊNDICE A – Lista de Classes Utilizadas no Cenário de Aplicação Proposto

7.1.1 Classes Implementadas neste Trabalho

7.1.1.1 Aplicação Livraria

- Livraria.java
- GerenciadorClientes.java
- GerenciadorProdutos.java
- GerenciadorEnderecos.java
- GerenciadorPagamentos.java
- GerenciadorVendas.java
- Cliente.java
- Endereço.java
- Venda.java
- ItemVenda.java
- Pagamento.java
- Produto.java
- FornecedorService.java
- Fornecedor.java
- FornecedorServiceLocator.java
- FornecedorSoapBindingStub.java
- OPeradoraService.java
- Operadora.java
- OperadoraServiceLocator.java
- OperadoraSoapBindingStub.java

7.1.1.2 Aplicação Operadora de Cartões de Crédito

- Operadora.java
- OpearadoraSoapBindingImpl.java
- GerenciadorCartoes.java
- GerenciadorClientes.java
- Cliente.java
- Endereco.java
- Cartao.java
- Fatura.java
- Itemfatura.java
- OperadoraDBMS.java

7.1.1.3 Aplicação Fornecedor de produtos

- Fornecedor.java
- FornecedorSoapBindingImpl.java
- GerenciadorEstoque.java
- GerenciadorPedidos.java
- GerenciadorProdutos.java
- ItemEstoque.java
- Pedido.java
- ItemPedido.java
- Produto.java
- FornecedorDBMS.java

7.1.2 Classes do Apache Kandula Utilizadas na Implementação

- org.apache.kandula.geronimo.Bridge
- org.apache.kandula.coordinator.at.TransactionImpl
- org.apache.kandula.coordinator.at.TransactionManagerImpl
- org.apache.kandula.coordinator.ActivationStub
- org.apache.kandula.coordinator.CoordinationContext
- org.apache.kandula.coordinator.CoordinationService

- org.apache.kandula.wscoor.CoordinationContextType
- org.apache.kandula.wscoor.CoordinationContextTypeIdentifier
- org.apache.kandula.utils.AddressingHeaders
- org.apache.kandula.utils.Service
- org.apache.kandula.utils.TCPSnifferHelper
- org.apache.kandula.wscoor.ActivationPortTypeRPCBindingStub
- org.apache.kandula.wscoor.CreateCoordinationContextResponseType
- org.apache.kandula.wscoor.CreateCoordinationContextType
- org.apache.kandula.coordinator.CoordinationContext
- org.apache.kandula.wscoor.Expires

7.1.3 Classes da Implementação da API JTA feita pelo Apache Geronimo Utilizadas na Implementação

- org.apache.geronimo.transaction.manager.NamedXAResource
- org.apache.geronimo.transaction.manager.TransactionManagerImpl
- org.apache.geronimo.transaction.manager.XidFactoryImpl

7.1.4 Classes da API JTA Utilizadas na Implementação

- javax.transaction.xa.XAException
- javax.transaction.xa.XAResource
- javax.transaction.xa.Xid
- javax.transaction.RollbackException
- javax.transaction.Transaction
- javax.transaction.TransactionManager
- javax.transaction.UserTransaction

7.2 APÊNDICE B – Artigo

Gerenciamento de Transações Distribuídas em Web Services

Priscilla Francielle Poleza [1]

[1]Departamento de informática e Estatística
Universidade Federal de Santa Catarina (UFSC) – Florianópolis, SC – Brasil.
prifp@inf.ufsc.br

Abstract. *Web services architecture was created to facilitate the interconnection and the interaction of heterogeneous systems. Web services are based on standard technologies, particularly XML and http, are weakly connected and guarantee internal communication operations among applications; that is, web services are independent of the operational system and the language programming. However, because it is such a new technology, certain aspects still need to be improved, for example, the achievement of the control of distributed transactions. This monograph entails the development of a research about the working and the use of the implementation of WS-Coordination and WS-Atomic Transaction protocols proposed by the Apache Kandula project; the research contemplates such use within a specific application scenario, in order to demonstrate how to control distributed transactions in a way that may ensure the consistency of the data of the relevant application. For that purpose, an application scenario was defined and implemented, in which atomic transactions take place between one application and two web services. The co-ordination service (implemented by Kandula) was used to co-ordenate the distributed transactions carried out within the implemented scenario, and the implementation of JTA (by Apache Geronimo) was used to carry out the local control of transactions by the applications that implement the web services involved in the scenario.*

Resumo. *A arquitetura de Serviços Web surgiu com o intuito de facilitar a interligação/integração de sistemas heterogêneos. Ela é baseada em tecnologias padronizadas, em particular XML e http, é fracamente acoplada e garante a interoperabilidade na comunicação entre aplicações, ou seja, é independente de sistema operacional e de linguagem de programação. Porém, por se tratar de uma tecnologia muito nova alguns aspectos ainda precisam ser melhorados, como por exemplo, a realização do controle das transações distribuídas. Este trabalho consiste na realização de uma pesquisa a respeito do funcionamento e utilização da implementação dos protocolos WS-Coordination e WS-AtomicTransaction, proposta pelo projeto Apache Kandula; e na utilização desta implementação em um cenário de aplicação específico a fim de demonstrar como pode ser feito um controle de transações distribuídas de maneira a garantir a consistência dos dados das aplicações envolvidas. Para isso definiu-se e implementou-se um cenário de aplicação, no qual ocorrem transações atômicas*

entre uma aplicação e dois serviços web. E utilizou-se o serviço de coordenação, implementado pelo Kandula, para coordenar as transações distribuídas realizadas no cenário implementado e a implementação do JTA, realizada pelo Apache Geronimo, para realizar o controle local das transações pelas aplicações que implementam os serviços web envolvidos no cenário.

