UNIVERSIDADE FEDERAL DO PARANÁ

FABRÍCIO JOSÉ DE OLIVEIRA CESCHIN

WEB SERVICES E TRANSAÇÕES ATÔMICAS EM SISTEMAS DISTRIBUÍDOS

CURITIBA

2015

FABRÍCIO JOSÉ DE OLIVEIRA CESCHIN

WEB SERVICES E TRANSAÇÕES ATÔMICAS EM SISTEMAS DISTRIBUÍDOS

Trabalho de Graduação apresentado como requisito parcial à obtenção do grau de Bacharel em Ciência da Computação da Universidade Federal do Paraná.
Orientador: Prof. Dr. Bruno Müller Junior

CURITIBA

2015

“Se você quer ser bem sucedido, precisa ter dedicação total,

buscar seu último limite e dar o melhor de si.“

Ayrton Senna

RESUMO

Atualmente, a integração de sistemas via web services é uma prática cada vez mais comum. Redes sociais, como o Twitter, disponibilizam web services que possibilitam utilizar seus dados em outros sistemas[17]. Contudo, todo software está propenso a falhas, sejam elas do próprio software ou hardware, que podem acabar gerando inconsistência entre os sistemas envolvidos em uma comunicação. Para resolver estes tipos de problemas, foram desenvolvidos protocolos de coordenação específicos para a comunicação entre web services. Este trabalho tem como objetivo apresentar o funcionamento desses protocolos e como utilizá-los em um ambiente distribuído, através de exemplos práticos utilizando sistemas web.

Palavras-chave: Sistemas Distribuídos, Web, Web Service, SOAP, WS-Coordination, WS-Transaction, Completion, Two-Phase Commit, Ruby on Rails.

ABSTRACT

Nowadays, systems integration through web services is an increasingly common practice. Social networks, like Twitter, provide web services that enable use their data in third-party systems[17]. However, all software might fail, whether the own software or hardware, and generate inconsistency between all the systems involved in a communication. To solve these problems, coordination protocols were designed specially for web services communication. This work aims to present the operation of these protocols and how to use them in a distributed environment, through practical examples using web systems.

Keywords: Distributed Systems, Web, Web Service, SOAP, WS-Coordination, WS-Transaction, Completion, Two-Phase Commit, Ruby on Rails.

SUMÁRIO

CAPÍTULO 1

INTRODUÇÃO

Em meados da década de 1980, os computadores se tornaram mais acessíveis e mais rápidos. A evolução da tecnologia foi tão grande que máquinas de milhões de dólares, que executavam apenas uma instrução por segundo, deram lugar à máquinas de mil dólares, que executavam um bilhão de instruções por segundo[36]. Isso foi um dos fatores decisivos para a sua popularização[32], gerando um aumento no parque computacional de muitas empresas. Entretanto, muitas máquinas acabavam ociosas e não havia como utilizá-las, tendo em vista que não era possível rodar, simultaneamente, um mesmo aplicativo em mais de um computador. Nesta mesma época, surgiram também as redes de alta velocidade, permitindo que uma maior quantidade de dados fosse transmitida. A união dessas tecnologias (computadores mais rápidos e redes de alta velocidade) viabilizou o desenvolvimento de aplicativos que podem ser executados em mais de um computador, comunicando-se através de uma rede, denominados sistemas distribuídos[36]. Um exemplo de sistema distribuído é uma aplicação que possui módulos em execução simultânea em diversos computadores. Embora melhore o desempenho, o paradigma de programação distribuída inclui a sincronização de tarefas entre os vários módulos, o que torna essa prática complexa de ser implementada pelos programadores. Consequentemente, novas tecnologias foram desenvolvidas por diversos fabricantes para facilitar o desenvolvimento de sistemas distribuídos, dentre elas o RPC (remote procedure call)[2], uma tecnologia de comunicação entre processos que permite a um programa chamar um procedimento em outro computador[4]. Devido ao seu sucesso, diversos padrões de RPC foram propostos, cada um compatível com determinadas plataformas e muitos incompatíveis entre si, o que tornava a integração de sistemas diferentes muito difícil[2] e muitas vezes forçava as empresas a adotarem um sistema operacional ou linguagem específica. Um dos padrões mais conhecidos é o CORBA (Common Object Request Broker Architecture), que estende

o modelo para o paradigma de orientação a objetos e fornece várias ferramentas que simplificam o desenvolvimento de aplicações distribuídas orientadas a objetos[30].

Com a criação da internet, em 1984[32], e o desenvolvimento de programas que permitiram a sua popularização, como o World Wide Web, em 1991[7], e o navegador Netscape, em 1994 [7], surgiu a necessidade de integrar softwares de diferentes plataformas, devido a heterogeneidade das aplicações web. Softwares de vendas online, por exemplo, tinham que se comunicar com o sistema de determinado banco para validar os pagamentos dos clientes. Entretanto, dependendo da plataforma adotada, a comunicação através do RPC não era possível. Para resolver esses problemas, em 1999, diversas empresas de tecnologia, como Microsoft e IBM, se uniram e criaram o SOAP, uma forma de trocar mensagens através da linguagem de marcação XML, adotada pelo W3C (World Wide Web Consortium), a principal organização de padronização da World Wide Web[24]. Assim surgiu o conceito de web service[2], uma extensão do RPC para a internet independente de plataforma.

Uma infraestrutura básica de um web service permite que um cliente solicite a execução de uma determinada ação no servidor, através de operações simples[30], assim como no RPC. Contudo, no mundo real, as interações costumam ser mais complexas e envolvem um conjunto de operações que necessitam ser executadas em sequência[30] e atomicamente, uma vez que, caso haja uma falha, as aplicações envolvidas podem ter sua consistência comprometida[31]. Para ilustrar este tipo de problema, considere a figura 1.1, um diagrama de sequência para o caso de uso "comprar produto" em um sistema distribuído que envolve uma loja e um banco. Se algum ator envolvido (cliente, servidor ou banco) tiver sua comunicação interrompida, o aplicativo deve garantir que nenhuma ação inconsistente ocorra. A figura mostra uma situação que pode gerar inconsistência entre as aplicações. Quando um cliente compra um produto, a aplicação da loja se comunica com o banco para verificar se existe dinheiro disponível. Supondo que haja uma queda no sistema da loja e o cliente possua dinheiro suficiente para efetuar a compra, o mesmo será descontado de sua conta. Entretanto, o resultado da operação não será conhecido pela loja, não efetivando a compra. Uma das formas de garantir a consistência é conhecida na literatura como transação atômica [34], um mecanismo responsável por garantir que ou

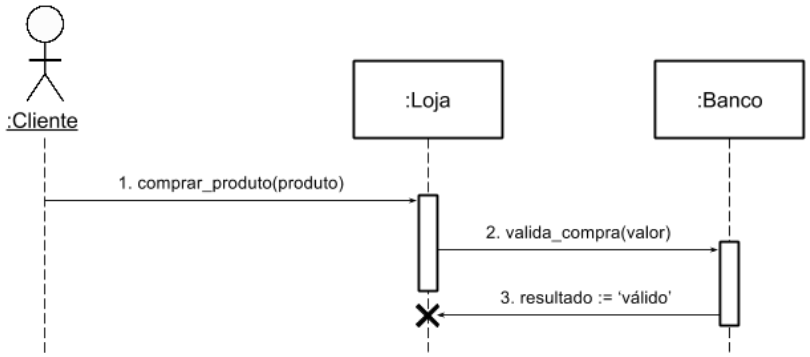

Figura 1.1: Diagrama de sequência que exemplifica a inconsistência gerada por um erro (uma queda da máquina da loja, por exemplo).

toda transação ocorre ou nada da transação ocorre. Por conta desses tipos de problemas, em 2002, a IBM, Microsoft e a BEA propuseram o WS-Coordination, um framework para suportar protocolos de coordenação como o WS-Transaction, proposto na mesma época pelas mesmas empresas[30] e utiliza o conceito de transação atômica.

Este trabalho tem como objetivo descrever o funcionamento como utilizar o WS-Coordination e o WS-Transaction em transações envolvendo web services, usando como exemplo implementações feitas com o framework ruby on rails. Primeiramente, no capitulo 2, são apresentados os conceitos utilizados para a realização do trabalho: sistemas distribuídos, na seção 2.1, sistemas web, na seção 2.2, e tecnologias, na seção 2.3. No capítulo 3, são apresentados os protocolos de coordenação citados: o WS-Coordination, na seção 3.1, e o WS-Transaction, na seção 3.2. Já no capítulo 4, foram desenvolvidos sistemas para exemplificar o uso dos protocolos apresentados. Na seção 4.1 são apresentadas as bibliotecas utilizadas para o desenvolvimento do sistema utilizando operações simples, disponível na seção 4.2, e utilizando protocolos de coordenação, disponível na seção 4.3.

CAPÍTULO 2

REVISÃO BIBLIOGRÁFICA

Este capítulo tem como objetivo apresentar os conceitos que contribuíram para a realização deste trabalho. Primeiramente, serão apresentados os sistemas distribuídos e alguns protocolos de sincronização. Em seguida, serão apresentadas as semelhanças de um sistema distribuído e um sistema web e os desafios envolvidos. Após explicar as tecnologias utilizadas no trabalho, o objetivo do mesmo será descrito utilizando os conceitos apresentados.

2.1 SISTEMAS DISTRIBUÍDOS

Um sistema distribuído é um conjunto de computadores independentes que se apresenta a seus usuários como um sistema único e coerente[30]. Este paradigma apresenta uma série de desafios, uma vez que seu desenvolvimento é mais complexo que um sistema centralizado. É necessário lidar com a heterogeneidade (diferentes redes, hardware, sistemas operacionais e linguagens de programação), a segurança (criptografia e autenticidade), a escalabilidade (desempenho satisfatório), a transparência (sistema deve ser visto como um todo) e a tolerância a falhas[35], que será discutida na próxima seção.

2.1.1 Tolerância a Falhas e Transações

Compostos por diversos componentes, os sistemas distribuídos são propensos a falhas parciais. Entretanto, tais falhas podem não afetar toda a aplicação, diferente de um sistema centralizado em geral[36]. Um sistema distribuído deve ser capaz de se recuperar automaticamente desse tipo de falha, sem gerar inconsistência entre seus componentes. A atomicidade é uma propriedade importante nestes casos[36], já que pode evitar inconsistência por meio de transações, que garantem que ou todas ou nenhuma das operações sejam executadas. Esta propriedade é uma das quatro características de uma transação,

conhecidas como ACID[36], regra que todo SGBD (Sistema Gerenciador de Banco de Dados) deve cumprir[3]:

1. Atomicidade: a transação deve acontecer de forma indivisível. Ou acontece completamente ou não acontece[36].

2. Consistência: a transação deve levar o sistema de um estado consistente para outro estado consistente[1].

3. Isolamento: cada transação deve parecer como se estivesse sendo executada isoladamente de outras transações[33].

4. Durabilidade: uma vez que uma transação acontece, ela deve continuar e seus resultados devem ser permanentes. Nenhuma falha pode desfazer os resultados ou provocar sua perda[36].

Entretanto, cada máquina do sistema distribuído controla apenas uma transação. Isso não garante que todas as transações, espalhas por diversos componentes, executem de modo atômico, ou seja não garante a consistência de um sistema distribuído. Para isso, é necessário utilizar, além de transações, um protocolo de coordenação. Um dos principais protocolos de coordenação é o *two-phase commit*, apresentado a seguir e implementado neste trabalho.

2.1.2 Two-Phase Commmit Protocol

O *two-phase commit* é um protocolo de transação distribuída que tem como pilares os seguintes itens[31]:

- Todo componente participante de uma transação distribuída deve possuir um log local que possui informações suficientes para colocar o participante em um estado consistente, caso ocorra alguma falha ou queda, através de algoritmos de recuperação.

- Cada computador participante de uma transação é capaz de tornar duráveis, de modo não definitivo, qualquer alterações de dados. Ou seja, cada computador deve ser capaz de desfazer as alterações, já que podem não ser definitivas.

- Um dos computadores participantes de uma transação distribuída é chamado de coordenador, que é responsável por gerenciar a transação (decide se ela é efetivada ou abortada).

Como o próprio nome diz, o protocolo é dividido em duas fases:

1. Votação: o coordenador inicia uma transação, enviando mensagens "prepare" a cada um dos participantes. Estes participantes decidem se desejam efetivar ou abortar a transação. Caso um participante queira efetivar esta transação, ele efetua as alterações dos dados, de forma não definitiva, adicionando "ready" ao seu log local e informando ao coordenador que seu voto é "sim". Em contrapartida, caso um participante queira abortar a transação ou não puder fazer as alterações de forma não definitiva, ele escreve "don't commit" em seu log local e informa ao coordenador que seu voto é "não"[31].

2. Efetivação: o coordenador efetua a apuração dos votos. Se todos os participantes votarem "sim", o coordenador escreve "commit" em seu log, envia uma mensagem de commit a todos os participantes e a transação é efetivada. Assim que um participante recebe uma mensagem de commit do coordenador, ele deve efetivar as alterações dos dados (que não eram definitivas) e escrever "commit" em seu log. Caso contrário, se houver ao menos um voto "não", o coordenador grava "rollback" em seu log e envia uma mensagem de rollback para todos os participantes que votaram "sim". Assim que recebe a mensagem de rollback, o participante desfaz as alterações (que não eram definitivas) e registra "rollback" em seu log[31].

2.2 SISTEMAS WEB

A World Wide Web (WWW) pode ser considerada um enorme sistema, composto de milhões de clientes e servidores, para acessar documentos, serviços e aplicações[36].

2.2.1 Sistemas Distribuídos e a Web

Um problema comum em sistemas web é a facilidade com que eles ficam sobrecarregados [36]. Tal problema pode trazer prejuízo para os responsáveis e frustração para os usuários, já que o mesmo pode ficar indisponível ou com um tempo de resposta muito alto. Para cada minuto que o Facebook fica fora do ar, por exemplo, a empresa perde cerca de US$ 20 mil[6]. Uma solução para a sobrecarga seria replicar um servidor em diversas máquinas, formando um cluster de servidores, e utilizar um único computador, chamado de front end, para redirecionar as requisições a um desses servidores[36].

Além dos clusters de servidores, aplicações que utilizam web services também podem ser considerados sistemas distribuídos, já que envolvem um conjunto de computadores e se apresentam ao usuário como um único sistema[36]. Por conta disso, web services estão propensos as mesmas falhas apresentadas na seção anterior. A seção a seguir mostra um exemplo de falhas envolvendo web services.

