# Um serviço de transações atômicas para Web services

**Ivan Bittencourt de Araújo e Silva Neto**

DISSERTAÇÃO APRESENTADA
AO
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA
DA
UNIVERSIDADE DE SÃO PAULO
PARA
OBTENÇÃO DO TÍTULO
DE
MESTRE EM CIÊNCIAS

Área de Concentração: **Ciência da Computação**
Orientador: **Prof. Dr. Francisco Carlos da Rocha Reverbel**

*Durante a elaboração deste trabalho o autor recebeu auxílio financeiro da CAPES.*

São Paulo, novembro de 2007

# Um serviço de transações atômicas para Web services

Este exemplar corresponde à redação final da dissertação devidamente corrigida e defendida por Ivan Bittencourt de Araújo e Silva Neto e aprovada pela Comissão Julgadora.

Banca Examinadora :

Prof. Dr. Francisco Carlos da Rocha Reverbel (orientador) – IME-USP

Prof. Dr. Fabio Kon – IME-USP

Prof. Dr. Wilson Vicente Ruggiero – Poli-USP

## Agradecimentos

Primeiramente, agradeço à minha família, por todo o apoio, dedicação e incentivo. Da mesma forma, sou muito grato à Fabiana, pelo seu incondicional apoio. Sua paciência e carinho foram fundamentais para mim durante a realização deste trabalho.

Expresso aqui também a minha gratidão ao meu orientador, o professor Francisco Carlos da Rocha Reverbel. Sou grato pela confiança que ele depositou em mim, tendo aceitado ser meu orientador, pela sua dedicação a este trabalho de mestrado e pelo grande aprendizado que ele me proporcionou.

Agradeço também aos professores Fabio Kon e Wilson Vicente Ruggiero por terem participado na banca da minha defesa de mestrado. Suas opiniões e sugestões certamente contribuíram para a melhoria deste trabalho.

Finalmente, agradeço à CAPES, pelo apoio financeiro, e a todos os amigos e colegas que contribuíram de forma direta ou indireta para a realização deste trabalho.

## Resumo

Sistemas computacionais são constituídos por componentes de *hardware* e *software* que podem eventualmente falhar. Por esse motivo, o mecanismo de transação sempre foi imprescindível para a construção de sistemas robustos. O suporte transacional para a tecnologia *Web services* foi definido em agosto de 2005, num conjunto de três especificações denominadas *WS-Coordination*, *WS-AtomicTransaction* e *WS-BusinessActivity*. Juntas, essas especificações definem um alicerce sobre o qual aplicações robustas baseadas em *Web services* podem ser construídas.

Nesta dissertação realizamos um estudo sobre transações atômicas em ambientes *Web services*. Em particular, estendemos o gerenciador de transações presente no servidor de aplicações *JBoss*, de modo que ele passasse a comportar transações distribuídas envolvendo *Web services*. Além disso, avaliamos o desempenho desse gerenciador de transações quando ele emprega cada um dos seguintes mecanismos de chamada remota: *Web services/SOAP*, *CORBA/IIOP* e *JBoss Remoting*. Finalmente, realizamos experimentos de escalabilidade e interoperabilidade.

## Abstract

Computing systems consist of a multitude of hardware and software components that may fail. For this reason, the transaction mechanism has always been essential for the development of robust systems. Transactional support for the Web services technology was defined in August 2005, in a set of three specifications, namely WS-Coordination, WS-AtomicTransaction, and WS-BusinessActivity. Together, such specifications enable the development of robust Web services applications.

In this dissertation we studied atomic transactions in the Web services realm. Particularly, we added Web services atomic transaction support to the existing JBoss application server transaction manager. Furthermore, we evaluated the performance of this transaction manager when it employs each of the following remote method invocation mechanisms: Web services/SOAP, CORBA/IIOP and JBoss Remoting. Finally, we performed scalability and interoperability experiments.

# Sumário

# 1 Introdução

Inicialmente apresentados em meados do ano 2000, os *Web services* [3] têm se tornado cada vez mais populares no contexto da computação distribuída. Eles surgiram como uma solução para a crescente necessidade de integração entre aplicações, permitindo que tais aplicações se comunicassem entre si de modo independente de plataforma e linguagem de programação. O sucesso da tecnologia pode ser justificado pela solução que ela proporciona a um dos grandes desafios enfrentados por muitas empresas: integrar seus diversos sistemas já existentes.