1. Introdução

Com o intuito de sobreviver e obter sucesso no contexto da economia de mercado atual, as empresas vêm percebendo a necessidade de interligar seus processos de negócio, bem como trocar informações com fornecedores, clientes e parceiros através de processos de negócios externos. No entanto, para que a empresa consiga atender de forma eficaz e eficiente as exigências dos clientes e de estarem atentas às mudanças de mercado e/ou ameaças da concorrência, é necessário que seja possível fazer uma fácil integração entre os diferentes sistemas que implementam os processos utilizados por ela.

A arquitetura de Web Services possibilita a integração de sistemas heterogêneos devido às seguintes características: é baseada em tecnologias padronizadas, em particular XML e HTTP; é fracamente acoplada; e oferece interoperabilidade, ou seja, é independente de sistema operacional e de linguagem de programação. Porém, para que esta tecnologia seja adotada de forma mais efetiva comercialmente é necessário que se tenha uma infra-estrutura confiável para o seu desenvolvimento e implantação, ou seja, é necessário investir ainda na melhoria de alguns pontos.

O requisito confiabilidade é imprescindível na maioria dos sistemas computacionais existentes, como por exemplo, sistemas bancários, sistemas de comércio eletrônico, sistemas eleitorais, sistemas de diagnóstico médico e muitos outros. Várias propriedades e protocolos já foram definidos a fim de garantir que os sistemas sejam confiáveis a nível das transações realizadas. No entanto, a tecnologia de Web Services possui necessidades diferentes das transações tradicionais em virtude das suas características.

Portanto, a fim de garantir que as transações envolvendo Web Services sejam tratadas de forma adequada, um consórcio de várias empresas especificou os protocolos *Web Services Coordination* (WS-C) (Orchard et al. 2005) e *Web Services Transaction* (WS-Tx) (Cox et al. 2004), que visam propor uma infra-estrutura de suporte a transações distribuídas efetuadas por aplicações que utilizam a tecnologia de Web Services.

O principal objetivo deste trabalho consiste no estudo e utilização da implementação dos protocolos *WS-Coordination* e *WS-AtomicTransaction* proposta pelo projeto Kandula em uma aplicação real, a fim de demonstrar a utilização de uma infra-estrutura de suporte a transações distribuídas em um ambiente que utilize a tecnologia de Web Services.

2. Organização do texto

Este artigo está organizado da seguinte maneira. As seções 3, 4, 5 e 6 tratam da revisão bibliográfica, com ênfase em Serviços Web, transações, transações distribuídas e transações em Web Services. Na seção 7, faz-se uma descrição do cenário de aplicação que foi implementado a fim de demonstrar como pode ser feito o gerenciamento de transações distribuídas em Web Servises utilizando-se a implementação dos protocolos WS-Coordination e WS-AtomicTransaction proposta pelo Apache kandula. Bem como as

ferramentas e tecnologias utilizadas no desenvolvimento deste trabalho. Finalmente, as conclusões e os resultados deste trabalho são apontados na seção 8. Nesta seção também são apresentadas sugestões para trabalhos futuros.

3. Web services

Web service pode ser definido como uma peça da lógica do negócio capaz de integrar sistemas e possibilitar a comunicação entre diferentes aplicações, possibilitando desta forma a interação de novas aplicações com aplicações já existentes e a compatibilidade de sistemas desenvolvidos em plataformas diferentes. A interoperabilidade provida pelo Web Service é garantida devido ao fato desta tecnologia permitir que as aplicações enviem e recebam dados no formato XML.

Os Web Services foram desenvolvidos com base na Arquitetura Orientada a Serviços (*Service-oriented Architeture – SOA)*. Como o próprio nome já diz, o modelo SOA baseia-se em serviços, os quais segundo Papazoglou (2003), podem ser definidos como funções de negócios implementadas em software e acessíveis através das suas interfaces. A função da interface é fornecer o mecanismo pelo qual os serviços se comunicam com outros serviços e aplicações e apresentar o conjunto de operações disponíveis para invocação dos clientes do serviço.

O modelo SOA baseia-se na interação entre agentes de software, os quais assumem papéis distintos entre si, tais como: *provider* (provedor), *broker* (mediador ou intermediador) e *requisitor* (consumidor ou invocador). A interação entre os agentes é realizada através de troca e mensagens.

Para que essas interações possam ocorrer, um sistema SOA deve prover os seguintes componentes de arquitetura. (LEANDRO, 2005):

- **Transporte:** este componente representa os formatos e protocolos usados na comunicação com um serviço.
- **Descrição:** este componente representa as linguagens que são utilizadas na descrição de um serviço, provendo as informações necessárias para acessá-lo.
- **Descobrimento:** este componente representa os mecanismos utilizados para registrar, anunciar e encontrar um serviço e sua descrição.

Analisando a *Figura* 21, a qual ilustra a arquitetura Web Service, pode-se observar que os componentes de transporte, descrição e descobrimento, citados anteriormente, são implementados respectivamente pelos padrões SOAP, WSDL, e UDDI. O padrão WSDL é utilizado na **descrição** das interfaces dos serviços web fornecidos pelo provedor de serviço. Através do padrão UDDI é possível realizar tanto a publicação da descrição das interfaces para o mediador (*broker*) de serviços, quanto a **descoberta** dos serviços requisitados pelos consumidores. O padrão UDDI é o responsável também pela especificação do local no qual o serviço requerido pode ser encontrado bem como qual deve ser o formato das mensagens trocadas. Já o padrão SOAP é utilizado no envio e recebimento, ou seja, **transporte** das mensagens trocadas entre o consumidor, mediador e o provedor de serviços.