2.2.2 Exemplo

Considere um sistema de loja online semelhante a Amazon, que permite a venda de produtos por terceiros. Estes possuem uma aplicação de gerenciamento de estoque, que se comunica diretamente com a loja. Para que clientes possam comprar os produtos, a loja deve ser capaz de se comunicar com sistemas de diversos bancos. Dado estes cenários, temos um sistema distribuído envolvendo três aplicações, que se comunicam através de web services: uma loja, um gerenciador de estoque de um fornecedor e um banco. Na maior parte das vezes, a comunicação acontece entre a loja e seus fornecedores, para listar apenas produtos que estão disponíveis aos usuários e fornecer informações a respeito de determinado produto. Como se trata apenas de consultas, sem nenhuma escrita, tais trocas de mensagem não apresentam problemas de consistência no caso de falhas. Entretanto, quando um cliente deseja comprar um produto, caso o banco valide a compra, operações de escrita devem ser feitas nos sistemas do fornecedor e do banco. Neste caso, é possível que alguma falha ocorra, gerando inconsistência em ambos os sistemas. Suponha que um cliente compre um produto. A loja se comunica com o fornecedor e

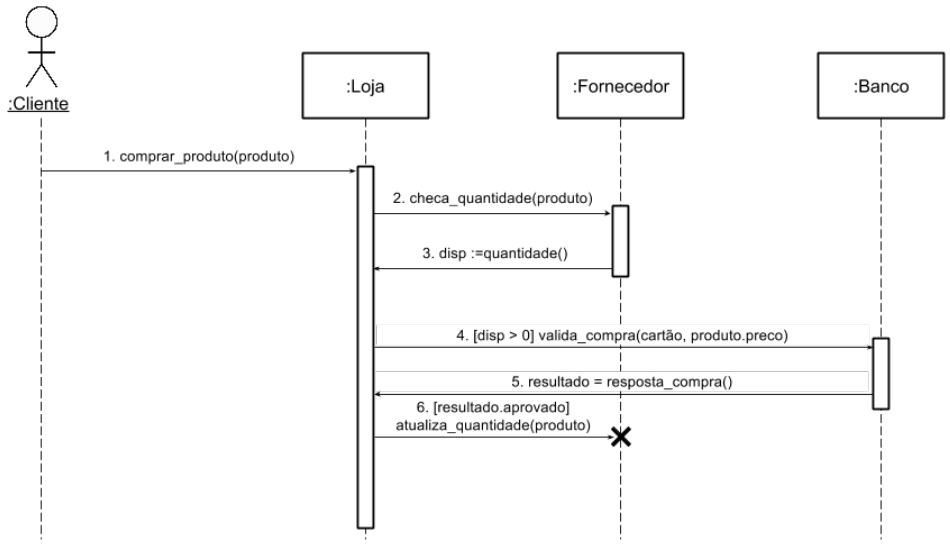

Figura 2.1: Diagrama de sequência que exemplifica a inconsistência gerada pela queda do fornecedor.

verifica que o item ainda está disponível. Assim, a loja deve agora se comunicar com o banco, para verificar se o cliente possui saldo para efetuar a compra. Supondo que o cliente possua saldo disponível, o banco desconta o valor do produto de sua conta e avisa a loja que a compra foi validada. O próximo passo seria avisar ao fornecedor que a compra foi efetuada, para diminuir a quantidade de produtos disponível em estoque. Entretanto, neste meio tempo, o sistema de gerenciamento de estoque do fornecedor caiu, gerando inconsistência: o sistema do fornecedor continuará com o mesmo número de produtos no estoque sendo que banco já descontou o valor referente a compra, ou seja, a compra foi aprovada. A figura 2.1 representa o exemplo citado.

2.3 TECNOLOGIAS

Esta seção apresenta as tecnologias utilizadas para a realização deste trabalho.

2.3.1 HTML

Criado em 1991[23], o HTML (HyperText Markup Language) é uma linguagem de marcação utilizada na construção de páginas web, que são interpretadas por navegadores. Todo documento HTML possui elementos formados por tags, que podem conter atributos,

valores e elementos filhos[22]. A linguagem é um padrão internacional cuja especificações são mantidas pelo W3C (World Wide Web Consortium) e o WHATWG (Web Hypertext Application Technology Working Group)[21]. Lançada em 2014, a especificação mais recente é o HTML5.

2.3.2 XML

Recomendação da W3C desde 1998[26], o XML (eXtensible Markup Language) é uma linguagem de marcação utilizada na criação de documentos com dados organizados de forma hierárquica. Por conta da sua portabilidade (formato independente de plataformas de hardware ou software), o XML é geralmente utilizado na integração de sistemas[27].

2.3.3 CSS

CSS (Cascading Style Sheets) é uma linguagem de estilo, criada em 1996[19], utilizada para descrever a apresentação de um arquivo HTML ou XML. Assim como o HTML, ela também é padronizada pelo W3C[18]. O CSS define regras, chamadas de seletores, que fazem as definições de estilo casarem com um ou mais elementos[19].

2.3.4 JavaScript

Javascript é a principal linguagem de programação utilizada no lado do cliente (client side) na web. É uma linguagem interpretada, criada em 1995, que permite controlar o navegador, realizar comunicações assíncronas e alterar o conteúdo exibido[8].

2.3.5 Framework

Um framework provê uma solução para um conjunto de problemas semelhantes através de várias classes e interfaces. Os objetos dessas classes, que são flexíveis e extensíveis, colaboram para cumprir suas responsabilidades afim de resolver determinados problemas. Resumidamente, um framework é uma aplicação quase completa, mas com pedaços específicos faltando. Estes pedaços devem ser "completados" por desenvolvedores, que resolvem

determinado problema[9].

2.3.6 Ruby

Ruby é uma linguagem de programação interpretada e multiparadigma, criada no Japão em 1995. A linguagem, inspirada principalmente por Python, Perl, Smaltalk, Eiffel, Ada e Lisp, suporta programação funcional, orientada a objetos, imperativa e reflexiva[13]. A versão mais recente do Ruby é a 2.2.3[10].

2.3.7 RubyGems

RubyGems é um gerenciador de pacotes para a linguagem Ruby que padroniza a distribuição de programas e bibliotecas, chamadas de gema. A ferramenta, que facilita a instalação das gemas, foi criada em 2003 e inclusa na biblioteca padrão do Ruby a partir da versão 1.9[14].

2.3.8 Ruby on Rails

Ruby on Rails é um framework open-source escrito em Ruby voltado ao desenvolvimento ágil. Criado em 2004, o framework utiliza o padrão de arquitetura MVC (Model-View-Controller), que divide o software em três partes interconectadas[12]. A versão mais recente do Rails é a 4.2.4[11].

2.3.9 Web Service

Embora existam diversas definições, um web service pode ser considerado uma aplicação acessível para outras aplicações através da web. Os web services surgiram para suprir a necessidade de integrar sistemas de plataformas diferentes[30]. Existem duas implementações de web services: SOAP e REST. Neste trabalho foi utilizado apenas a implementação SOAP.

2.3.9.1 SOAP

O SOAP (Simple Object Access Protocol) foi criado em 1999 através da união de várias empresas de tecnologia, como a IBM e a Microsoft, com o objetivo de padronizar a integração de aplicações através da web utilizando mensagens XML. As mensagens XML podem ser transportadas sobre o protocolo HTPP ou SMTP e são utilizadas como um envelope que permite inclusão de dados pela aplicação.[30]. A estrutura de uma mensagem SOAP é a seguinte:

- Envelope: identifica o documento XML como uma mensagem SOAP.
- Header: contém informações sobre a mensagem (horário de envio, credenciais, etc).
- Body: contém os dados de uma chamada de procedimento (identificador e parâmetros) ou de uma resposta (o retorno do método).
- Fault: elemento filho do body que possui informações sobre erros.

Abaixo, um exemplo de mensagem SOAP[25]:

```
<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
<soap:Header>
...
</soap:Header>
<soap:Body>
...
<soap:Fault>
...
</soap:Fault>
</soap:Body>
</soap:Envelope>
```

O cliente de um web service SOAP pode acessar seus serviços através do WSDL (Web Services Description Language), um documento que descreve o web service, especificando a localização do serviço e seus métodos[28]. Através deste documento, o cliente pode executar qualquer método listado. O WSDL possui os seguintes elementos:

- Types: define os tipos de dados utilizados pelo web service.
- Messages: define o conteúdo de cada mensagem do web service (parâmetros e retorno de método).
- PortType: descreve as operações que podem ser feitas e as mensagens envolvidas.
- Binding: define o protocolo e o formato de dados para cada port type.

Abaixo, um exemplo de WSDL[25]:

```
<definitions>
<types>
data type definitions.....
</types>
<message>
definition of the data being communicated....
</message>
<portType>
set of operations.....
</portType>
<binding>
protocol and data format specification....
</binding>
</definitions>
```

Além do WSDL, existe um documento capaz de armazenar informações de vários web services, chamado de UDDI (Universal Description Discovery), que tem como objetivo facilitar a descoberta de serviços aos desenvolvedores[30].

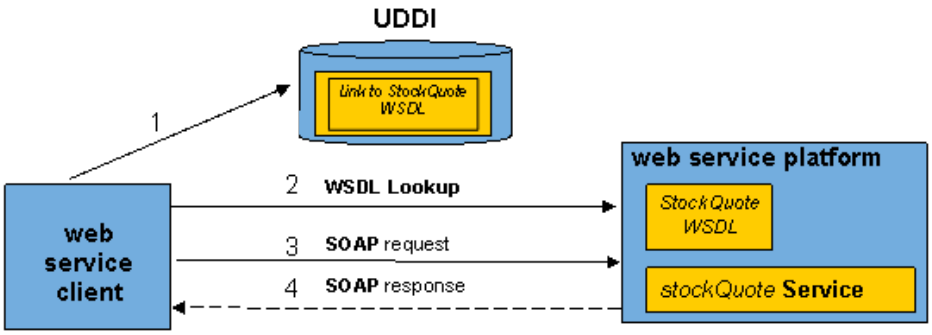

Figura 2.2: Diagrama que representa o funcionamento de um web service SOAP.

A figura 2.2 exemplifica o funcionamento de um web service SOAP[20]: um cliente descobre a localização do WSDL de um web service utilizando o UDDI (1). Em seguida, utiliza o WSDL para localizar seus métodos (2). Então, o cliente executa um método (3) e o servidor lhe envia o retorno (4).

2.3.10 Semáforo

Um semáforo é uma variável ou tipo abstrato de dados que tem como objetivo controlar o acesso a recursos compartilhados em um sistema concorrente. Criado em 1965 por Edsger Dijkstra, semáforos foram utilizados pela primeira vez no sistema operacional THEOS. Em situações que necessitam de exclusão mútua, utiliza-se um semáforo binário, chamado de mutex, permitindo que apenas um processo execute por vez[16].

2.4 OBJETIVO DO TRABALHO

Este trabalho tem como objetivo descrever o funcionamento e a utilização dos protocolos de coordenação e transação em sistemas distribuídos que utilizam web services. Para isso, foram desenvolvidos três sistemas web utilizando o framework Ruby on Rails (uma loja, um fornecedor e um banco), todos integrados através de web services SOAP. Inicialmente são apresentadas as falhas geradas ao não utilizar transações, sem ao menos usar semáforos para controlar regiões críticas. Em seguida, para resolver problemas de concorrência, são utilizados semáforos e apresentados os erros que esta solução não cobre.

A solução final é proposta utilizando os protocolos WS-Coordination e WS-Transaction, apresentados no capítulo a seguir.

CAPÍTULO 3

PROTOCOLOS DE COORDENAÇÃO

Assim como no RPC, os web services fazem com que chamadas de procedimentos remotos pareçam exatamente como as locais[36]. Entretanto, conforme visto na seção 2.2.2, este paradigma está propenso a falhas. No caso citado, são cinco classes diferentes de falhas[36]:

1. Cliente não consegue localizar o servidor.
2. Mensagem de requisição do cliente para o servidor se perde.
3. Servidor cai após receber uma requisição.
4. Mensagem de resposta do servidor para o cliente se perde.
5. Cliente cai após enviar uma requisição.

As falhas 1 e 2 podem ser facilmente solucionadas utilizando um tempo de timeout, ou seja, se não houver uma resposta em um determinado tempo, o servidor não foi localizado ou a mensagem se perdeu. Entretanto, as falhas 3, 4 e 5 podem ser complexas de mais para serem resolvidas, principalmente quando se trata de uma comunicação envolvendo várias máquinas. A figura 3.1 ilustra três acontecimentos possíveis em uma comunicação com um servidor. A primeira delas 3.1(a) mostra um comportamento normal, ou seja, uma requisição de um cliente chega ao servidor, é processada e a resposta é enviada ao cliente. Já na figura 3.1(b), podemos observar a queda do servidor após o processamento

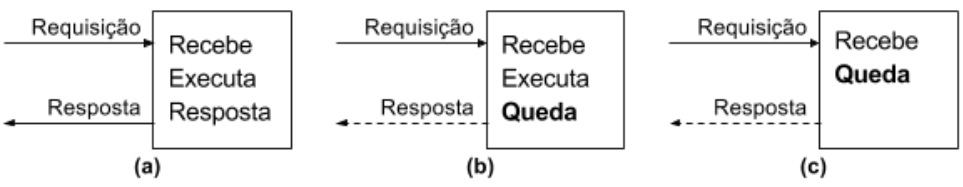

Figura 3.1: Queda do servidor ao comunicar-se com o cliente. (a) caso normal. (b) queda após execução. (c) queda antes da execução.

de uma requisição enviada pelo cliente, que não recebe a resposta, gerando, possivelmente, uma inconsistência. Em contrapartida, na figura 3.1(c), o servidor recebe a mensagem do cliente e, logo em seguida, cai, sem processar a requisição. Em ambos os casos (b) e (c), o cliente não sabe se a requisição foi processada ou não.

Com o intuito de permitir a implementação de mecanismos que possibilitem a detecção e tratamento deste tipo de falha, a IBM, Microsoft e a BEA propuseram, em 2002, as especificações dos protocolos WS-Coordination e WS-Transaction[30], apresentados nas seções 3.1 e 3.2, respectivamente.

3.1 WS-COORDINATION

O WS-Coordination é um framework criado para coordenar atividades genéricas e suportar protocolos de coordenação. Ele só é útil quando sua funcionalidade é estendida através de protocolos de coordenação, definidos em outras especificações. Logo, o framework não define nenhum protocolo de coordenação, mas sim sua extensão (WS-Transaction, no caso deste trabalho)[30, 31]. As entidades básicas do WS-Coordination são os coordenadores, que atuam como mediadores, e os participantes, que desejam ser coordenados durante uma comunicação. Além dessas entidades, a especificação utiliza três abstrações para descrever a interação entre um coordenador e um participante:

- Protocolo de coordenação: define a semântica e as regras para as trocas de mensagens entre o coordenador e os participantes de uma atividade coordenada[31]. Exemplo: *two-phase commit (2PC)*.

- Tipo de coordenação: representa um conjunto de protocolos de coordenação logicamente relacionados. Exemplo: WS-Transaction, que agrupa o *two-phase commit* e o *completion*, utilizado pelos participantes que querem ser informados sobre o resultado do *two-phase commit*.

- Contexto de coordenação: estrutura de dados utilizada para identificar as mensagens pertencentes a mesma conversa (chamada de coordenação). O contexto contém um campo que identifica o tipo de coordenação e outro que identifica unicamente

a instância desse tipo de coordenação. Todos os participantes de uma conversa incluem o contexto no header da mensagem SOAP[30].

São definidas também três formas de interação entre um coordenador e seus participantes:

1. Serviço de ativação: A figura 3.2 ilustra o funcionamento do serviço de ativação.

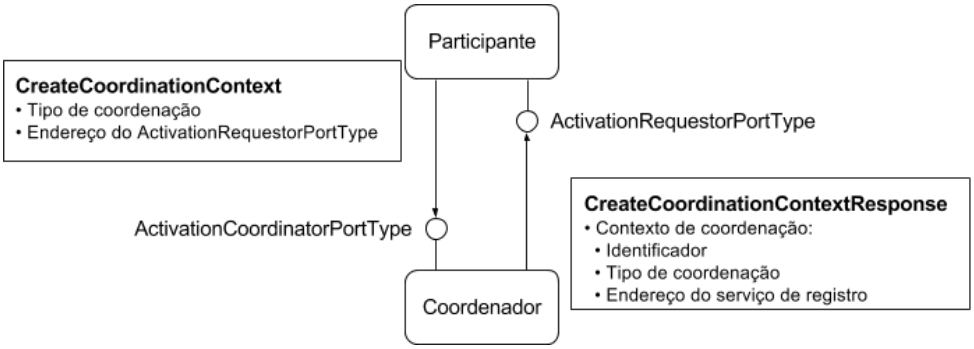

Figura 3.2: WS-Coordination: ilustração do serviço de ativação.