Uma importante característica dos *Web services* é que eles são baseados em padrões abertos e amplamente adotados pela indústria, tais como a *eXtensible Markup Language* (*XML*) [15] e o *HyperText Transfer Protocol* (*HTTP*) [24]. Ademais, há atualmente um enorme esforço por parte da indústria para a disseminação dos *Web services*. Entre as empresas que oferecem produtos e soluções para a tecnologia destacam-se a *Microsoft*, a *IBM* e a *Sun Microsystems*. Apesar de ser uma tecnologia relativamente nova, *Web services* já atingiram um alto grau de maturidade e constituem hoje uma importante parte das plataformas *Java Enterprise Edition* (*Java EE*) [74] e *Microsoft .NET* [20].

*Web services* são utilizados principalmente como pontos de acesso para sistemas de informações já existentes (Figura 1.1). O principal papel da tecnologia é expor, através da *Web* [10], a funcionalidade desses sistemas. Assim sendo, os *Web services* podem ser vistos como uma espécie de invólucro (*wrapper*) que esconde a heterogeneidade dos sistemas e possibilita a interoperabilidade entre eles. Tal fato torna a tecnologia *Web services* interessante não apenas para a integração de sistemas dentro de uma única empresa, mas principalmente para a integração de sistemas presentes em várias empresas (integração negócio-a-negócio, ou *business-to-business*).

Em ambientes *Web services*, as aplicações são construídas a partir da vinculação de componentes também denominados *Web services*. O *World Wide Web Consortium* (*W3C*) define *Web service* do seguinte modo [12]:

> "Um *Web service* é um sistema de software projetado para comportar interações máquina-a-máquina através de uma rede. Ele possui uma interface descrita num formato propício para processamento por computadores (*WSDL* [21, 22]). Outros

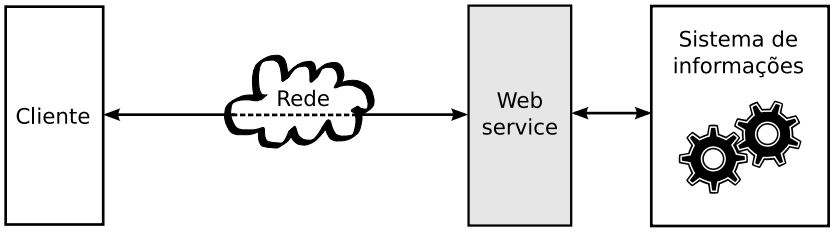



Figura 1.1: *Web services* são utilizados predominantemente como pontos de acesso para sistemas de informações já existentes.

> sistemas interagem com o *Web service* do modo prescrito pela sua interface, utilizando mensagens *SOAP* [30, 31] tipicamente transportadas utilizando o *HTTP* e seriação *XML* em conjunto com outros padrões *Web* relacionados."

Em outras palavras, um *Web service* é um sistema de software com uma interface bem definida que pode ser acessado através de uma rede por meio mensagens *XML* padronizadas.

Em aplicações distribuídas baseadas em *Web services*, é muito comum que um conjunto de interações entre *Web services* tenha que ser executado de modo atômico. Caso isso não seja possível, a aplicação pode ter a sua consistência comprometida. O que se deseja em muitos casos, portanto, é que as interações entre *Web services* possam ser executadas dentro do escopo de uma transação, de modo que tais interações se comportem como um bloco atômico. Para lidar com situações como essas, um grupo de empresas publicou, em agosto de 2005, um conjunto de três especificações definindo um serviço transacional para a tecnologia *Web services*. Essas três especificações são:

- *WS-Coordination* [19]: essa especificação atua como uma espécie de fundação sobre a qual outras especificações, como por exemplo *WS-AtomicTransaction* e *WS-BusinessActivity*, são definidas. Ela descreve um coordenador genérico que possui duas funções principais: *(i)* permitir a criação de contextos de coordenação e *(ii)* prover uma infra-estrutura para o registro dos participantes envolvidos numa atividade coordenada. O coordenador definido em *WS-Coordination* é na realidade um arcabouço, e ele só é útil quando a sua funcionalidade é estendida.

- *WS-AtomicTransaction* [17]: essa especificação estende o coordenador genérico definido em *WS-Coordination*, habilitando-o a lidar com transações atômicas.