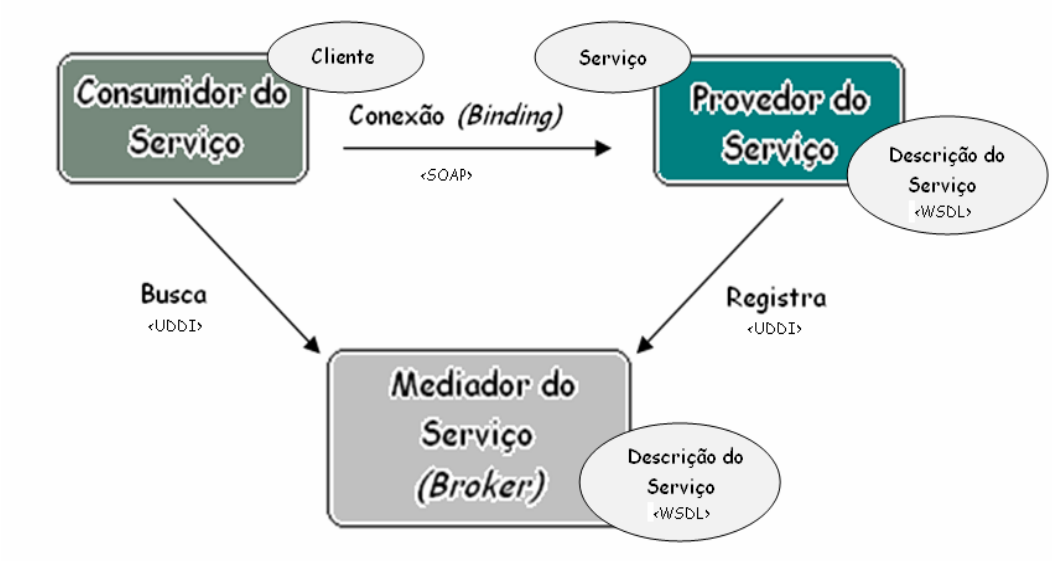

Figura 1. – Arquitetura do Web Service.

4. Transações

Segundo Silberschatz, Korth e Sudarshan (2006), um conjunto de operações que formam uma única unidade lógica de trabalho é chamado de transação. Para que um sistema seja considerado confiável é necessário que ele garanta a execução apropriada de transações, apesar da possibilidade de falhas. Deste modo, evita-se que os dados sejam corrompidos ou tornem-se inconsistentes, garantindo ao usuário uma resposta correta, ou ao menos uma mensagem de indicação de erro do sistema.

O conceito de transações atômicas surgiu da necessidade de que, mesmo nos casos em que falhas ocorram, a consistência das aplicações deve ser garantida. Uma transação atômica pode ser definida como uma transação indivisível, ou seja, ou ela é totalmente executada ou nenhuma de suas operações são efetivadas. (ELSMARI; NAVATHE, 2000).

De acordo com Silberschatz, Korth e Sudarshan (2006), a fim de que a integridade dos dados seja garantida torna-se necessário garantir as seguintes propriedades:

- **Atomicidade** – esta propriedade garante que ou todas as operações de uma transação são efetivadas com sucesso ou nenhuma delas é efetivada;
- **Consistência** – uma transação mal-sucedida não deve causar inconsistência no banco de dados. Caso haja uma falha em uma operação da transação, todas as modificações realizadas por esta operação e pelas demais operações da transação em questão devem ser revertidas ao estado original;
- **Isolamento** – nesta propriedade é necessário garantir que as transações estejam isoladas uma das outras, no sentido de que modificações realizadas por uma transação T1, não se tornem visíveis para qualquer outra transação, até que T1 execute com sucesso uma operação *commit*;
- **Durabilidade** – para que uma transação esteja de acordo com esta propriedade é necessário garantir que ela é durável (persistente), ou seja, após

uma transação executar uma operação *commit* é necessário que todas as atualizações realizadas por ela sejam aplicadas ao banco de dados, mesmo nos casos de ocorrência de falha do sistema em algum instante.

Essas propriedades são conhecidas como propriedades ACID, acrônimo este derivado da junção da letra inicial de cada propriedade.

5. Transações Distribuídas

Em um ambiente de banco de dados distribuídos, uma transação pode acessar dados gravados em mais de um local (*site*), sendo neste caso denominada uma **transação distribuída**. (BELL; GRIMSON, 1992).

Segundo Silberschatz, Korth e Sudarshan (2006), os sistemas de transações distribuídas são compostos por dois componentes:

- **Gerenciador de transações** – é responsável por controlar as transações (ou sub-transações) que acessam os dados localizados naquele *site* local.
- **Coordenador de transação** – coordena a execução de várias transações (locais e globais) iniciadas em um determinado *site*.

A efetivação ou aborto de uma transação distribuída torna-se muito mais difícil se comparada à ação equivalente em ambiente centralizado devido à descentralização de um sistema distribuído, no qual podem ocorrer problemas de rede e falhas de máquinas. Garantir a atomicidade de uma transação distribuída pode não ser uma tarefa fácil, uma vez que as transações (sub-transações) devem ser confirmadas ou abortadas em todos os sites envolvidos na transação. Portanto, para garantir essa propriedade, um coordenador de transação precisa de um protocolo de efetivação (protocolo commit) que garanta a sincronia do estado da transação de maneira consistente. (SILBERSCHATZ; KORTH; SUDARSHAN, 2006).