Um participante solicita a um coordenador para criar um novo contexto de coordenação. Novos contextos são criados sempre que um participante inicializa uma instância de um tipo de coordenação (uma conversa). Isso acontece sempre que uma transação é inicializada. O WS-Coordination define *port types*, um grupo de operações[31], para cada um desses papéis: *ActivationCoordinationPortType* e *ActivationRequestorPortType*. O primeiro deve ser implementado pelo coordenador e possui uma operação chamada *CreateCoordinationContext*, que permite criar novos contextos de coordenação. Já o segundo deve ser implementado pelo participante e possui a operação *CreateCoordinationContextResponse*, que é responsável por receber a resposta do método *CreateCoordinationContext* do coordenador. A mensagem que o participante envia ao coordenador (*CreateCoordinationContext*) deve especificar o tipo de coordenação do contexto a ser criado (WS-Transaction, por exemplo) e o seu endereço. A resposta do coordenador (*CreateCoordinationContextResponse*) contém os dados do contexto de coordenação criado (identificador e o tipo de coordenação), incluindo o endereço do serviço de registro, que pode ser utilizado pelo participante[30, 31].

2. Serviço de registro: A figura 3.3 ilustra o funcionamento do serviço de registro.

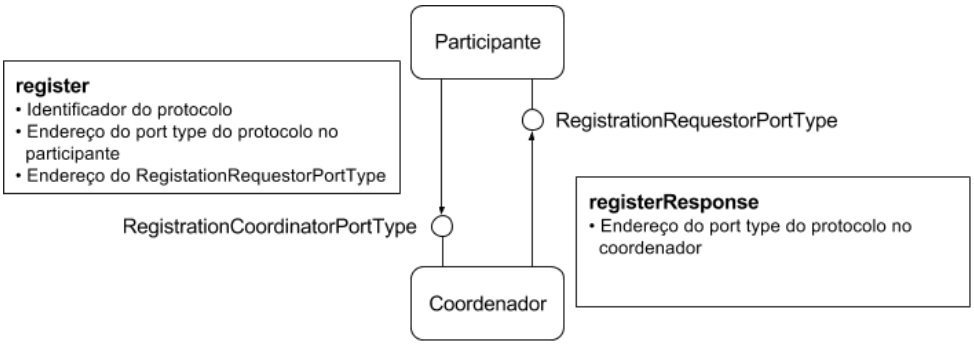

Figura 3.3: WS-Coordination: ilustração do serviço de registro.

Um participante se registra como um participante de um protocolo de coordenação com um coordenador. Efetuando este registro, um web service declara que ele está participando da execução do protocolo e que ele deve ser notificado quando os passos determinados de coordenação são executados. Através disso, um participante pode se registrar em uma transação atômica, disponibilizando suas informações ao coordenador. Assim como a ativação, o WS-Coordination também define *port types* para o coordenador e o participante: *RegistrationCoordinatorPortType* e *RegistrationRequestorPortType*. O primeiro é implementado pelo coordenador e possui a operação *Register* e, o segundo, implementado pelo participante e possui a operação *RegisterResponse*. A mensagem *Register* que o participante envia ao coordenador deve conter o identificador do protocolo utilizado nesta comunicação e o endereço do *port type* deste protocolo. Após enviar a mensagem *Register*, o participante recebe a resposta (*RegisterResponse*), que contém o endereço do coordenador do protocolo que o participante se registrou. Este endereço é utilizado para enviar mensagens ao coordenador, de acordo com o tipo de coordenação. No caso do *two-phase commit* (WS-Transaction), o participante pode enviar as mensagens *prepared*, *aborted*, *readonly*, *committed* e *replay*. Assim, o participante e o coordenador conhecem seus endereços, possibilitando que exista uma troca de mensagens entre eles (interações específicas do protocolo, próximo item)[30, 31].

3. Interações específicas do protocolo: após a ativação e o registro, o coordenador e seus participantes trocam mensagens especificas de acordo com o protocolo de

coordenação.

Resumidamente, o WS-Coordination define como funcionam as interações para ativação e registro, independente do tipo de coordenação utilizado. Este tipo de coordenação deve definir quais são as interações após os passos executados pelo WS-Coordination.

3.2 WS-TRANSACTION

O WS-Transaction é um conjunto de especificações construídas sobre o WS-Coordination[30] e define o tipo de coordenação "transação atômica", possibilitando que o coordenador genérico do WS-Coordination lide com transações atômicas distribuídas.

3.2.1 Protocolos de Coordenação

O WS-Transaction define o uso de ao menos dois tipos de protocolos de coordenação: o *completion* e o *two-phase commit*[31].

3.2.1.1 Completion

O *completion* é um protocolo que tem como propósito informar ao coordenador que ele deve iniciar o *two-phase commit* para verificar o resultado da transação[30]. Neste protocolo são definidos dois papéis: o coordenador, mesmo do WS-Coordination, e o iniciador (ou cliente), responsável por informar ao coordenador que ele deve iniciar o *two-phase commit*. O WS-Transaction define *port types* para ambos os papéis: *CompletionCoordinatorPortType*, implementado pelo coordenador e possui as operações *Commit* e *Rollback*, e *CompletionInitiatorPortType*, implementado pelo iniciador e possui as operações *Committed* e *Aborted*. Primeiramente, o iniciador envia uma mensagem *Commit* ou *Rollback* ao coordenador. Caso envie a mensagem *Rollback*, a transação é abortada e ele responde ao iniciador *Aborted*, caso contrário, ele executa o protocolo *two-phase commit*. Após a execução, o coordenador deve responder ao iniciador *Committed* ou *Aborted*[31]. A figura 3.4 ilustra o funcionamento do protocolo *completion*.

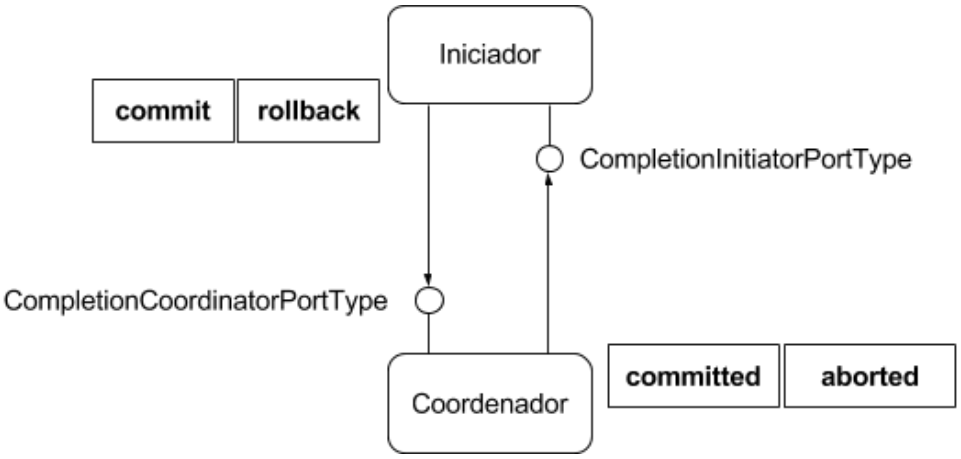

Figura 3.4: WS-Transaction: ilustração do protocolo *completion*.

3.2.1.2 Two-phase Commit

O *two-phase commit* é um mecanismo que permite que múltiplos participantes cheguem a um consenso sobre o desfecho de uma transação. O protocolo WS-Transaction define *port types* para o coordenador (*CoordinatorPortType*) e os participantes (*ParticipantPortType*). O *CoordinatorPortType*, implementado pelo coordenador, possui cinco operações: *Prepared*, *Abort*, *ReadOnly*, *Committed* e *Replay*. O *ParticipantPortType*, implementado pelo participante, possui três operações: *Prepare*, *Commit* e *RollBack*. As duas fases do protocolo, apresentadas na seção 2.1.2, são implementadas da seguinte forma pelo WS-Transaction[31]:

1. Votação: o coordenador começa uma transação atômica distribuída enviando mensagens *Prepare* ao *PartipantPortType* para todos os participantes registrados no protocolo *two-phase commit*. Então, os participantes respondem ao *CoordinatorPortType* uma das seguintes mensagens:

    - *Prepared*: o participante vota "sim" para a efetivação da transação e tornou duráveis, de modo não definitivo, as alterações dos seus dados.
    - *ReadOnly*: o participante vota "sim" para a efetivação da transação, porém não deseja participar da segunda fase do protocolo (efetivação).
    - *Aborted*: o participante vota "não" para a efetivação da transação e abortou as alteração dos seus dados (não efetivou as alterações).

2. Efetivação: é feita a apuração dos votos pelo coordenador. Se todos os participantes votaram "sim", ou seja, responderam *Prepared* ou *ReadOnly*, a transação é efetivada e o coordenador envia uma mensagem *Commit* ao *ParticipantPortType* dos participantes que responderam com a mensagem *Prepared* na primeira fase. Estes participantes respondem com a mensagem *Committed* ao *CoordinatorPortType* do coordenador, indicando que a transação foi efetivada nos seus dados. Caso algum dos participantes votou "não", ou seja, responderam *Aborted*, o coordenador envia uma mensagem *RollBack* ao *ParticipantPortType* de todos os participantes que votaram "sim" (que responderam com *Prepared*) na primeira fase. Estes participantes respondem ao *CoordinatorPortType* do coordenador com a mensagem *Aborted*, informando que ele abortou a transação em seus dados (desfez as alterações feitas de forma não definitiva na primeira fase).

A figura 3.5 ilustra o funcionamento do protocolo *two-phase commit*.

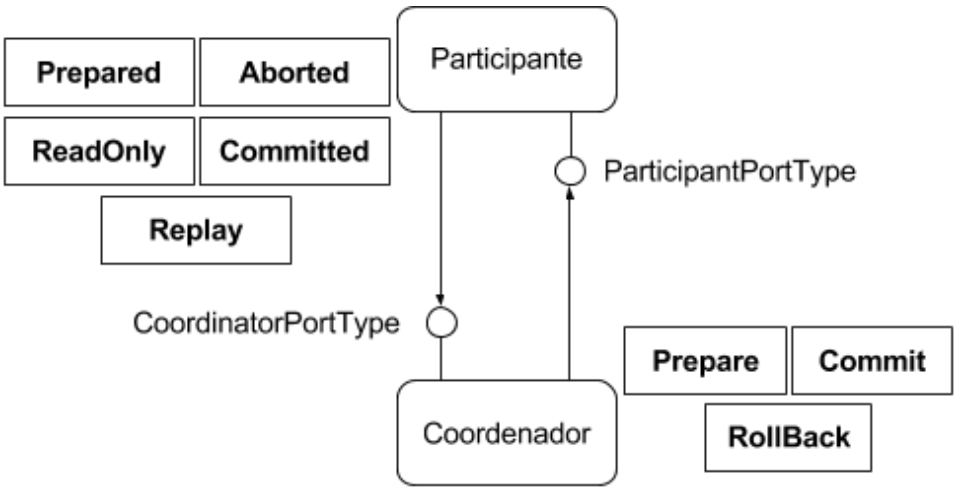

Figura 3.5: WS-Transaction: ilustração do protocolo *two-phase commit*.

3.2.2 Exemplo

Apresentados os protocolos WS-Coordination e WS-Transaction, esta subseção apresenta um exemplo do uso dos mesmos. A figura 3.6 mostra um diagrama de sequência, representando um iniciador, um coordenador e dois participantes[31].

Os passos da transação distribuída apresentados no diagrama são os seguintes:

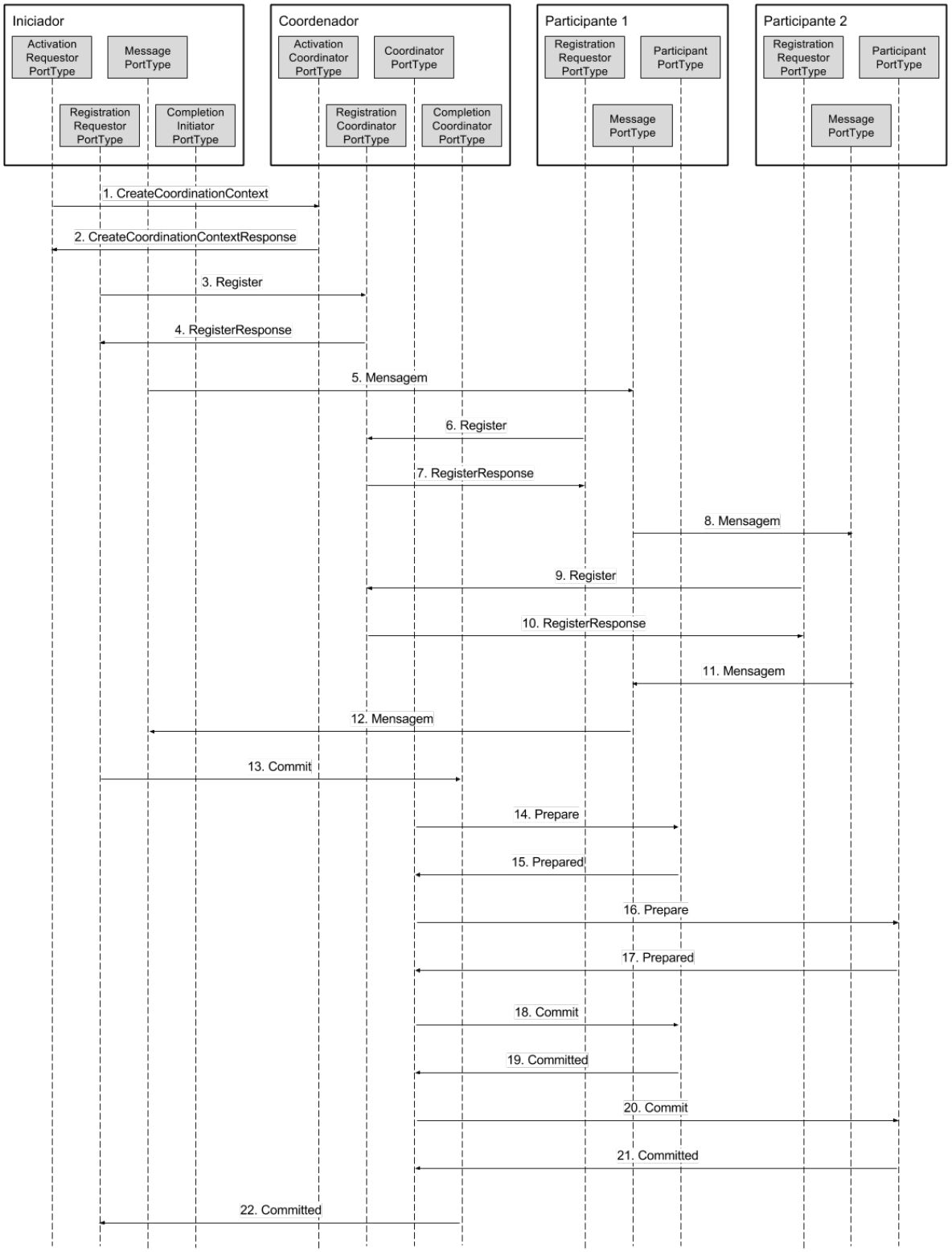

Figura 3.6: Exemplo de transação distribuída utilizando protocolo de coordenação.

1. Iniciador envia a mensagem *CreateCoordinationContext* ao coordenador.

2. Coordenador inicia uma transação, cria um novo contexto transacional e envia a mensagem *CreateCoordinationContextResponse* ao iniciador, contendo o contexto criado.