- *WS-BusinessActivity* [18]: essa especificação também estende o coordenador genérico definido em *WS-Coordination*, habilitando-o a lidar com transações distribuídas de longa duração.

Juntas, essas três especificações definem um alicerce sobre o qual aplicações robustas baseadas em *Web services* podem ser construídas.

Nesta dissertação, realizamos um estudo sobre transações atômicas em ambientes *Web services*. Em particular, estendemos o gerenciador de transações presente no servidor de aplicações *JBoss* [25], de modo que ele passasse a comportar transações distribuídas envolvendo *Web services*. Essa extensão foi efetuada de acordo com as diretrizes descritas nas especificações *WS-Coordination* e *WS-AtomicTransaction*. Efetuamos também uma avaliação comparativa do desempenho de nossa versão estendida do gerenciador de transações do *JBoss* para os seguintes mecanismos de chamada remota: *Web services/SOAP*, *CORBA/IIOP* e *JBoss Remoting* [37]. Os dois últimos mecanismos já eram comportados pelo gerenciador de transações; o suporte ao primeiro é fruto de nosso trabalho. Finalmente, realizamos experimentos de escalabilidade e interoperabilidade.

Nossa versão aprimorada do gerenciador de transações do *JBoss* é capaz de coordenar transações distribuídas envolvendo simultaneamente quatro tipos de recursos transacionais. São eles: recursos compatíveis com o padrão *XA* [75], recursos remotos acessíveis via *JBoss Remoting*, recursos remotos acessíveis via *CORBA* e recursos remotos acessíveis como *Web services*[1]. Até onde temos conhecimento, nossa versão estendida do gerenciador de transações do *JBoss* é o único gerenciador capaz de coordenar transações envolvendo simultaneamente esses quatro tipos de recursos (Figura 1.2).

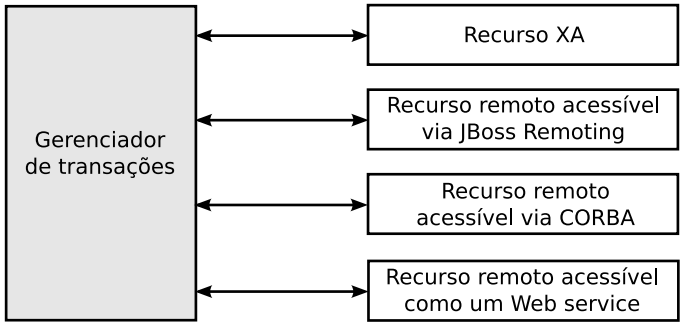



Figura 1.2: Os quatro tipos de recursos transacionais comportados pela nossa versão aprimorada do gerenciador de transações do *JBoss*.

[1]Maiores detalhes serão apresentados no Capítulo 4.

## 1.1 Organização do texto

Esta dissertação está organizada da seguinte forma: o Capítulo 2 descreve trabalhos relacionados, o Capítulo 3 apresenta uma visão geral de transações distribuídas em ambientes *Web services*, o Capítulo 4 expõe o projeto desenvolvido como parte do nosso estudo, o Capítulo 5 contém uma avaliação de desempenho da nossa versão estendida do gerenciador de transações do *JBoss* e, finalmente, o Capítulo 6 traz nossas conclusões.

# 2 Trabalhos relacionados

Este capítulo apresenta os trabalhos relacionados que foram analisados e que contribuíram para o nosso estudo. O capítulo está dividido em três partes. Na primeira apresentamos o serviço de transações *CORBA*; na segunda e na terceira analisamos dois serviços de transações que oferecem suporte para *Web services*. São eles: o *Arjuna Transaction Service* [56] e o *Apache Kandula* [5].

## 2.1 O serviço de transações CORBA

Apesar da importância do conceito de transação na construção de aplicações robustas, a primeira versão de *CORBA*, datada de 1991, não definia nenhum serviço transacional. Conseqüentemente, a execução de transações distribuídas em ambientes *CORBA* era infactível ou viabilizada apenas através de abordagens *ad hoc*.

Esse cenário começou a mudar em 1994, quando o *Object Management Group* (*OMG*) publicou a especificação do serviço de transações *CORBA* (*CORBA Transaction Service*, ou *OTS*[1]) [52], um serviço transacional capaz de operar em ambientes baseados na arquitetura *CORBA*. O *OTS* foi concebido tendo dois principais objetivos: *(i)* contemplar transações envolvendo objetos distribuídos por múltiplos *ORBs*[2] e *(ii)* ser capaz de interagir com os sistemas de processamento de transações que o antecederam. Este último ponto era importante para favorecer a adoção do *OTS* e para preservar os investimentos feitos pelas empresas em outros sistemas de processamento de transações, como por exemplo os compatíveis com o modelo *X/Open Distributed Transaction Processing* [75].