Para isso foram propostos vários protocolos. Dentre eles os Protocolos de efetivação (commit) em Duas Fases (2PC) e em Três Fases (3PC). O objetivo destes protocolos de efetivação consiste basicamente em chegar a um consenso.

- **Protocolo Commit de Duas Fases (2PC):** Dentre os protocolos de efetivação propostos, o de duas fases (2PC) é um dos mais simples e mais utilizados. Como o próprio nome sugere, este protocolo é composto por duas fases: uma fase de votação e uma de decisão. A idéia principal deste protocolo é que todos os participantes, gerenciadores de transações (sub-transações) locais, sejam consultados pelo coordenador da transação a fim de verificar se eles estão preparados ou não para efetivarem (commited) suas transações. (BELL; GRIMSON, 1992).
- **Protocolo Commit de Três Fases 3PC** - Este protocolo é uma extensão do protocolo commit de duas fases (2PC), que busca minimizar a possibilidade de incerteza em caso de falha do coordenador. É tolerante a falhas de sites e falhas de comunicação. Porém a complexidade da sua implementação é bem maior que a do 2PC, e ele proporciona um overhead maior na rede. (BELL; GRIMSON, 1992).

6. Transações Distribuídas em Web Services

Transações em Web Services possuem necessidades diferentes das necessidades previstas pelas transações tradicionais. Transações em Web Services, consistem em transações prolongadas e relaxadas. Transações prolongadas permitem o agrupamento de suas operações dentro de estruturas hierárquicas (sub-transações), enquanto que transações relaxadas indicam que um modelo de transação pode deixar de implementar alguma das propriedades ACID. (BELLI, 2005).

Segundo Limthanmaphon e Zhang (2004), as transações em Web Services não podem ser gerenciadas da mesma forma que as transações em sistemas distribuídos devido às seguintes características:

- Transações em web services geralmente são conduzidas além dos limites organizacionais. Isto significa que os participantes da transação podem ser independentes e estarem distribuídos pela Internet..
- Transações em web services podem ser de longa duração, ou seja, podem durar horas, dias e até mesmo meses. A utilização de um controle rígido por *timeout* torna-se inviável diante de tal cenário, uma vez que seria impossível determinar o valor ideal do *timeout*.

Portanto, em transações em web services deve-se tomar o cuidado para que os recursos não fiquem bloqueados por um longo período de tempo. No entanto, a garantia deste quesito torna-se impossível quando se pretende seguir as propriedades ACID, pois ACID indica que os recursos utilizados por uma transação devem ser bloqueados até que a execução da transação termine, para que as propriedades de isolamento e consistência sejam preservadas.

Devido à necessidade de que estes problemas sejam resolvidos, alguns protocolos de suporte a transações em Web Services já foram propostos. Dentre eles temos: os protocolos *OASIS Business Transaction Protocol* (BTP) e *Web Service Coordination/Web Services Transactions,* Orchard et al. (2005).

O protocolo BTP foi projetado para suportar aplicações independentes de localização e administração, que necessitam de suporte transacional diferente do suportado pelas transações ACID. Seu objetivo é tornar-se um protocolo base que oferece suporte transacional em termos de coordenação distribuída para múltiplas funcionalidades de negócio independentes, na forma de serviço. No entanto, segundo Little e Freund (2003), este protocolo não foi desenvolvido especificamente para transações em Web Services, e a intenção é que ele possa ser utilizado em outros ambientes. Já os protocolos Web Services Coordination (WS-C) e Web Service Transactions (WS-TX) foram projetados especificamente para Web Services e, portanto implementam as definições básicas da infra-estrutura de Web Services.

WS-C e WS-Tx foram propostos por um consórcio de várias empresas, dentre elas a Microsoft, a IBM e a BEA System. (BELLI, 2005). E segundo Little e Freund (2003), possuem o propósito de definirem semânticas para as transações. A especificação *Web Services Coordination* (WS-C) define um *framework* genérico de coordenação que pode suportar vários tipos de protocolos de coordenação. Ela prevê as seguintes regras:

- Instanciação (ou ativação) de um novo coordenador para um protocolo de coordenação específico;
- Registro dos participantes da transação;
- Propagação das informações do contexto entre os Web Services envolvidos na aplicação;
- Uma entidade para gerenciar o protocolo de coordenação até sua finalização.

Os três primeiros requisitos estão diretamente relacionados à especificação WS-C enquanto que o quarto requisito é definido pela especificação WS-Tx, sendo que geralmente é a aplicação cliente que controla a aplicação como um todo.

7. Cenário de Aplicação

O cenário implementado neste trabalho a fim de propor uma infra-estrutura de suporte a transações distribuídas em Web Services é o de comércio eletrônico, mais precisamente uma livraria virtual.

Conforme pode ser verificado na *Figura* , o cenário é composto por três aplicações, uma Livraria, uma Operadora de cartões de créditos e um Fornecedor de produtos (livros), no caso poderia haver mais de um fornecedor envolvido na transação . A operadora de cartões e o fornecedor de produtos desempenharão o papel do agente de software *provedor de serviços* da arquitetura SOA disponibilizando serviços web (Web Service) que serão posteriormente consumidos pela Livraria, que assumirá o papel do agente de software *consumidor de serviços*.

O serviço disponibilizado pela Operadora de Cartões consistirá na autorização ou não de uma compra por determinado cliente com base no saldo do seu cartão e no valor da sua compra. Por se tratar de um cenário hipotético, somente para fins didáticos, a única forma de pagamento que está Livraria suportará será através do cartão de crédito. Já o serviço disponibilizado pelo Fornecedor de Produtos informará a Livraria se determinada quantidade de certo produto está disponível ou não para venda.