3. Iniciador envia a mensagem *Register* ao coordenador, se registrando no protocolo *completion*, que possibilita que o protocolo *two-phase commit* seja iniciado e que o iniciador obtenha seu resultado.

4. Coordenador envia resposta ao iniciador através da mensagem *RegisterResponse*, contendo o endereço do *CompletionCoordinatorPortType*, que será usado porteriormente.

5. Iniciador envia mensagem ao participante 1, com os dados necessários para efetuar a transação no mesmo. Como a mensagem está sendo enviada dentro de uma transação, o contexto transacional também é enviado.

6. Participante 1 envia a mensagem *Register* ao coordenador, se registrando no protocolo *two-phase commit*.

7. Coordenador registra o participante 1 no protocolo *two-phase commit* e lhe envia a mensagem *RegisterResponde*, contendo o endereço do seu *CoordinatorPortType*.

8. Participante 2 envia mensagem ao participante 2, com os dados necessários para efetuar a transação no mesmo. Assim como a mensagem cinco, o contexto transacional também é enviado.

9. Participante 2 envia a mensagem *Register* ao coordenador, se registrando no protocolo *two-phase commit*.

10. Coordenador registra o participante 2 no protocolo *two-phase commit* e lhe envia a mensagem *RegisterResponse*, contendo o endereço do seu *CoordinatorPortType*.

11. Participante 2 envia o resultado das operações executadas ao participante 1.

12. Participante 1 envia o resultado das operações executadas ao iniciador.

13. Iniciador envia a mensagem *Commit* ao *CompletionCoordinationPortType* do coordenador para iniciar o protocolo *two-phase commit*.

14. Coordenador envia a mensagem *Prepare* para o participante 1, iniciando o *two-phase commit*.

15. Participante 1 responde ao coordenador com a mensagem *Prepared*, votando "sim" para a efetivação da transação.

16. Coordenador envia a mensagem *Prepare* para o participante 2, prosseguindo a execução do *two-phase commit*.

17. Participante 2 responde ao coordenador com a mensagem *Prepared*, votando "sim" para a efetivação da transação.

18. Após fazer a apuração dos votos, o coordenador verifica que todos os participantes votaram "sim" (segunda fase do *two-phase commit*) e envia a mensagem *Commit*, primeiramente ao participante 1.

19. Participante 1 efetiva sua parte da transação e responde com a mensagem *Committed* ao coordenador.

20. Coordenador continua a execução do protocolo, enviando a mensagem *Commit* ao participante 2.

21. Participante 2 efetiva sua parte da transação e responde com a mensagem *Committed* ao coordenador.

22. Após a conclusão da execução do *two-phase commit*, a transação distribuída foi concluída e o coordenador avisa ao iniciador, através da mensagem *Committed*.

CAPÍTULO 4

EXEMPLO PRÁTICO

Inicialmente, para exemplificar o uso de transações em web services, foram criados três sistemas:

- Banco: possui contas, cada uma pertence a um titular, que pode possuir diversos cartões. Disponibiliza um web service para a compra de produtos, que desconta a quantia determinada da conta de um titular, através do uso de um cartão.

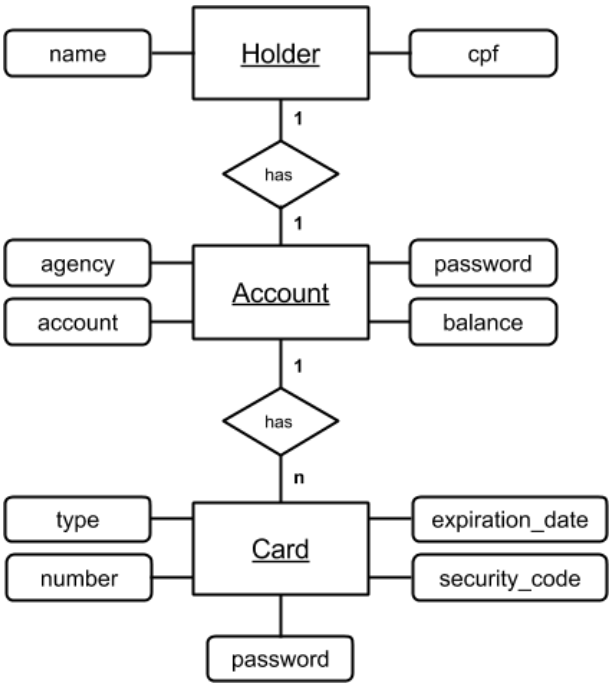

Figura 4.1: Modelo entidade relacionamento do sistema bancário.

- Fornecedor: fornece produtos para a loja e disponibiliza um web service para acessar seus produtos. O web service fornece métodos para obter todos os produtos da loja ou um produto especifico, com todas as suas informações;

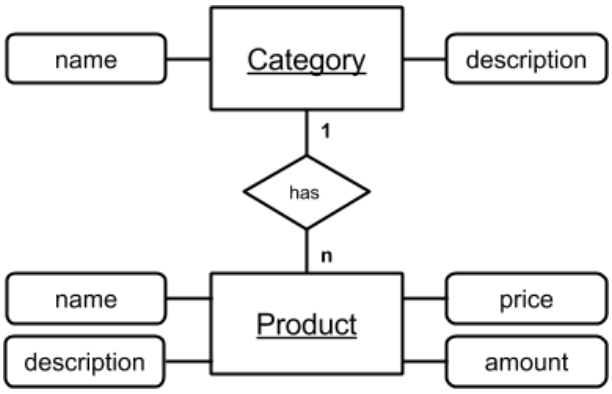

Figura 4.2: Modelo entidade relacionamento do fornecedor.

- Loja: vende produtos de diversos fornecedores aos usuários, cobrando uma taxa porcentual acima do preço normal do produto. Contém um cadastro de todos os fornecedores e bancos disponíveis para utilizar seus web services, além de coordenar toda a lógica da compra.

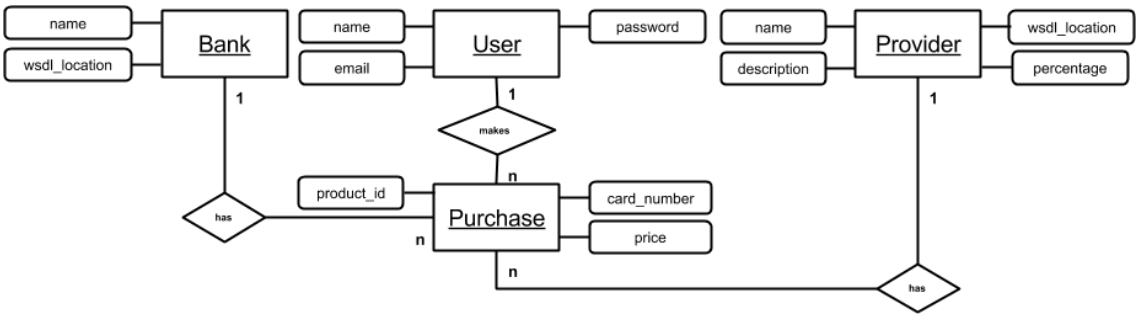

Figura 4.3: Modelo entidade relacionamento da loja.

4.1 BIBLIOTECAS UTILIZADAS

Nesta seção serão apresentadas as gemas utilizadas para a criação e utilização de um web service SOAP.

4.1.1 WashOut

WashOut é uma gema que simplifica a criação de um web service SOAP. A biblioteca funciona a partir da versão 3.0 do rails e permite criar um WSDL com a localização e informações dos métodos de um determinado controlador[29]. Apesar de oferecer essa

praticidade, esta gema não tem suporte a extensões como o WS-Coordination e WS-Transaction.

4.1.2 Savon

Savon é um cliente SOAP para a linguagem Ruby[15], que possibilita chamar e localizar métodos de um servidor através do seu WSDL.

4.2 OPERAÇÕES SIMPLES

Com o intuito de analisar os problemas envolvendo operações simples (sem o uso de transação), os sistemas foram desenvolvidos inicialmente sem os protocolos de coordenação citados na seção 3.

Assim como o exemplo da Amazon (seção 2.2.2), o único passo do problema apresentado que utiliza transações é a compra de um produto. Assim, após um usuário clicar em comprar na página de um produto, a loja deve verificar se há estoque do produto no fornecedor, descontar o valor do produto na conta associada ao cartão digitado e, por fim, diminuir o número disponível no estoque do fornecedor. Para desenvolver o sistema da loja utilizando operações simples, os seguintes passos foram seguidos:

1. Obter todas as informações do produto através do web service do fornecedor;
2. Se o produto estiver disponível no fornecedor (quantidade disponível maior que zero), deve-se utilizar o método de compra do banco, passando as informações digitadas pelo usuário. O sistema bancário então valida esses dados e, se tudo estiver correto e a conta associada ao cartão tiver saldo disponível, a quantia é descontada da conta do usuário;
3. Após a efetivação da compra, deve-se diminuir a quantidade do produto comprado no fornecedor.

Como é possível observar, os passos acima devem ser executados de forma atômica sobre um produto de determinada fornecedora. Para mostrar a necessidade disso, foram

desenvolvidas duas soluções, apresentadas a seguir.

4.2.1 Sem Região Crítica

Nesta solução, os passos citados anteriormente são executados sem nenhuma área de exclusão mútua, técnica que evita que dois processos tenham acesso simultaneamente a um recurso compartilhado[5]. Isso possibilita que uma compra seja efetuada mesmo que não haja produtos em estoque no fornecedor. Neste caso em específico, não há um isolamento das transações (sincronização) e haverá inconsistência no banco de dados do sistema bancário, já que uma compra será efetivada, mesmo não havendo o produto no fornecedor. A figura 4.4 apresenta o diagrama de sequência referente ao caso de uso comprar produto.

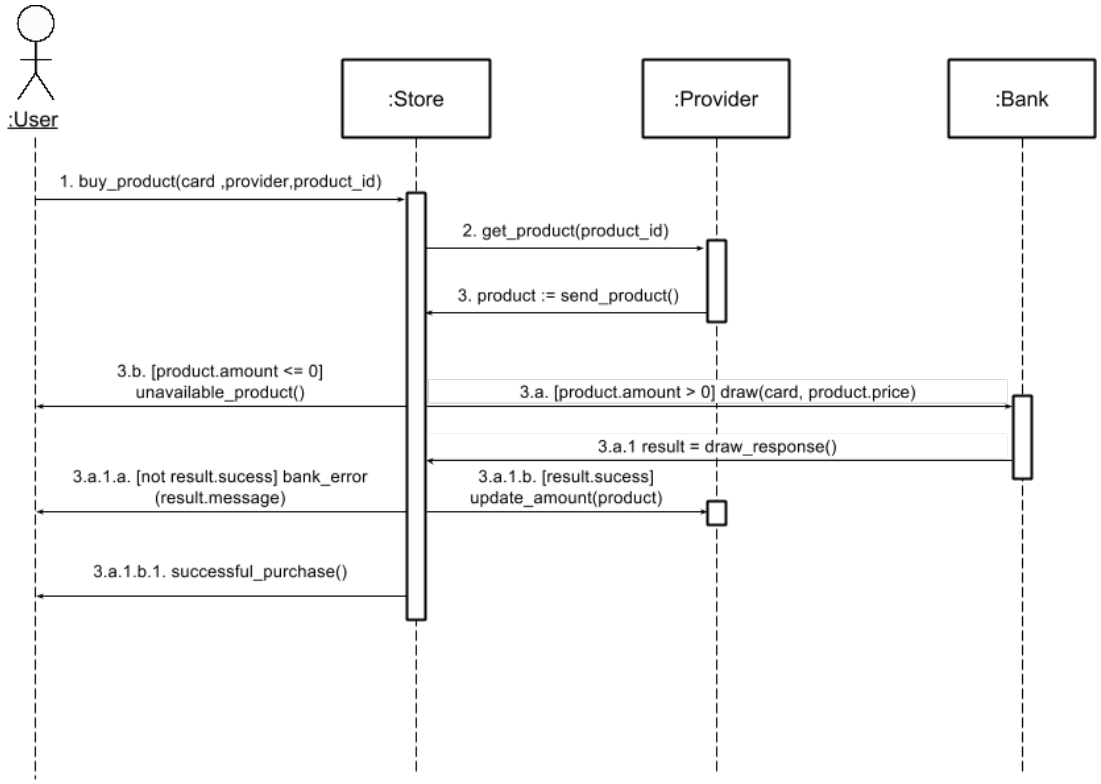

Figura 4.4: Diagrama de sequência que representa a execução dos passos de uma compra sem região crítica.

Para simular o problema desta implementação, considere o exemplo a seguir. Dois usuários compram um mesmo produto fornecido pela mesma fornecedora, que possui apenas um deste em estoque. Considere que o usuário 1 tem conta no banco A (e que

possui saldo suficiente para comprar o produto), que o usuário 2 tem conta no banco B (e que possui saldo suficiente para comprar o produto), que a conexão do banco A seja praticamente instantânea e que a conexão com o banco B leve cerca de dez segundos. Assim, quando o usuário 2 clica em comprar, o sistema obtém as informações do produto para verificar se há o produto em estoque. Como há um produto em estoque, o sistema se comunica com o banco, para verificar se há saldo disponível e se os dados do cartão estão corretos. Como a requisição ao banco B é lenta, a efetivação da compra também demora. Neste intervalo de tempo, o usuário 1 também clica em comprar e o sistema obtém os dados do produto novamente. Como a quantidade ainda não foi diminuída (a transação do usuário 2 não foi efetivada), o sistema desconta o valor do produto na conta do usuário 1 no banco A e então atualiza a quantidade no fornecedor, diminuindo a quantia para zero. Assim, o usuário 1 recebe a mensagem de que a compra foi efetivada. Enquanto isso, o usuário 2 aguarda a confirmação do banco B que, ao ser recebida pelo sistema, diminui a quantidade deste produto no fornecedor. Entretanto, as duas compras são efetivadas e o valor é descontado da conta de ambos os usuários sendo que só há apenas um produto disponível, gerando uma inconsistência.

4.2.2 Com Região Crítica

Neste caso, os passos da compra são executados em uma área de exclusão mútua utilizando um conjunto de semáforos, apresentados na seção 2.3.10. Assim, apenas uma compra por vez de um determinado produto de um fornecedor pode acessar o trecho do código que valida a compra. A inclusão do semáforo binário (mutex) é apresentada na figura 4.5, em vermelho.