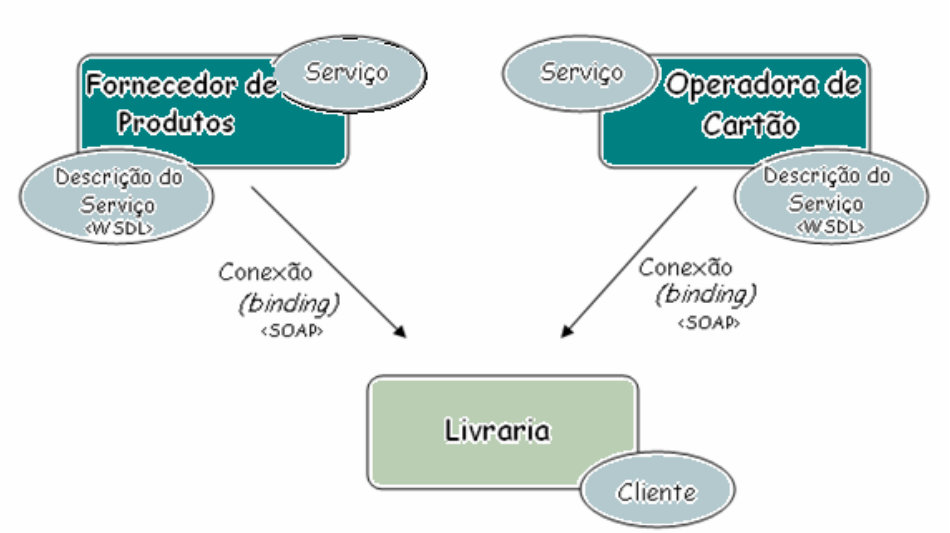

Figura 2. – Arquitetura do cenário proposto.

As três aplicações, envolvidas no cenário de aplicação proposto, foram implementadas utilizando-se a linguagem de programação Java, com o auxilio da IDE Eclipse versão 3.2. Para a construção dos serviços Web disponibilizado pela Operadora de Cartões e pelo Fornecedor de Produtos foi utilizado o Apache Axis, versão 1.4, que é um framework *open source*, implementado atualmente na linguagem de programação Java e baseado no padrão XML. Esta escolha foi feita com base na ampla aceitação e adoção deste framework na construção de WS.

A implementação do Web Service envolveu ainda a utilização de outras tecnologias como o Apache TomCat versão 5.5.23 e o parser XML *Xerces Java Parser* versão 2.9.0. O Apache Tomcat é um contêiner de Servlets, o qual será utilizado para disponibilizarmos os serviços Web da Operadora de Cartões e do Fornecedor de Produtos.

Para realizar o controle das transações distribuídas envolvidas no cenário proposto foi utilizado a implementação dos protocolos WS-Coordination e WS-AtomicTransaction proposta pelo projeto Apache Kandula. (KANDULA, 2007a). Essa escolha foi feita pelo fato desta implementação rodar sobre o apache axis.

Apesar do *framework* de coordenação de web services ser independente de plataforma, os serviços participantes inevitavelmente precisam utilizar tecnologias de plataformas especificas para realizar o controle transacional local. Neste trabalho foi utilizado o controle transacional local proposto pela Apache Geronimo, o qual implementa a API JTA. Esta escolha foi devido ao fato de que atualmente a implementação do kandula suporta somente este tipo de controle transacional local.

API JTA é uma API pertencente à plataforma Java EE, que especifica interfaces para a demarcação de transações em aplicações escritas na linguagem Java. (CHEUNG; MATENA, 2007). Através da interface JTA o desenvolvedor interage com o monitor de transação para determinar as fronteiras de uma aplicação, isto é, através da interface JTA ele define o inicio da transação e determina se ela será confirmada (*commit*) ou não (*rollback*).

O diagrama de seqüência, apresentado na *Figura* , demonstra como é realizado o gerenciamento global das transações distribuídas envolvidas no cenário proposto nesse trabalho. Como podemos verificar este cenário envolve a utilização de vários serviços: serviços de coordenação, autorizaVenda e autorizaCompra. No qual o papel de cliente dos serviços é alternado entre as aplicações Livraria, Operadora de cartões e Fornecedor de produtos

O serviço de coordenação, consiste basicamente na implementação da especificação WS-Coordination feita pelo Kandula, e dispõem três serviços: serviço de ativação, o qual dá suporte à criação de coordenadores para protocolos específicos e para seus contextos; serviço de registro, permite que os participantes se registrem para receberem o protocolo de mensagem associado a um coordenador particular. O contexto possui uma função muito importante na coordenação das transações, pois ele é o responsável por dar suporte à conexão das aplicações pertencentes aos Web Services em uma única aplicação coordenada. Serviço de finalização responsável por finalizar o gerenciamento da transação invocando os métodos commit ou rollback, implementados pelos gerenciadores de transação locais, dependendo do que ocorreu durante a execução das operações envolvidas durante a transação global. (KANDULA, 2007b).