O conjunto de semáforos atua sobre um produto de determinado fornecedor e permite que apenas uma requisição seja executada neste trecho de código. Assim, se outro usuário tentar efetuar uma compra enquanto outra esta sendo efetivada, sua requisição deve esperar até que a requisição que está utilizando a região crítica termine. Para implementar essa solução precisamos de um identificador único para o fornecedor e para o produto. No rails, temos o id de cada um deles sendo único: o fornecedor tem seu id na loja e

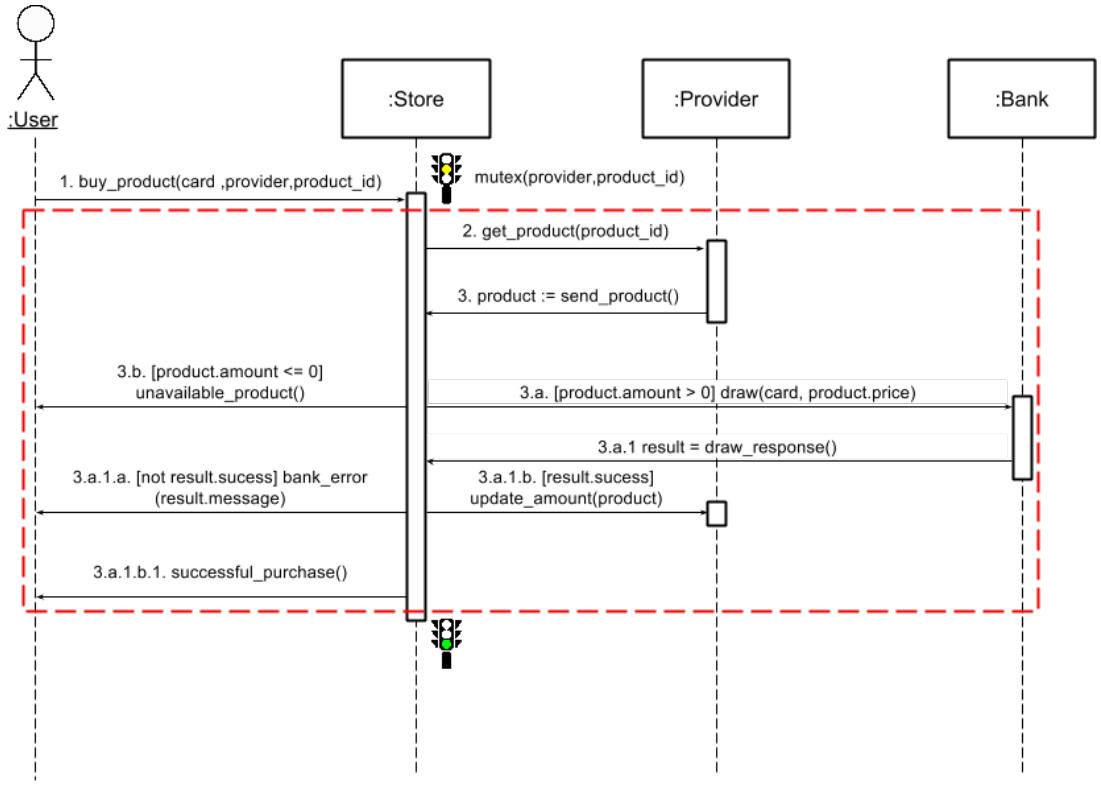

Figura 4.5: Diagrama de sequência que representa a execução dos passos de uma compra com região crítica.

o produto tem um id no fornecedor. Logo a tupla "id do fornecedor" e "id do produto" formam uma chave para uma hash de semáforos, que é acessada pelo método de compra quando o usuário clica em comprar e bloqueia o acesso as outras requisições neste produto deste fornecedor. A figura 4.6 demonstra a utilização da hash de semáforos binários por múltiplas threads (várias requisições). Cada linha da região crítica representa um produto de determinado fornecedor, identificado pela chave "id do fornecedor" e "id do produto". As threads que tentarem acessar uma região que está em uso (lock) devem esperar (wait) até que a mesma seja liberada (unlock).

A garantia de exclusão mútua é feita através da classe *Mutex*, nativa do Ruby, que implementa um semáforo simples e pode ser usada para coordenar acesso à recursos compartilhados por mútiplas threads concorrentes. A hash de semáforos é uma variável global, compartilhada entre todas as threads da aplicação. Uma requisição de um produto p de um fornecedor f é feita da seguinte forma:

1. Checa se existe mutex criado para *f-p*. Se não existir, cria uma chave *f-p* na hash

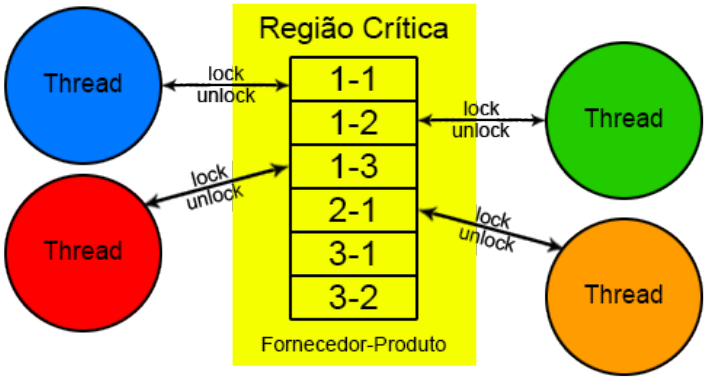

Figura 4.6: Diagrama que representa a hash de semáforos.

de semáforos e aloca um novo mutex para acessar este produto.

2. Utiliza o método synchronize no mutex *f-p*, que verifica se a região crítica deste mutex está em uso. Se está, espera até que ela seja liberada, caso contrário, bloqueia a região crítica (lock) e executa a transação. Ao fim, a região crítica é liberada (unlock)

Embora a utilização de semáforos tenha resolvido o problema de sincronização e consistência para o caso apresentado, esta solução não resolve problemas de quedas e perdas de conexão que envolvem sistemas distribuídos. Para apresentar estes problemas considere as mensagens apresentadas na figura 4.5 e verifique as consequências de uma queda em cada uma delas. Começando pelas mensagens 2 e 3, é possível concluir que uma queda da loja ou do fornecedor durante a troca dessas duas mensagens não deve afetar nenhum dos sistemas, uma vez que se trata apenas de uma operação de leitura. No caso de queda do fornecedor, quando a loja envia a mensagem 2, deve haver timeout e a loja deve mostrar uma mensagem de erro ao usuário. O mesmo ocorre ao fornecedor ao tentar enviar a mensagem 3, em caso de queda da loja. Caso o banco caia, a mensagem 3.a. não deve ser recebida e a loja deve mostrar uma mensagem de erro ao usuário, após o timeout. Já se a loja cair durante a mensagem 3.a.1., o banco terá descontado da conta do usuário a quantia referente ao valor do produto e o sistema da loja não irá prosseguir com os passos da transação, a compra não será efetivada (e nem a quantia será atualizada no fornecedor), gerando uma inconsistência. O mesmo problema ocorre caso o fornecedor

caia durante a mensagem 3.a.1.b.. Neste caso, a quantia do produto não será atualizada e a compra será efetivada, gerando inconsistência no banco do fornecedor.

4.3 PROTOCOLOS DE COORDENAÇÃO

Para resolver os problemas apresentados nas soluções anteriores, foram utilizados os protocolos apresentados na seção 3 (WS-Coordination e WS-Transaction). Para tal, foi necessário criar um novo sistema, o coordenador, responsável pelo gerenciamento das transações envolvendo os demais sistemas. Este sistema possui apenas as classes necessárias para o funcionamento do protocolo: *Activation Coordinator PortType*, *RegistrationCoordinatorPortType*, *CompletionCoordinatorPortType* e *CoordinatorPortType*, além de uma página que permite visualizar os contextos de transação criados no momento. Por conta da complexidade do problema envolvendo os quatro sistemas, o diagrama de sequência desta solução foi dividida em cinco partes, cada uma referente a uma fase do protocolo de coordenação, nas subseções a seguir.

Vale lembrar que as biblioteca WashOut[29] e Savon[15] não possuem suporte aos protocolos de coordenação. Por conta disso, as informações dos contextos de coordenação são enviadas no corpo da mensagem (*body*) e não no cabeçalho (*header*), como citado no capítulo 3. Além disso, ainda não há um sistema de recuperação que lê os logs gerados pelo algoritmo após uma queda.

4.3.1 Ativação

Quando o cliente decide comprar determinado produto e informa os dados do cartão, o controlador *ProductsController*, responsável pela interação da loja com o usuário, envia uma mensagem ao *ActivationRequestorPortType*, que envia a mensagem *create_coordination_context* ao *ActivationCoordinatorPortType* do coordenador, informando o tipo de coordenação a ser criado, no caso, o WS-Transaction. Então, o coordenador cria um novo contexto de coordenação e o envia para a loja. Ao final da fase de ativação, ambos os sistemas (coordenador e loja) terão as informações referentes ao contexto de

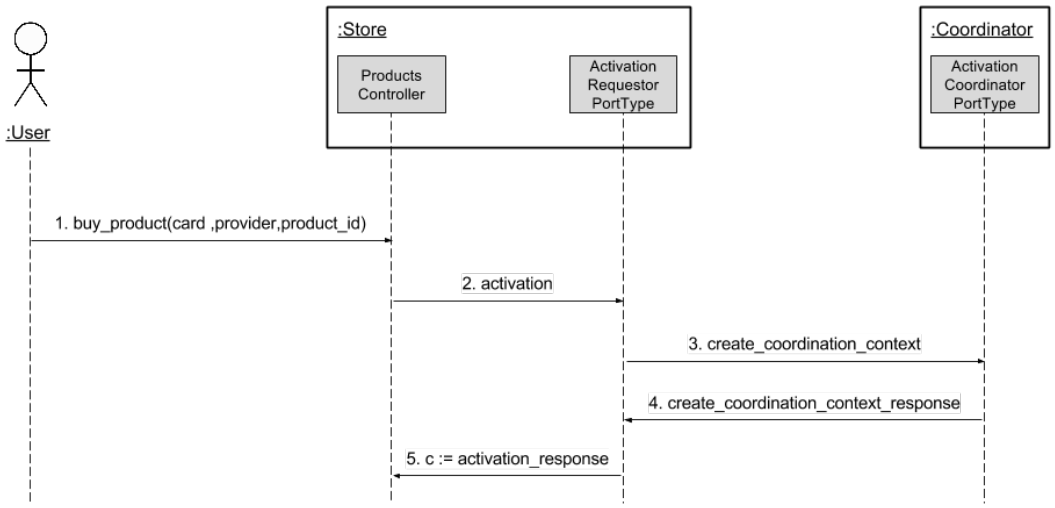

Figura 4.7: Diagrama de sequência da fase de ativação do WS-Coordination.

coordenação criado. Vale lembrar que, além das informações do contexto de coordenação criado, o coordenador também envia à loja o endereço do serviço de registro, armazenado localmente no contexto de coordenação da loja e utilizado na próxima fase. A figura 4.7 apresenta o diagrama de sequência referente à fase de ativação e a tabela 4.1 representa os dados informados pelo coordenador após a execução desta fase.

id	**coordination type**	**completion participants**	**2pc participants**
31718520	ws-t	none	none

Tabela 4.1: Contexto de coordenação criado na fase de ativação.

Os códigos A.1 e A.2 ilustram o funcionamento do participante e do coordenador, conforme a figura 4.7. Primeiramente, o participante (código A.1) - a loja - se comunica com o *ActivationCoordinatorPortType* do coordenador, através do endereço do seu WSDL, na linha 4, utilizando a biblioteca Savon, apresentada na seção 4.1.2. Em seguida, a loja chama o método *create_coordination_context*, localizado no coordenador. Então, o coordenador (código A.2) cria um novo contexto de coordenação do tipo WS-Transaction (*ws-t*), na linha 5 e escreve as informações destes contexto em um log, na linha 6. Na linha 7, ele adiciona o mesmo a uma variável global, uma hash cuja chave é o identificador do contexto, que reúne todos os contextos de coordenação criados. Na linha 8, utilizando a biblioteca WashOut, apresentada na seção 4.1.1, o coordenador envia as informações do

contexto de coordenação criado (identificador e tipo de coordenação), incluindo o endereço do serviço de registro. Ao receber as informações, o participante transforma o retorno do método chamado em uma hash para obter os dados do contexto de coordenação (linha 6 do código A.1). O participante então cria um novo contexto de coordenação local, utilizando os dados da hash criada (linha 7) e retorna o contexto de coordenação criado (linha 8).

4.3.2 Registro no completion

Com o endereço do serviço de registro do coordenador, a loja envia a mensagem *register* ao *RegistrationCoordinatorPortType* do coordenador, informando que quer se registrar no protocolo *completion*. O coordenador, por sua vez, registra a loja como participante do protocolo *completion* no contexto de coordenação criado anteriormente. Ao enviar a resposta à loja, o coordenador também envia o endereço do serviço do protocolo *completion*, utilizado para inicializar o *two-phase commit*. A figura 4.8 apresenta o diagrama de

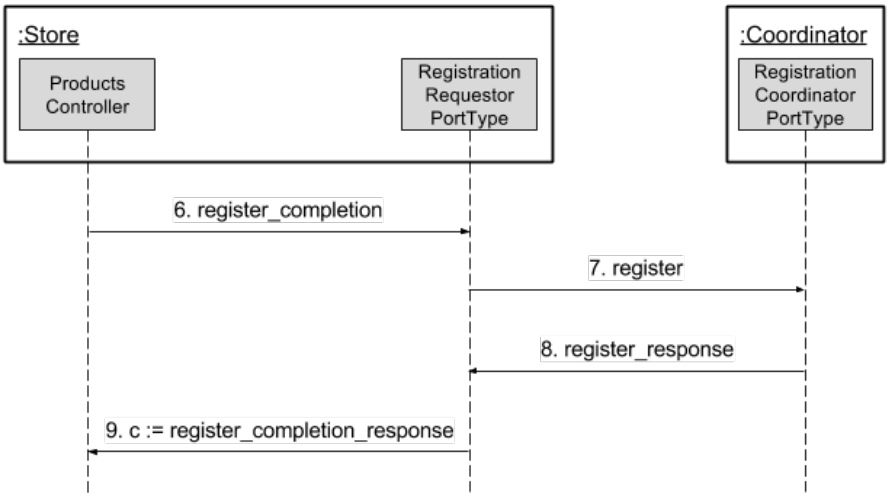

Figura 4.8: Diagrama.

sequência desta fase e a tabela 4.2 representa o contexto de coordenação do coordenador após a execução desta fase.

Os códigos A.3 e A.4 ilustram o funcionamento do *RegistrationRequestorPortType* e *RegistrationCoordinatorPortType*, respectivamente, apresentados na figura 4.8. Primeiramente, a loja conecta-se ao serviço de registro do coordenador e executa o método

id	coordination type	completion participants	2pc participants
31718520	ws-t	`localhost:3000/completion_initiator_port_type/wsdl`	none

Tabela 4.2: Contexto de coordenação criado na fase de registro da loja no protocolo *completion*.

register, nas linhas 3 e 4 do código A.3, passando como parâmetro o contexto de coordenação referente à transação, o protocolo de coordenação que ele quer se registrar (*completion*) e o endereço do seu *CompletionInitiatorPortType*. O coordenaador, por sua vez, procura pelo contexto de coordenação, na linha 5 do código A.4, através do identificador do contexto enviado pelo iniciador. Se o protocolo de coordenação escolhido for o *completion*, o endereço do *CompletionInitiatorPortType* é adicionado aos participantes deste protocolo no contexto de coordenação do coordenador (linha 6). Caso o protocolo escolhido seja o *two-phase commit*, o endereço do *ParticipantPortType* é adicionado aos participantes deste protocolo no contexto de coordenação do coordenador. Em ambos os casos, o coordenador responde com o contexto de coordenação e o endereço do protocolo escolhido (CompletionCoordinatorPortType, no *completion*, ou CoordinatorPortType, no *two-phase commit*, nas linhas 9 e 19.

4.3.3 Registro no two-phase commit

Após o registro da loja, é necessário registrar os demais sistemas para que a transação possa continuar. Os dois sistemas restantes (fornecedor e banco) serão os participantes do protocolo *two-phase commit*, responsável por verificar se a transação pode ou não ser efetivada. Para que o registro seja possível, ambos os sistemas devem conhecer o endereço de registro do coordenador, o identificador do contexto de coordenação referente à transação desejada e ter as informações de negócio referente à transação. Para isso, a loja envia uma mensagem de registro para cada um dos sistemas, contendo essas informações.