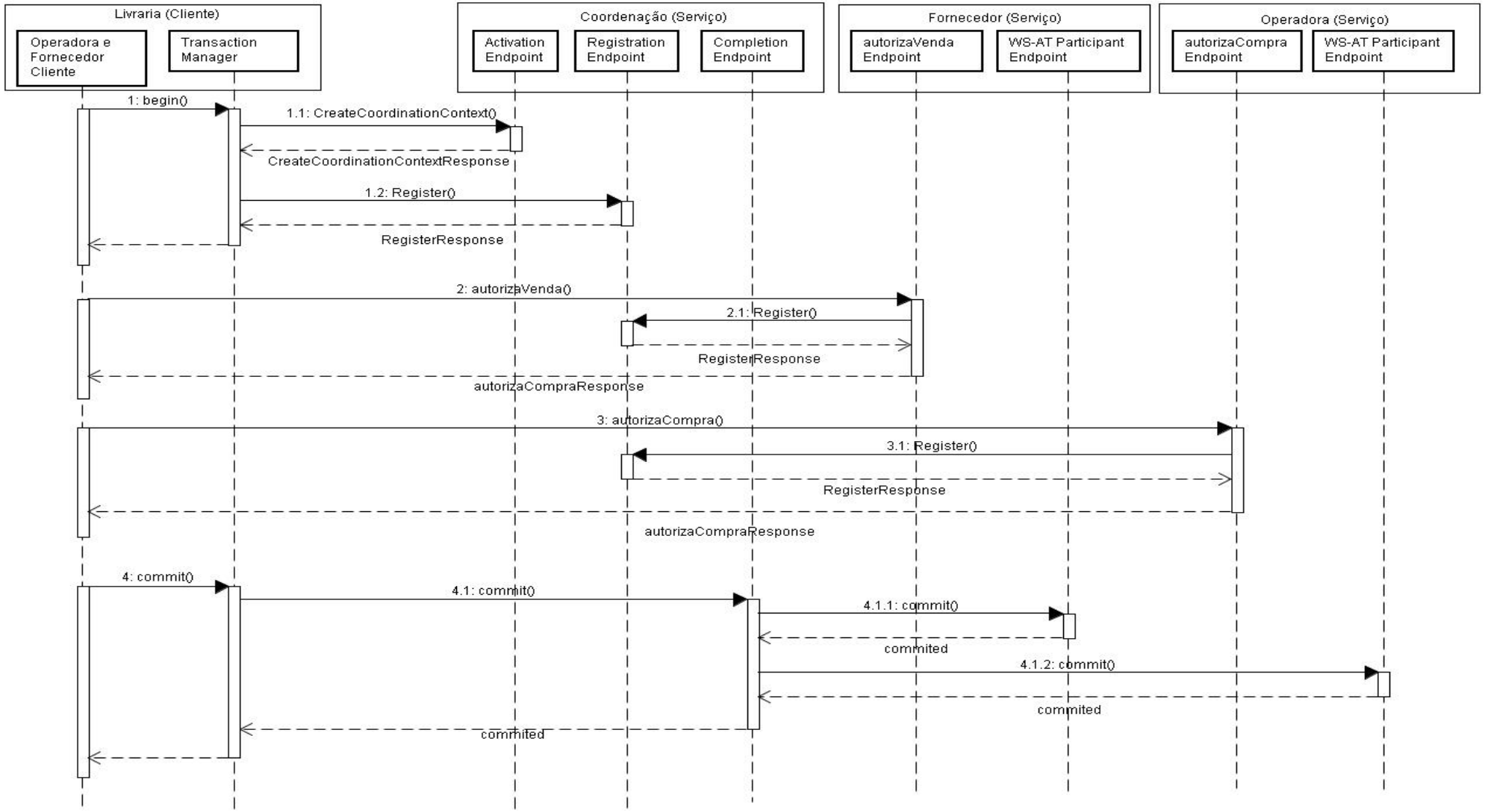

Figura 3. – Diagrama de Seqüência do gerenciamento de transações global no caso de efetivação de todas as transações locais envolvidas .

As especificações WS-C e WS-Tx interagem para executarem as tarefas do WS-C. Dois modelos de transação são definidos pela especificação WS-Tx, de modo que cada uma suporta uma semântica diferente O modelo de transação atômica (AT) é utilizado para coordenar atividades de curta duração. seguem as propriedades ACID e garantem que todos os participantes terão o mesmo resultado (atômico): todos efetivam a transação ou todos abortam a transação (propriedade "tudo ou nada"). Ele é baseado no protocolo de efetivação (commit) de duas fases, o modelo de transação de negócios (BA – Business Activity**)** fornece flexibilidade às propriedades das transações e foi projetado especificamente para interações de longa duração, nas quais o controle de recursos torna-se muitas vezes impossível ou impraticável. O cenário implementado neste trabalho envolve apenas transações atômicas, pois até o momento de conclusão deste trabalho somente este modelo estava implementado pelo projeto kandula.

A aplicação Livraria possui um gerenciador de transações, representado pelo objeto do tipo org.apache.kandula.coordinator.at.TransactionManagerImpl, o qual possui como função demarcar as fronteiras transacionais onde as operações afetam o contexto transacional das threads chamadas.

De acordo com o diagrama de seqüência um objeto TransactionManager é instanciado e excuta o método begin(), o qual por sua vez acessa o serviço de coordenação e cria um contexto de coordenação através do endpoint de ativação. Após a criação do contexto ele registra-se acessando o endpoint de Registro do serviço de coordenação. Na seqüência o serviço autorizaVenda é acessado através do seu endpoint pela Livraria. Nesse Nesse momento o controle de transacional local implementado na aplicação Fornecedor é executado. Durante esta execução ocorre o registro dessa transação, como participante da transação global, através do endpoint de registro do serviço de coordenação. Após o registro da transação o serviço autorizaVenda, propriamente dito, é executado e após sua execução retorna como resposta para a aplicação Livraria se possui ou não em seu estoque os produtos e as respectivas quantidades desejadas.

Na seqüência o serviço autorizaCompra é acessado através do seu endpoint pela Livraria, e ocorre o mesmo fluxo de execução ocorrido no acesso ao serviço autorizaVenda. Se a resposta obtida no acesso ah ambos os serviços forem positivas a aplicação Livraria informa ao objeto TransactionManager que as transações participantes devem ser efetivadas, ou seja o método commit() é invocado. Caso uma das respostas seja negativa o objeto TransactionManager é informado que ambas as transações participantes devem ser desfeitas , ou seja, o método rollback deve ser invocado.

Em ambas as situações quando o método commit ou rollback da instancia TransactionManager são invocados o endpoint de finalização do serviço de coordenação é acessado e ele invoca o método correspondente ao commit ou rollback implementado pelo controle de transações local da aplicação Fornecedor e da aplicação Operadora, garantindo assim a consistência dos dados envolvidos na transação global confirma venda da aplicação Livraria. Este diagrama de seqüência apresentado mostra apenas o fluxo de execução no caso em que as transações participantes devem ser efetivadas (commited). Um diagrama muito semelhante representaria o fluxo de execução no caso em que as transações precisassem ser desfeitas. A única coisa que mudaria é que a invocação de todos os métodos commit seria substituída pela invocação do método rollback.

Em termos de código de programação o controle transacional global realizado pela Livraria torna-se muito simples com a utilização do serviço de coordenação

implementado pelo projeto Kandula. Ela resume-se a instanciar um objeto do tipo *org.apache.kandula.coordinator.at.TransactionManagerImpl* invocar o seu método *begin,* depois invocar os serviços desejados, nesse caso especifico, *autorizaVenda* e *autorizaCompra,* e invocar o método *commit* ou *rollback* dependendo das respostas dos serviços invocados, isto pode ser verificado na *Figura* 4. Toda a complexidade do gerenciamento global das transações distribuídas dos serviços web é encapsulada pelo serviço de coordenação do projeto kandula.

Ambos os serviços *autorizaVenda* e *autorizaCompra* são invocados e suas respostas armazenadas em variáveis *booleanas*. Na seqüência suas respostas são verificadas, caso ambas sejam afirmativas (*true*) as transações locais realizadas tanto na Operadora de Cartões de Credito como no Fornecedor de Produtos são efetivadas. Caso uma delas seja negativa (*false*), ambas são desfeitas. Isso garante a consistência dos dados e consequentemente um sistema confiável. Note também que caso ocorra alguma exceção, como por exemplo, de conexão, durante o acesso a estes serviços ambas as transações são desfeitas.

```
public void confirmaVenda(long numeroCartao, float totalFatura, String idLoja,
        int[] idProd, int[] qtdades, String nomeLoja){

    boolean autorizaCompra;
    boolean autorizaVenda;

    TransactionManagerImpl tm = TransactionManagerImpl.getInstance();

    OperadoraService operadora = new OperadoraServiceLocator();
    Operadora op;
    try {
        op = operadora.getOperadora();

        FornecedorService fornecedor = new FornecedorServiceLocator();
        Fornecedor forn = fornecedor.getFornecedorSoapBindingImpl();

        tm.begin();
        autorizaCompra = op.autorizaCompra(numeroCartao, totalFatura, idLoja);
        autorizaVenda = forn.autorizaVenda(idProd, qtdades, nomeLoja);

        if (!autorizaCompra || !autorizaVenda){
            tm.rollback();
        }else{
            tm.commit();
        }
    }catch (RemoteException e1) {
        e1.printStackTrace();
        try {
            tm.rollback();
        } catch (RemoteException e2) {
            e1.printStackTrace();
        }
```

```
    } catch (ServiceException e1) {
        e1.printStackTrace();
        try {
            tm.rollback();
        } catch (RemoteException e2) {
            e1.printStackTrace();
        }
    }catch (Exception e) {
        e.printStackTrace();
        try {
            tm.rollback();
        } catch (RemoteException e1) {
            e1.printStackTrace();
        }
    }
}
```

Figura 4. – Trecho de código da classe Livraria.

8. Conclusão e Trabalhos Futuros

Com o surgimento dos Web Services, que se tornam cada vez mais populares devido as suas atrativas características, bem como interoperabilidade, flexibilidade, redução de custos operacionais e de desenvolvimento, reutilização de código e melhora no relacionamento entre parceiros e clientes, surgiu também a necessidade da especificação e implementação de novos protocolos de gerenciamento de transações. Isto se deve ao fato de que as transações realizadas em serviços web possuem características diferentes das transações tradicionais, como por exemplo, a possibilidade de transpor os limites organizacionais e o seu tempo de duração, que em alguns casos pode levar dias, dentre outros.

Neste trabalho foram apresentadas algumas soluções já propostas para o gerenciamento de transações distribuídas em web services. Dentre elas a implementação realizada pelo Apache, através do projeto Kandula, dos protocolos *WS-Coordination* e *WS-AtomicTransaction*, qual foi utilizada para gerenciar transações atômicas de serviços web distribuídos utilizados por um sistema de Livraria Virtual.

A utilização da implementação proposta pelo projeto Kandula nos permite realizar o gerenciamento global de transações distribuídas de forma bastante simplificada, uma vez que toda a complexidade deste gerenciamento é encapsulada pelo serviço de coordenação disponibilizado pelo projeto. No entanto, o projeto Kandula, por ainda estar em fase de desenvolvimento, é bastante instável e possui algumas deficiências, listadas a seguir, que dificultam a sua utilização.

- Quase total falta de documentação e suporte na distribuição do Kandula e a total falta de documentação na distribuição Kandula 2. Para poder utilizar a implementação proposta pelo projeto foi necessário analisar o seu código fonte;
- Fornecimento de informações contraditórias dependendo da parte do site do projeto acessada. Por exemplo, em determinados locais é informado que o projeto implementa o protocolo *WS-BusinnessActivity* e em outro local já informa que atualmente não há suporte para este protocolo, o que confunde o usuário desenvolvedor.