Primeiramente, a loja envia a mensagem *send_product* ao *MessagePortType* do fornecedor, contendo os dados do contexto de coordenação criado e o identificador do produto que o cliente deseja comprar. Com o identificador do contexto de coordenação e o endereço do serviço de registro, o fornecedor envia a mensagem *register* ao *Registration-*

CoordinatorPortType, informando que quer se registrar no protocolo *two-phase commit*. O coordenador então registra o fornecedor como participante do *two-phase commit* no contexto de coordenação correspondente ao identificador recebido e responde, informando a localização do serviço do protocolo (*CoordinatorPortType*). O fornecedor armazena no seu contexto de coordenação local o identificador do produto, assim como o endereço do *CoordinatorPortType* recebido, utilizado na execução do *two-phase commit*. A figura 4.9 apresenta o diagrama de sequência do registro do fornecedor no *two-phase commit* e a tabela 4.3 mostra o contexto de coordenação logo após o registro do fornecedor.

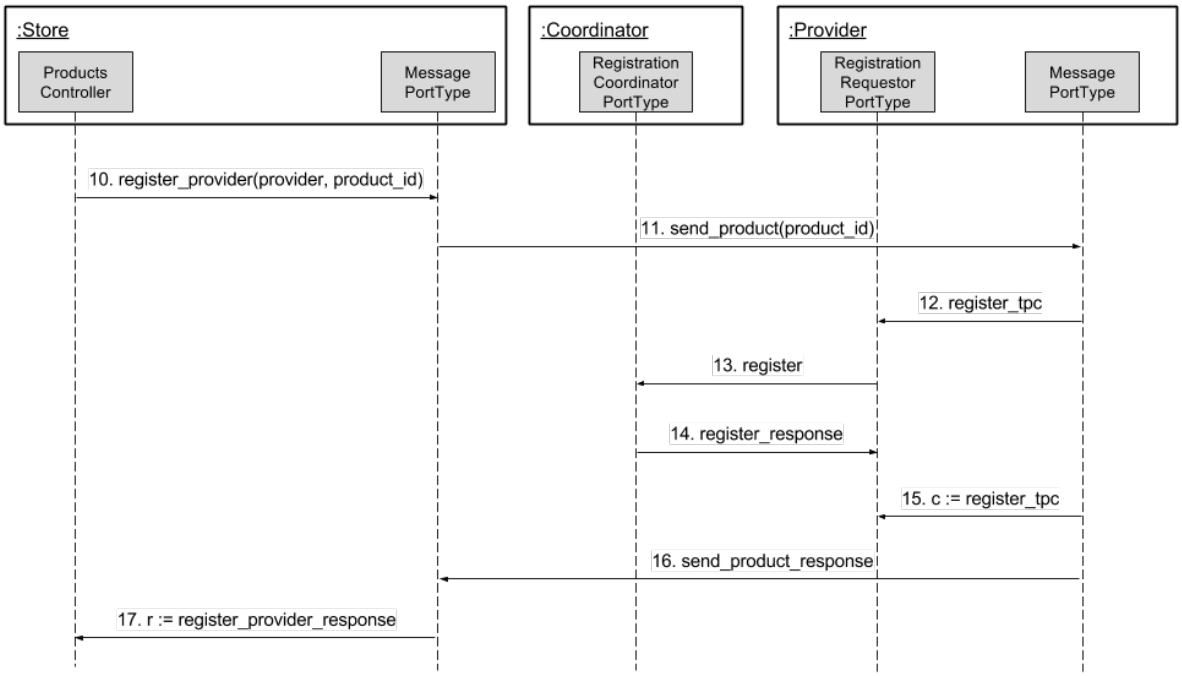

Figura 4.9: Diagrama: registro do fornecedor no *two-phase commit*.

id	coordination type	completion participants	2pc participants
31718520	ws-t	`localhost:3000/completion_initiator_port_type/wsdl`	`localhost:3001/participant_port_type/wsdl`

Tabela 4.3: Contexto de coordenação após o registro do fornecedor.

Os códigos A.5 e A.6 ilustram o funcionamento dos *MessagePortTypes* da loja e do fornecedor, apresentados na figura 4.9. Inicialmente, a loja procura pelo fornecedor em seu banco de dados (linha 3 do código A.5) e conecta-se ao seu *MessagePortType*, através do seu WSDL (linha 4). Em seguida, a loja executa o método *send_product* do forne-

cedor, passando o contexto de coordenação como parâmetro, bem como o identificador do produto que o usuário deseja comprar (linha 5). O fornecedor, por sua vez, cria um novo contexto de coordenação local, com as informações enviadas pela loja (identificador, tipo de coordenação, endereço do serviço de registro e identificador do produto) e escreve um log local, cujo nome é o mesmo do identificador do contexto de coordenação, com as informações do contexto de coordenação (linhas 9 e 10 do código A.6). O código A.7 ilustra o funcionamento do *RegistrationRequestorPortType* do fornedor, semelhante ao código A.3, referente ao *RegistrationRequestorPortType* da loja. A diferença está na adição do contexto de coordenação a uma variável global, uma hash, da mesma forma feita no coordenador, no código A.2. Isso é necessário para que o fornecedor saiba sobre qual transação se trata no *two-phase commit*.

Após o registro do fornecedor, é necessário registrar o banco. Para isso, a mesma lógica é seguida: a loja envia a mensagem *send_ card* ao banco, contendo os dados do contexto de coordenação, do cartão (número, senha ecódigo de segurança) e do produto a ser comprado (valor do produto). Através do endereço do serviço de registro recebido, o banco envia a mensagem *register* ao *RegistrationCoordinatorPortType* do coordenador, contendo o identificador do contexto de coordenação e o protocolo ao qual quer se registrar (*two-phase commit*). Ao receber a mensagem, o coordenador registra o banco no contexto de coordenação e responde com o endereço do *CoordinatorPortType*, o mesmo enviado ao fornecedor. Em seguida, o banco salva em seu contexto de coordenação local o endereço do *CoordinatorPortType* do coordenador e os dados do cartão que o cliente forneceu, além do valor do produto desejado. A figura 4.10 apresenta o diagrama de sequência do registro do banco no *two-phase commit* e a tabela 4.4 mostra o contexto de coordenação logo após o registro do fornecedor, apresentando o endereço do *ParticipantPortType* do fornecedor e do banco como participantes do protocolo.

id	**coordination type**	**completion participants**	**2pc participants**
31718520	ws-t	`localhost:3000/completion_initiator_port_type/wsdl`	`localhost:3001/participant_port_type/wsdl` `localhost:3002/participant_port_type/wsdl`

Tabela 4.4: Contexto de coordenação após o registro do banco.

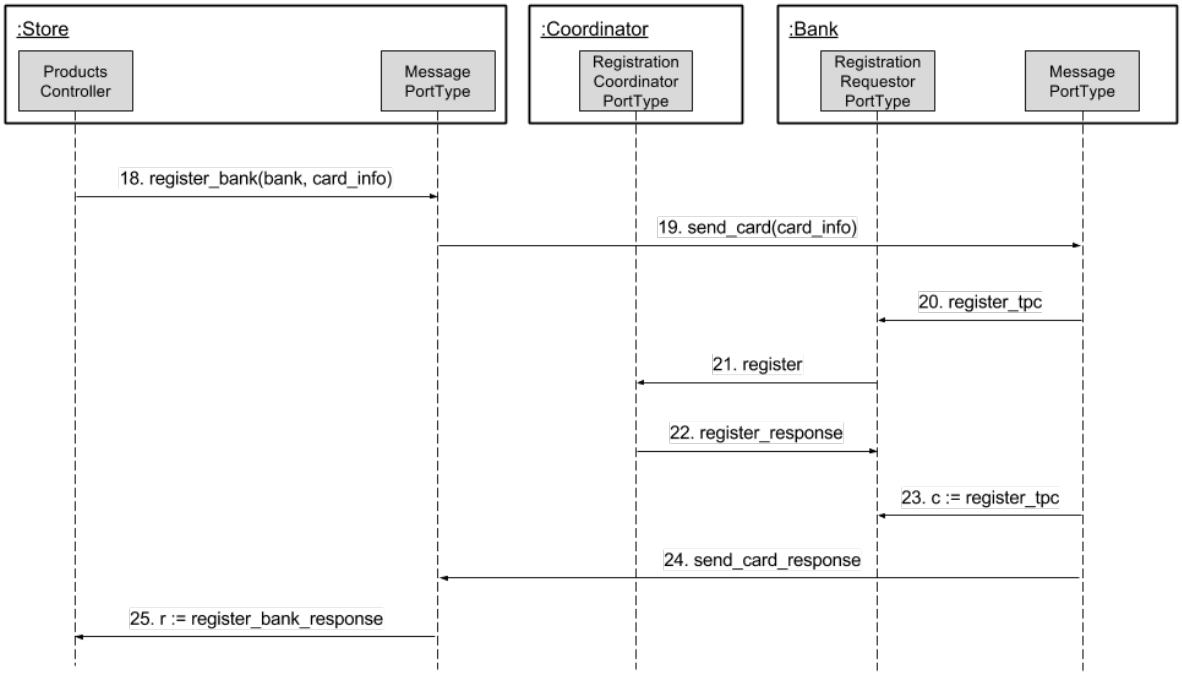

Figura 4.10: Diagrama: registro do banco no *two-phase commit*.

Os códigos A.8 e A.9 ilustram o funcionamento dos *MessagePortTypes* da loja e do banco, apresentados na figura 4.10. Assim como o fornecedor, a loja procura pelo banco em seu banco de dados (linha 6 do código A.5) e conecta-se ao seu *MessagePortType*, através do seu WSDL (linha 7). Em seguida, a loja executa o método *send_ card* do banco, passando o contexto de coordenação como parâmetro, bem como os dados do cartão que o usuário informou (linha 8). O banco, por sua vez, cria um novo contexto de coordenação local, com as informações enviadas pela loja (identificador, tipo de coordenação, endereço do serviço de registro, número do cartão, senha, número de segurança e valor do produto) e escreve um log local, cujo nome é o mesmo do identificador do contexto de coordenação, com as informações do contexto de coordenação (linhas 12 e 13 do código A.9). O código A.10 ilustra o funcionamento do *RegistrationRequestorPortType* do banco, semelhante ao registro do fornecedor, com a adição do contexto de coordenação em uma variável global, uma hash, na linha 17.

4.3.4 Completion

Após o cadastro do fornecedor e do banco, a loja manda a mensagem *commit* ao *CompletionCoordinatorPortType*, informando ao coordenador que ele deve iniciar a execução do *two-phase commit* no contexto de coordenação enviado. O coordenador então executa o protocolo e responde ao *CompletionInitiatorPortType* da loja com seu resultado, apenas (*committed* ou *aborted*). O objetivo do protocolo *completion* é basicamente avisar ao coordenador quando todos os participantes estão registrados, para que ele possa iniciar o *two-phase commit* e obter o resultado da transação. Além disso, caso haja alguma queda ou problema no processo de registro de um participante, a loja pode avisar ao coordenador que houve um problema e que deseja abortar a transação. Neste caso, o protocolo *two-phase commit* não é inicializado e os contextos de coordenação são excluídos. A figura 4.11 apresenta o diagrama de sequência do protocolo *completion*. Os passos omitidos representam o *two-phase commit*, apresentado na seção a seguir.

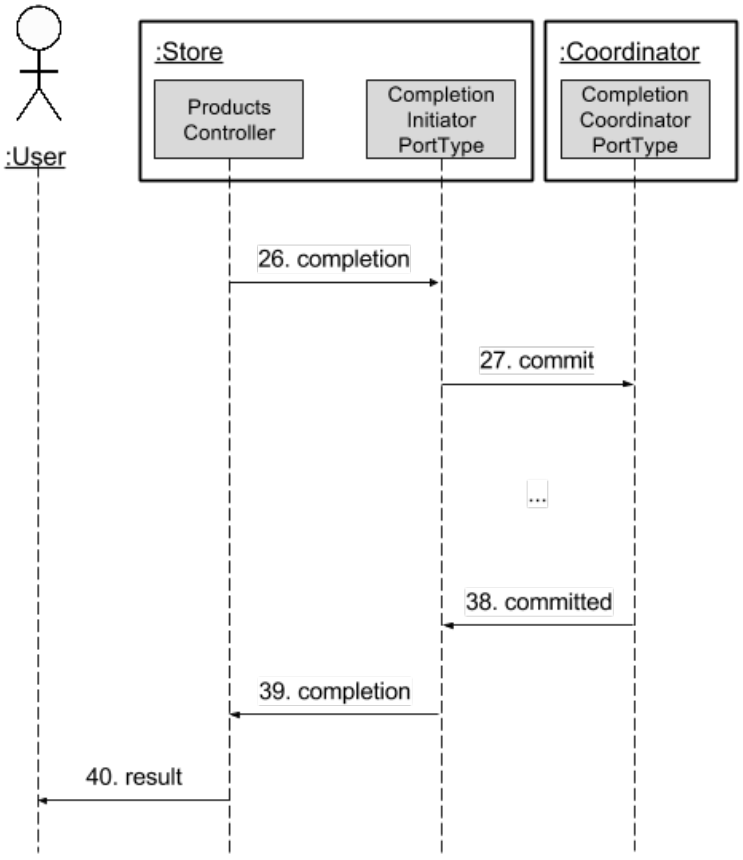

Figura 4.11: Diagrama: execução do protocolo *completion*.

Os códigos A.11 e A.12 ilustram o funcionamento do protocolo *completion* no caso apresentado pela figura 4.11. Considerando que não houve nenhum problema no registro de todos os participantes, a loja executa o método *commit* do *CompletionCoordinator-PortType* do coordenador, na linha 4 do código A.11, passando como parâmetro o contexto de coordenação. O coordenador, então, procura pelo contexto de coordenação, na linha 5, e inicia o protocolo *two-phase commit*, na linha 6. Nas linhas 7 e 8, o coordenador apaga o contexto de coordenação e envia a resposta do *two-phase commit* para a loja, na linha 9. Ao receber o resultado, o método *commit_response* retorna o resultado da transação, na linha 11 do código A.11.

4.3.5 Two-phase commit

Após a loja indicar que todos os participantes foram registrados, o coordenador pode iniciar a execução do protocolo *two-phase commit*. Primeiramente, na fase de votação, o coordenador envia a mensagem *prepared* para todos os participantes cadastrados no protocolo. No caso, ele envia ao fornecedor e ao banco, que decidem se desejam ou não efetivar a transação em seus sistemas. Para ambos os casos, foram criadas novas variáveis utilizadas especialmente nestas transações. No fornecedor, foi adicionado à tabela de produtos o campo "transaction_amount", uma cópia do campo "amount", que representa a quantidade de produtos disponíveis. Na fase de votação, o fornecedor verifica se o campo "transaction_amount" é maior que zero. Se for, esta quantia é diminuída em um e é salva no banco de dados, não afetando a quantidade original do produto. Neste caso, o fornecedor envia a mensagem *prepared* ao coordenador, avisando que vota "sim" para a efetivação da transação. Caso contrário, ele envia a mensagem *aborted* e não efetua nenhuma alteração. Já no banco, foi adicionado à tabela de contas o campo "transaction_balance", uma cópia do campo "balance", que representa o saldo da conta de um cliente. Nesta fase, o banco verifica se o campo "transaction_balance" é maior ou igual ao preço do produto. Se for, a quantia é diminuída deste saldo, salva no banco de dados e a resposta *prepared* é enviada ao coordenador, votando "sim" para a efetivação da transação. Caso contrário, a resposta *aborted* é enviada, votando "não" e nenhuma alteração é feita.