Um outro fator negativo da implementação do projeto Kandula é o fato de que atualmente ela suporta somente o controle de transações local JTA implementado Apache Geronimo, o que acarreta na perda de interoperabilidade entre sistemas heterogêneos, uma importante vantagem dos Web Services. Isto implica que, para utilizar o Kandula, todos os serviços web terão que ser implementados em Java e utilizar o controle de transações implementado pelo apache Geronimo.

Como possíveis trabalhos futuros, pode-se apontar:

- Uma extensão do cenário proposto com a inserção de um outro serviço web que envolva transações de longa duração que poderão ser coordenadas pelo protocolo Ws-B*usinnesActivity* .
- Migração das tecnologias utilizadas para implementação do cenário proposto neste trabalho do Axis para o Axis2 e consequentemente do Kandula para Kandula2.
- Experimentar outras implementações no controle de transações distribuídas e fazer um comparativo com a experiência realizada neste trabalho com a implementação proposta pelo Apache kandula.
- Devido a grande dificuldade encontrada na utilização da infra-estrutura proposta pelo projeto Apache Kandula seria interessante que fosse proposto uma maneira mais simples de se utilizar os protocolos WS-C e WS-Tx no controle de transações distribuídas em WS. Uma saída seria o desenvolvimento de um framework.

9. Referências

[Bell, Grimson, 1992] BELL, David; GRIMSON, Jane. **Distributed Database Systems,** England:Addison-Wesley Publishers, 1992. 410p.

[Belli, 2005] BELLI, Djan. Transações Distribuídas. Florianópolis, 10 p. Trabalho não publicado.

[Cheung, Matena, 2007] CHEUNG, Susan; MATENA, Vlada. **Java Transaction API (JTA)**, 2007. Disponível em: < http://sdlc-esd.sun.com/ESD31/JSCDL/jta/1.1/jta-1_1-spec.pdf?AuthParam=1181576773_733437da8a50b97dd8f07947c1b0feaa&TUrl=an1npDpbKod7kSYrROhENTonIec1W0D1Lc4nXz+pGFFranixdCdgxDTPbW4=&TicketId=dVF4NgBGMOo+8w==&GroupName=SDLC&BHost=sdlc6g.sun.com&FilePath=/ESD31/JSCDL/jta/1.1/jta-1_1-spec.pdf&File=jta-1_1-spec.pdf>. Acesso em 10 maio 2007.

[Cox et al., 2004] COX, CABRERA, Felipe; COPELAND, George; FREUND, Tom; KLEIN, Johannes; *STOREY,* Tony; THATTE, Satish . ***Web Services Transaction (WS-Transaction),*** 2004. Disponível em: < http://dev2dev.bea.com/pub/a/2004/01/ws-transaction.html>. Acesso em: 19 Ago. 2006.

[Elsmari, Navathe, 2000] ELSMARI. Ramez; NAVATHE, Shamkant B. **Sistema de Banco de Dados – Fundamentos e Aplicações.** Rio de Janeiro: LTC Editora, 2000. 795p.

[Kandula, 2007a] KANDULA. **Apache Kandula.** Disponível em < http://ws.apache.org/kandula/>. Acesso em: 9 maio 2006.

[Kandula, 2007b] KANDULA. **Architetural Design.** Disponível em < http://ws.apache.org/kandula/1/architecture-guide.html>. Acesso em: 22 maio 2007.

[Leandro, 2005] LEANDRO, Sabrina da Silva. **Balanceamento de Carga em Web Services**, 2005. 81 f. Trabalho de Conclusão de Curso (Bacharelado em Sistemas de Informação) – Universidade Federal de Santa Catarina, Florianópolis, 2005.

[Limthanmaphon, Zhang, 2004] LIMTHANMAPHON, Benchaphon; ZHANG, Yanchun. Web Service Composition Transaction Management. In: Fifteenth Australasian Database Conference (ADC2004), 50, 2003. **Research and Practice in Information Technology.** Dunedin: Australian Computer Society, 2003. V. 27. p. 171-179.

[Little, Freund, 2003] LITTLE, Mark; FREUND, Thomas; **A comparison of Web services transaction protocols** - A comparative analysis of WS-C/WS-Tx and OASIS BTP. Disponível em: <http://www-128.ibm.com/developerworks/webservices/library/ws-comproto/>. Acesso em: 05 Dez. 2006.

[Orchard et al., 2005] ORCHARD, David; CABRERA, Felipe; COPELAND, George; FREUND, Tom; KLEIN, Johannes; LANGWORTHY, David; SHEWCHUK, John; *STOREY,* Tony . ***Web Services Coordination (WS-Coordination),*** 2005. Disponível em: < ftp://www6.software.ibm.com/software/developer/library/WS-Coordination.pdf>. Acesso em: 07 Jul. 2006.

[Papazoglou, 2003] PAPAZOGLOU, Mike P. Service-Oriented Computing: Concepts, Characteristics and Directions. In: Web Information Systems Engineering Workshops, 2003 (WISE' 03), 04, 2003. **Proceedings Fourth International Conference on Web Information Systems Engineering.** IEEE, 2003. p. 3-12.

[Silberschatz, Korth, Sudarshan, 2006] SILBERSCHATZ, Abraham; KORTH, Henry F.; SUDARSHAN, S. **Sistema de Banco de Dados.** São Paulo: Elsevier Editora, 2006. 781p.