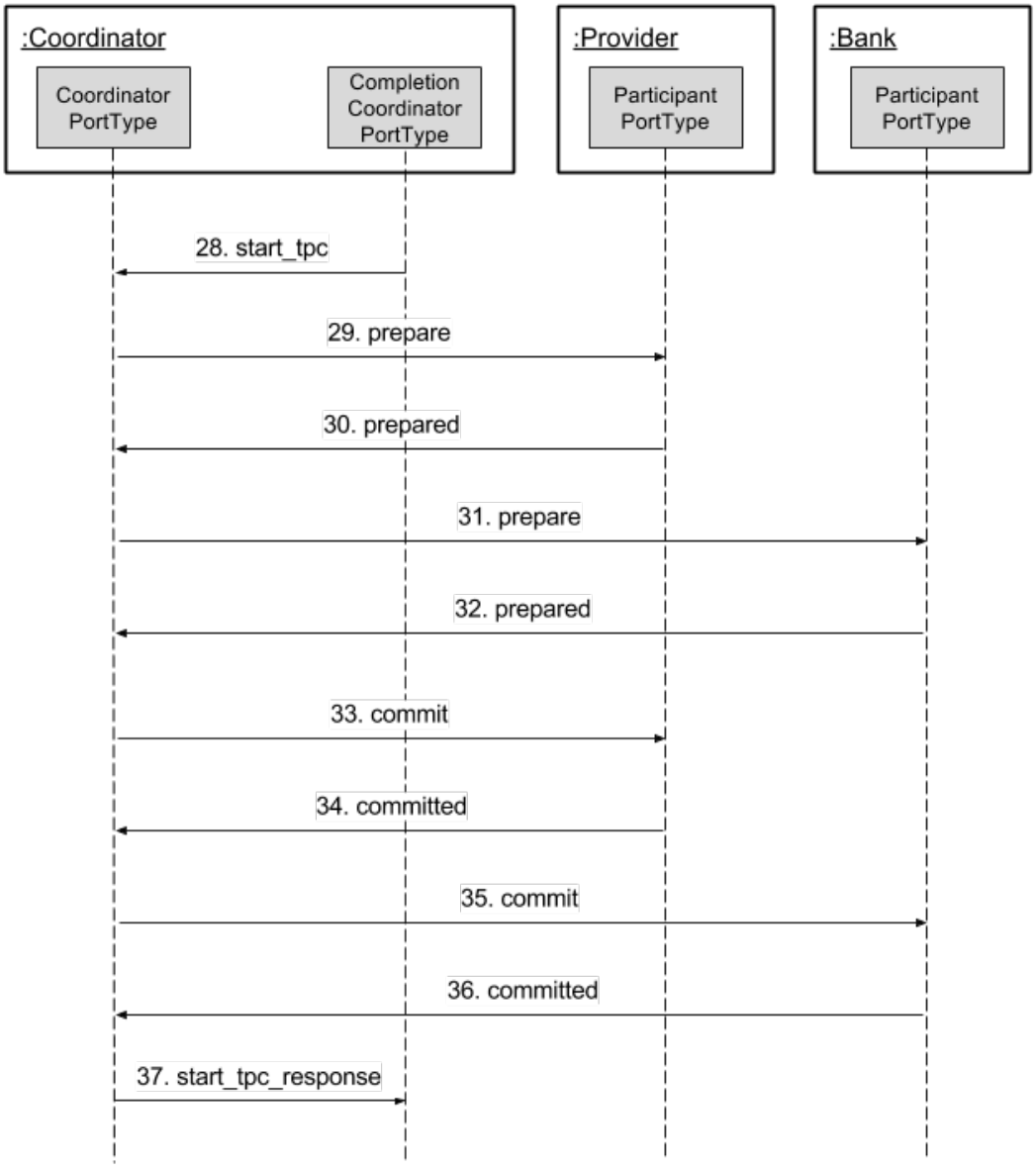

Figura 4.12: Diagrama: execução do protocolo *completion*.

Ao mesmo tempo que isso acontece, um log é gravado com os dados da transação, rotulado pelo identificador do contexto de transação, em ambos os sistemas. Este log possibilita que os valores das variáveis "transaction_amount" e "transaction_balance" sejam restaurados em caso de queda de algum destes sistemas.

Na fase de efetivação, o coordenador faz a apuração dos votos. Se todos votaram "sim", o coordenador envia a mensagem *commit* para todos os participantes, efetivando a transação em ambos os sistemas. Neste caso, os atributos "amount" e "balance" são atualizados no fornecedor e no banco, respectivamente. A figura 4.12 apresenta o diagrama

de sequência deste caso, efetivando a transação. Caso algum dos participantes vote "não", o coordenador envia a mensagem *rollback* aos participantes que votaram "sim", fazendo com que a transação não seja efetivada. No problema apresentado, a operação rollback efetua a operação inversa nos atributos "transaction_amount" e "transaction_balance", incrementando a quantia de produtos e o saldo disponíveis para transação (restaurando o valor antigo). Assim como na fase anterior, o resultado das operações são salvos no mesmo log, identificado pelo contexto de coordenação. Após a execução do protocolo, a resposta é enviada ao *CompletionCoordinatorPortType*, responsável por enviar a resposta da transação ao *CompletionInitiatorPortType* da loja.

O código A.13 apresenta a chamada do método start_2pc do *CoordinatorPortType* pelo *CompletionCoordinatorPortType*. A mensagem inicializa o *two-phase commit*, na linha 5.

O código A.14 apresenta o funcionamento do método *start_2pc* no *CoordinatorPortType* do coordenador, conforme a figura 4.12. Primeiramente, o coordenador cria um vetor para guardar os resultados da votação de cada participante, na linha 3. Em seguida, nas linhas 4, 5, 6 e 7, para cada participante registrado no protocolo *two-phase commit* neste contexto de coordenação (fornecedor e loja), o coordenador executa o método *prepare* do participante, ilustrado no código A.15, escreve um log local com o resultado e o adiciona ao vetor de resultados criado.

Os códigos A.16 e A.17 mostram o funcionamento do método *prepare*, no fornecedor e no banco, respectivamente. Primeiramente, nas linhas 4, 5 e 6 do código A.16, o fornecedor busca o contexto de transação e o produto, incluso neste contexto na troca de mensagens com a loja, nos códigos A.5 e A.6, referente à transação. Na linha 7 do código A.16, o fornecedor inicia uma transação sobre o produto, ou seja, apenas uma requisição executa aquele trecho de código por vez. Então, é verificado se aquele produto está disponível, através do atributo *transaction_amount*. Se a quantidade disponível para transação for maior que zero, esta quantidade é diminuída em 1, salva no banco de dados (linhas 8, 9 e 10). Um log com o resultado da transação (neste caso, *prepared*) é escrito e o resultado é enviado ao coordenador (linhas 11 e 12). Caso contrário, é escrito um log com o resultado

(*abort*), o contexto de transação local é excluído, já que é possível concluir que a transação não será efetivada, e o resultado é enviado ao coordenador, nas linhas 14, 15 e 16. Já no código A.17, o banco procura pelo cartão presente em seu contexto de coordenação e verifica se a senha e código se segurança são válidos (linhas 4, 5, 6 e 7). Se são válidos, o banco inicia uma transação sobre a conta relacionada ao cartão, da mesma forma que o fornecedor faz sobre o produto. Se o atributo *transaction_balance* for maior ou igual ao valor do produto a ser comprado, este é descontado da quantia disponível para transação e o valor é salvo no banco de dados (linhas 10, 11 e 12). O banco então escreve *prepared* em seu log local e envia a resposta ao coordenador (linhas 13 e 14). Caso as senhas sejam inválidas ou a quantia disponível para transação seja menor que o valor do produto, é escrito *abort* no log local, o contexto de coordenação é apagado e é enviada a resposta ao coordenador(linhas 16, 17, 18, 19, 23, 24, 25 e 26).

Após receber o resultado dos participantes e adicioná-los a um vetor, o coordenador faz a apuração dos votos, verificando se alguém votou não, ou seja, respondeu *abort* (linha 9 do código A.14). Se alguém votou sim, nas linhas 9, 10, 11, 12 e 13, o coordenador escreve *rollback* em seu log local e avisa a cada participante que votou sim, ou seja, respondeu *prepared* na fase anterior, que ele deve desfazer as alterações feitas anteriormente (*rollback*). Caso contrário, o coordenador escreve *commit* em seu log local e avisa aos participantes que eles podem efetivar a transação (*commit*), nas linhas 19 e 20. Enfim, o método retorna o resultado da transação (*aborted*, caso alguém vote não, e *committed*, caso todos votem sim).

O código A.18 apresenta o funcionamento do método *commit* do fornecedor. Primeiramente, nas linhas 5, 6 e 7. o fornecedor procura pelo contexto de coordenação e o produto referente à transação. Então, uma transação é feita sobre o produto, na linha 8, e é diminuída a quantidade do produto (e não a quantidade disponível para transação), na linha 9. Na linha 10, o produto é salvo no banco de dados e, na linha 11, um log com o resultado da operação (*committed*) é escrito. Finalmente, o contexto de coordenação é apagado (linhas 13 e 14) e o resultado é enviado ao coordenador (linha 15). Já o código A.19 mostra o método *rollback* do fornecedor. A operação faz exatamente o oposto do método

prepare do fornecedor, desfazendo as mudanças feitas no atributo *transaction_ amount* do produto, na linha 9. A alteração é salva no banco de dados e o log é preenchido com o resultado *aborted* (linhas 10 e 11). Então, o contexto de coordenação é apagado e a resposta *aborted* é enviada ao coordenador (linhas 13, 14 e 15).

O funcionamento do método *commit* do banco pode ser visto no código A.20. Na linha 8, é iniciada uma transação sobre a conta relacionada ao cartão do cliente. Em seguida, o saldo real da conta é descontado, as alterações são salvas no banco de dados e um log é escrito com o resultado da operação (*committed*), nas linhas 9, 10 e 11. O contexto de coordenação é apagado e a resposta é enviada ao coordenador, nas linhas 14 e 15. Finalmente, o código A.21 ilustra o funcionamento do método *rollback* do banco. Também efetuando uma transação sobre a conta relacionada o cartão (linha 8), o banco desfaz as alterações feitas no método *prepare*, adicionando o valor do produto à variável *transaction_ balance* e salvando em seu banco de dados (linhas 9 e 10). Na sequência, um log é escrito com o resultado da operação (*aborted*), na linha 11, o contexto de coordenação é apagado (linhas 13 e 14) e o resultado é enviado ao coordenador.

CAPÍTULO 5

CONCLUSÃO

Este trabalho apresentou a importância dos protocolos de coordenação na comunicação entre sistemas distribuídos via web services. Primeiramente no capitulo 2, foram apresentados os conceitos utilizados para a realização do trabalho, dentre eles, sistemas distribuídos (seção 2.1), sistemas web (seção 2.2), e tecnologias (seção 2.3). A seguir, no capítulo 3, foram apresentados os protocolos de coordenação WS-Coordination (seção 3.1) e WS-Transaction (seção 3.2). Finalmente, no capítulo 4, após apresentar as bibliotecas utilizadas (seção 4.1, foram apresentados exemplos que não utilizam os protocolos (seção 4.2, mostrando suas consequências. Em seguida, uma implementação utilizando os protocolos citados foi proposta (seção 4.3), resolvendo os problemas mostrados anteriormente.

5.1 Trabalhos Futuros

Como citado no capítulo 4, as bibliotecas WashOut[29] e Savon[15] não possuem suporte aos protocolos WS-Coordination e WS-Transaction. Por conta disso, a extensão dessas bibliotecas para suportar esses protocolos ou a implementação de uma nova biblioteca pode ser um trabalho futuro, incluindo algoritmos de recuperação, no caso de queda, que leiam os logs gerados durante uma transação para manter o sistema em um estado consistente.

APÊNDICE A

CÓDIGOS

```
1  class ActivationRequestorPortTypeController < ApplicationController
2      ...
3      def self.create_coordination_context_response
4          client = Savon::Client.new(wsdl:
               ActivationCoordinatorPortTypeWSDL)
5          result = cl.call(:create_coordination_context)
6          r = result.to_hash[:create_coordination_context_response][:
               coordination_context]
7          c = CoordinationContext.new(r[:coordination_type],r[:id], r
               [:registration_service])
8          c
9      end
10     ...
11 end
```

Código A.1: Ativação no participante.

```
1  class ActivationCoordinatorPortTypeController <
       ApplicationController
2      soap_service namespace: "coordinator:
           ActivationCoordinatorPortType"
3      ...
4      def create_coordination_context
5          c = CoordinationContext.new("ws-t")
6          write_log(c, 'Coordination context: ' + c.inspect)
7          $contexts[c.id] = c
8          render :soap => {
9              :coordination_context => {
10                 :id => c.id,
11                 :coordination_type =>
12                     c.coordination_type,
13                 :registration_service =>
                       RegistrationCoordinatorPortTypeWSDL
14             }
15         }
16     end
17     ...
18 end
```

Código A.2: Ativação no coordenador.

```
1  class RegistrationRequestorPortTypeController <
       ApplicationController %1
2      def self.register_response(coordination_context) %2
3          client = Savon::Client.new(wsdl: coordination_context.
               registration_service) %3
4          result = client.call(:register, message: {'
               coordination_context' => { %4
5                  'id' => coordination_context.id,
6                  'coordination_type' => coordination_context.
                       coordination_type,
7                  'registration_service' => coordination_context.
                       registration_service
8              }, 'coordination_protocol' => 'completion',
9              'client_service' => CompletionInitiatorPortTypeWSDL
10         })
11         coordinator_service = result.to_hash[:register_response][:
               coordination_context_response][:coordinator_service]
12         coordination_context.set_completion_service(
               coordinator_service)
13         coordination_context
14     end
15 end
```

Código A.3: Funcionamento do *RegistrationRequestorPortType*.

```
1  class RegistrationCoordinatorPortTypeController <
       ApplicationController
2      soap_service namespace: 'coordinator:
           RegistrationCoordinatorPortType'
3      def register
4          c = $contexts[params[:coordination_context][:id]]
5          if params[:coordination_protocol] == 'completion'
6              c.completion_participants.push params[:client_service]
7              write_log(c, 'Completion participant: ' + params[:
                   client_service])
8              render :soap => {
9                  :coordination_context_response => {
10                     :id => c.id,
11                     :coordination_type => c.coordination_type,
12                     :coordinator_service =>
                           CompletionCoordinatorPortType
13                 }
14             }
15         elsif params[:coordination_protocol] == '2pc'
16             c.tpc_participants.push params[:client_service]
17             write_log(c, '2PC participant: ' + params[:
                   client_service])
18             render :soap => {
19                 :coordination_context_response => {
20                     :id => c.id,
21                     :coordination_type => c.coordination_type,
22                     :coordinator_service => CoordinatorPortType
23                 }
24             }
25         else
26             render :soap => nil
27         end
28     end
29 end
```

Código A.4: Funcionamento do *RegistrationCoordinatorPortType*.

```
1  class MessagePortTypeController < ApplicationController
2      def self.send_product_response(provider_id, product_id,
           coordination_context)
3          p = Provider.find(provider_id)
4          client = Savon::Client.new(wsdl: p.wsdl_location)
5          result = client.call(:send_product,
6              message: {'coordination_context' =>
7              {
8                  'id' => coordination_context.id,
9                  'coordination_type' => coordination_context.
                       coordination_type,
10                 'coordinator_registration_service' =>
                       coordination_context.registration_service,
11                 'product_id' => product_id
12             }
13         })
14     end
15     ...
16 end
```

Código A.5: Funcionamento do método *send_product_response* *MessagePortType* da loja.

```
1  class MessagePortTypeController < ApplicationController
2      soap_service namespace: 'provider:MessagePortType'
3      ...
4      def send_product
5          id = params[:coordination_context][:id]
6          coordination_type = params[:coordination_context][:
               coordination_type]
7          registration_service = params[:coordination_context][:
               coordinator_registration_service]
8          product_id = params[:coordination_context][:product_id]
9          c = CoordinationContext.new(coordination_type, id,
               registration_service, product_id)
10         write_log(c,'Coordination context: ' + params.to_s)
11         c = RegistrationRequestorPortTypeController::
               register_response(c)
12     end
13     ...
14 end
```

Código A.6: Funcionamento do *MessagePortType* do fornecedor.

```
1  class RegistrationRequestorPortTypeController <
       ApplicationController
2      def self.register_response(coordination_context)
3          client = Savon::Client.new(wsdl: coordination_context.
               registration_service)
4          result = client.call(:register,
5              message: {'coordination_context' =>
6              {
7                  'id' => coordination_context.id,
8                  'coordination_type' => coordination_context.
                       coordination_type,
9                  'registration_service' => coordination_context.
                       registration_service
10             },
11             'coordination_protocol' => '2pc',
12             'client_service' => ParticipantPortTypeWSDL
13         })
14         coordinator_service = result.to_hash[:register_response][:
               coordination_context_response][:coordinator_service]
15         coordination_context.set_tpc_service(coordinator_service)
16         $contexts[coordination_context.id] = coordination_context
17         coordination_context
18     end
19
20 end
```

Código A.7: Funcionamento do *RegistrationRequestorPortType* do fornecedor.

```
class MessagePortTypeController < ApplicationController
    ...
    def self.send_card_response(bank_id, card_number,
                                card_password, card_security_code,
                                value, coordination_context)
        b = Bank.find(bank_id)
        client = Savon::Client.new(wsdl: b.wsdl_location)
        result = client.call(:send_card,
            message: {'coordination_context' =>
            {
                'id' => coordination_context.id,
                'coordination_type' => coordination_context.
                    coordination_type,
                'coordinator_registration_service' =>
                    coordination_context.registration_service,
                'card_number' => card_number,
                'card_password' => card_password,
                'card_security_code' => card_security_code,
                'value' => value
            },
        })
    end
end
```

Código A.8: Funcionamento do método *send_card_response* do *MessagePortType* da loja.

```
1  class MessagePortTypeController < ApplicationController
2      soap_service namespace: 'bank:MessagePortType'
3      ...
4      def send_card
5          id = params[:coordination_context][:id]
6          coordination_type = params[:coordination_context][:
               coordination_type]
7          registration_service = params[:coordination_context][:
               coordinator_registration_service]
8          card_number = params[:coordination_context][:card_number]
9          card_password = params[:coordination_context][:card_password
               ]
10         card_security_code = params[:coordination_context][:
               card_security_code]
11         value = params[:coordination_context][:value]
12         c = CoordinationContext.new(coordination_type, id,
               registration_service, card_number, card_password,
               card_security_code, value)
13         write_log(c,'Coordination context: ' + params.to_s)
14         c = RegistrationRequestorPortTypeController::
               register_response(c)
15     end
16     ...
17 end
```

Código A.9: Funcionamento do *MessagePortType* do banco.

```
1  class RegistrationRequestorPortTypeController <
       ApplicationController
2
3      def self.register_response(coordination_context)
4          client = Savon::Client.new(wsdl: coordination_context.
               registration_service)
5          result = client.call(:register,
6              message: {'coordination_context' =>
7              {
8                  'id' => coordination_context.id,
9                  'coordination_type' => coordination_context.
                       coordination_type,
10                 'registration_service' => coordination_context.
                       registration_service
11             },
12             'coordination_protocol' => '2pc',
13             'client_service' => ParticipantPortTypeWSDL
14         })
15         coordinator_service = result.to_hash[:register_response][:
               coordination_context_response][:coordinator_service]
16         coordination_context.set_tpc_service(coordinator_service)
17         $contexts[coordination_context.id] = coordination_context
18         coordination_context
19     end
20 end
```

Código A.10: Funcionamento do *RegistrationRequestorPortType* do banco.

```
1  class CompletionInitiatorPortTypeController < ApplicationController
2      def self.commit_response(coordination_context)
3          client = Savon::Client.new(wsdl: coordination_context.
               completion_service)
4          result = client.call(:commit,
5              message: {'coordination_context' =>
6              {
7                  'id' => coordination_context.id,
8                  'coordination_type' => coordination_context.
                       coordination_type,
9              },
10         })
11         result.to_hash[:commit_response][:result]
12     end
13 end
```

Código A.11: Funcionamento do método *commit_response* do *CompletionInitiatorPortType* da loja.

```
1  class CompletionCoordinatorPortTypeController <
       ApplicationController
2      soap_service namespace: 'coordinator:
           CompletionCoordinatorPortType'
3      ...
4      def commit
5          c = $contexts[params[:coordination_context][:id]]
6          result = start_2pc_response(c)
7          c = nil
8          $contexts.delete(params[:coordination_context][:id])
9          render :soap => { :result => result }
10     end
11     ...
12 end
```

Código A.12: Funcionamento do método *commit* do *CompletionCoordinatorPortType* do coordenador.

```
1  class CompletionCoordinatorPortTypeController <
       ApplicationController
2      soap_service namespace: 'coordinator:
           CompletionCoordinatorPortType'
3      ...
4      def start_2pc_response(c)
5          CoordinatorPortTypeController::start_2pc(c)
6      end
7      ...
8  end
```

Código A.13: Funcionamento do método *start_2pc_response* do *CompletionCoordinatorPortType* do coordenador.

```
class CoordinatorPortTypeController < ApplicationController
    def self.two_phase_commit(c)
        results = []
        c.tpc_participants.each do |wsdl|
            r = CoordinatorPortTypeController::prepare_response(wsdl
                ,c)
            write_log(c,wsdl + " vote: " + r)
            results.push(r)
        end
        if results.include? "abort"
            write_log(c, "rollback")
            for i in 0..c.tpc_participants.size-1 do
                if results[i] == 'prepared'
                    r = CoordinatorPortTypeController::
                        rollback_response(c.tpc_participants[i],c)
                end
            end
            "aborted"
        else
            write_log(c, "commit")
            c.tpc_participants.each do |wsdl|
                r = CoordinatorPortTypeController::commit_response(
                    wsdl,c)
            end
            "committed"
        end
    end
end
```

Código A.14: Funcionamento do método *start_2pc* do *CoordinatorPortType* do coordenador.

```
1  class CoordinatorPortTypeController < ApplicationController
2      def self.prepare_response(participant_wsdl,coordination_context)
3          client = Savon::Client.new(wsdl: participant_wsdl)
4          result = client.call(:prepare, message: {
5              'coordination_context' =>
6              {
7                  'id' => coordination_context.id,
8                  'coordination_type' => coordination_context.
                       coordination_type,
9              },
10         })
11         result.to_hash[:prepare_response][:result]
12     end
13 end
```

Código A.15: Funcionamento do método *prepare_response* do *CoordinatorPortType* do coordenador.

```
class ParticipantPortTypeController < ApplicationController
    soap_service namespace: 'provider:ParticipantPortType'
    def prepare
        id = params[:coordination_context][:id]
        c = $contexts[id]
        p = Product.find(c.product_id)
        p.transaction do
            if p.transaction_amount > 0
                p.transaction_amount -= 1
                p.save!
                write_log(c, "2PC Vote: prepared")
                render :soap => { :result => 'prepared'}
            else
                write_log(c, "2PC Vote: abort")
                c = nil
                $contexts.delete(id)
                render :soap => { :result => 'abort'}
            end
        end
    end
end
```

Código A.16: Funcionamento do método *prepare* do *ParticipantPortType* do fornecedor.

```
class ParticipantPortTypeController < ApplicationController
    soap_service namespace: 'bank:ParticipantPortType'
    def prepare
        id = params[:coordination_context][:id]
        c = $contexts[id]
        card = Card.where(:number => c.card_number).first
        if card.password == c.card_password &&
           card.security_code == c.card_security_code
            card.account.transaction do
              if card.account.transaction_balance >= c.value.to_f
                card.account.transaction_balance -= c.value.to_f
                card.account.save!
                write_log(c,'2PC Vote: prepared')
                render :soap => { :result => 'prepared' }
              else
                write_log(c,'2PC Vote: abort')
                c = nil
                $contexts.delete(id)
                render :soap => { :result => 'abort' }
              end
            end
        else
          write_log(c,'2PC Vote: abort')
          c = nil
          $contexts.delete(id)
          render :soap => { :result => 'abort' }
        end
    end
end
```

Código A.17: Funcionamento do método *prepare* do *ParticipantPortType* do banco.

```
class ParticipantPortTypeController < ApplicationController
    soap_service namespace: 'provider:ParticipantPortType'
    ...
    def commit
        id = params[:commit_coordination_context][:id]
        c = $contexts[id]
        p = Product.find(c.product_id)
        p.transaction do
            p.amount -= 1
            p.save!
            write_log(c, "2PC Result: committed")
        end
        c = nil
        $contexts.delete(id)
        render :soap => { :result => 'committed'}
    end
end
```

Código A.18: Funcionamento do método *commit* do *ParticipantPortType* do fornecedor.

```
class ParticipantPortTypeController < ApplicationController
    soap_service namespace: 'provider:ParticipantPortType'
    ...
    def rollback
        id = params[:rollback_coordination_context][:id]
        c = $contexts[id]
        p = Product.find(c.product_id)
        p.transaction do
            p.transaction_amount += 1
            p.save!
            write_log(c, "2PC Result: aborted")
        end
        c = nil
        $contexts.delete(id)
        render :soap => { :result => 'aborted'}
    end
end
```

Código A.19: Funcionamento do método *rollback* do *ParticipantPortType* do fornecedor.

```
class ParticipantPortTypeController < ApplicationController
    soap_service namespace: 'bank:ParticipantPortType'
    ...
    def commit
        id = params[:commit_coordination_context][:id]
        c = $contexts[id]
        card = Card.where(:number => c.card_number).first
        card.account.transaction do
            card.account.balance -= c.value.to_f
            card.account.save!
            write_log(c,'2PC Result: committed')
        end
        c = nil
        $contexts.delete(id)
        render :soap => { :result => 'committed' }
    end
end
```

Código A.20: Funcionamento do método *commit* do *ParticipantPortType* do banco.

```
class ParticipantPortTypeController < ApplicationController
    soap_service namespace: 'bank:ParticipantPortType'
    ...
    def rollback
        id = params[:rollback_coordination_context][:id]
        c = $contexts[id]
        card = Card.where(:number => c.card_number).first
        card.account.transaction do
            card.account.transaction_balance += c.value.to_f
            card.account.save!
            write_log(c,'2PC Result: aborted')
        end
        c = nil
        $contexts.delete(id)
        render :soap => { :result => 'aborted' }
    end
end
```

Código A.21: Funcionamento do método *rollback* do *ParticipantPortType* do banco.

REFERÊNCIAS

[1] Acid. https://pt.wikipedia.org/wiki/ACID. Acessado no dia 31/10/2015.

[2] Artigo .net magazine 54 - introdução a web services. http://www.devmedia.com.br/artigo-net-magazine-54-introducao-a-web-services/10726. Acessado no dia 12/10/2015.

[3] Banco de dados, https://pt.wikipedia.org/wiki/Banco_de_dados. Acessado no dia 23/12/2015.

[4] Chamada de procedimento remoto. https://pt.wikipedia.org/wiki/Chamada_de_procedimento_remoto. Acessado no dia 15/10/2015.

[5] Exclusão mútua, https://pt.wikipedia.org/wiki/Exclus~ao_mútua. Acessado no dia 23/12/2015.

[6] Facebook caiu? entenda o quanto cada minuto fora do ar impacta a rede. http://www.techtudo.com.br/artigos/noticia/2014/08/facebook-caiu-entenda-o-quanto-cada-minuto-fora-do-ar-impacta-rede.html. Acessado no dia 31/10/2015.

[7] História da informática e da internet: 1990-1999. http://www.ufpa.br/dicas/net1/int-h199.htm. Acessado no dia 15/10/2015.

[8] Javascript. https://pt.wikipedia.org/wiki/JavaScript. Acessado no dia 09/11/2015.

[9] O que é um framework? http://www.dsc.ufcg.edu.br/~jacques/cursos/map/html/frame/oque.htm. Acessado no dia 09/11/2015.

[10] Ruby 2.2.3 released. https://www.ruby-lang.org/en/news/2015/08/18/ruby-2-2-3-released/. Acessado no dia 10/11/2015.

[11] Ruby on rails. http://rubyonrails.org/. Acessado no dia 10/11/2015.

[12] Ruby on rails. https://en.wikipedia.org/wiki/Ruby_on_Rails. Acessado no dia 10/11/2015.

[13] Ruby. https://pt.wikipedia.org/wiki/Ruby_(linguagem_de_programaç~ao). Acessado no dia 10/11/2015.

[14] Rubygems. https://en.wikipedia.org/wiki/RubyGems. Acessado no dia 10/11/2015.

[15] Savon. http://savonrb.com/. Acessado no dia 16/11/2015.

[16] Semáforo. https://pt.wikipedia.org/wiki/Semáforo_(Computaç~ao). Acessado no dia 12/11/2015.

[17] Twitter rest api. https://dev.twitter.com/rest/public. Acessado no dia 09/12/2015.

[18] CSS. https://developer.mozilla.org/pt-BR/docs/Web/CSS. Acessado no dia 09/11/2015.

[19] CSS. https://en.wikipedia.org/wiki/Cascading_Style_Sheets. Acessado no dia 09/11/2015.

[20] HP SOAP. http://h71000.www7.hp.com/openvms/journal/v4/examining_web_services.html. Acessado no dia 12/11/2015.

[21] HTML. https://developer.mozilla.org/pt-BR/docs/Web/HTML. Acessado no dia 09/11/2015.

[22] HTML. https://pt.wikipedia.org/wiki/HTML. Acessado no dia 09/11/2015.

[23] HTML. http://www.w3schools.com/html/html_intro.asp. Acessado no dia 09/11/2015.

[24] W3C. https://pt.wikipedia.org/wiki/W3C. Acessado no dia 19/10/2015.

[25] XML SOAP. http://www.w3schools.com/xml/xml_soap.asp. Acessado no dia 10/11/2015.

[26] XML, http://courses.ischool.berkeley.edu/i290-14/s05/lecture-2/slide2.html. Acessado no dia 09/11/2015.

[27] XML, https://pt.wikipedia.org/wiki/XML. Acessado no dia 09/11/2015.

[28] XML WSDL. http://www.w3schools.com/xml/xml_wsdl.asp. Acessado no dia 11/11/2015.

[29] Washout. https://github.com/inossidabile/wash_out. Acessado no dia 16/11/2015.

[30] Kuno H. Machiraju V. Alonso G., Casati F. *Web Services: concepts, architectures and applications*. 2010.

[31] Ivan Bittencourt de Araújo e Silva Neto. Um serviço de transações atômicas para web services. 2007. Acessado no dia 10/12/2015.

[32] Julia Gadelha. A evolução dos computadores. http://www2.ic.uff.br/~aconci/evolucao.html. Acessado no dia 13/10/2015.

[33] Carmem Hara. Aula 10: Conceitos de transações (cap. 19). http://www.inf.ufpr.br/carmem/ci218/aulas/aula10-trans. Acessado no dia 28/10/2015.

[34] Weihl W. Fekete A. Lynch N., Merritt M. *Atomic Transactions*. 1994.

[35] Cesar Augusto Tacla. Introdução aos sistemas distribuídos. http://dainf.ct.utfpr.edu.br/~tacla/JAVAProgParSD/0010-IntroducaoSD.pdf. Acessado no dia 28/10/2015.

[36] Andrew S. Tanenbaum. *Sistemas Distribuídos: princípios e paradigmas*, volume 2. 2008.