Apesar dos esforços para estimular a sua adoção, a aceitação inicial do *OTS* foi muito inferior à esperada. A primeira implementação compatível com a especificação, o *ArjunaTS* [45], surgiu apenas em 1997, cerca de três anos após a publicação do *OTS*. Contudo, apesar do longo período inicial sem nenhuma implementação, a partir de 1997 a aceitação do *OTS* começou a

[1] O serviço de transações *CORBA* era originalmente denominado *Object Transaction Service* (*OTS*). Apesar da alteração do nome para *CORBA Transaction Service*, a sigla *OTS* continua sendo utilizada.

[2] O *Object Request Broker* (*ORB*) é o elemento de middleware que possibilita o acesso remoto aos objetos *CORBA*. Ele implementa um mecanismo de chamada remota de método que é independente de linguagem de programação.

crescer. A especificação ganhou diversas implementações e cerca de dois anos depois, em meados de 1999, uma série de produtos compatíveis com o *OTS* estavam disponíveis. Atualmente o *OTS* possui um alto grau de maturidade e constitui um importante serviço da arquitetura *CORBA*. Ele é o componente chave para o desenvolvimento de aplicações *CORBA* robustas e confiáveis.

Existem vários fornecedores de middleware que hoje apresentam implementações compatíveis com a especificação. Como exemplos podemos mencionar o *Orbix OTS* [54] da *IONA*, o *VisiBroker ITS* [77] da *Borland*, o *ArjunaTS* da *Arjuna*, além das implementações do *OTS* integradas aos servidores de aplicações *WebLogic* [79], da *BEA*, e *WebSphere* [80], da *IBM*.

### 2.1.1 Uma visão geral do OTS

A especificação do *OTS* descreve um serviço transacional para ambientes *CORBA*. Resumidamente, esse serviço define: *(i)* um formato para o contexto transacional[3] e *(ii)* um conjunto de interfaces para a comunicação entre as partes envolvidas numa transação distribuída. A definição de um formato para o contexto transacional, feita em *CORBA IDL* [53], é importante para assegurar a interoperabilidade entre diferentes implementações do *OTS*. Como o contexto transacional é propagado juntamente com as requisições *CORBA* que ocorrem dentro do escopo de uma transação, duas implementações distintas do *OTS* devem empregar o mesmo formato de contexto transacional para que a comunicação entre elas seja possível. Já a definição de interfaces padronizadas para a comunicação entre os membros de uma transação distribuída é importante por, pelo menos, dois motivos:

1. Ela possibilita que a comunicação entre os membros de uma transação distribuída seja feita através de operações pré-determinadas e conhecidas de antemão por todos esses membros. Como as interfaces são as mesmas para todas as implementações do *OTS*, um cliente pode chamar as operações dessas interfaces independentemente da implementação do *OTS* subjacente.

2. Como apenas as interfaces são definidas, os implementadores da especificação têm a liberdade de implementar as operações dessas interfaces da maneira que acharem mais conveniente. Não há nenhum tipo de imposição sobre o modo como as operações devem ser implementadas.

A especificação do *OTS* define mais de uma dezena de interfaces para viabilizar a comunicação entre os membros de uma transação distribuída. Entre elas, há quatro que merecem

[3] O contexto transacional contém informações sobre uma transação distribuída. Ele é propagado juntamente com as chamadas remotas efetuadas dentro do escopo da transação.

destaque: `Current`, `Coordinator`, `Resource` e `Terminator`. A interface `Current` funciona como uma fachada [27] e é utilizada por um cliente (uma aplicação escrita em Java ou C++, por exemplo) para acessar as funcionalidades do serviço de transações *CORBA*. Isso significa que, ao invés de interagir com o serviço de transações através de interfaces de baixo nível, o cliente pode utilizar uma interface de alto nível que esconde boa parte da complexidade do serviço (Figura 2.1). A seguir apresentamos algumas das operações presentes na interface `Current`:

- `begin`: cria uma nova transação e a associa à linha de execução (*thread*) chamadora. Requisições *CORBA* efetuadas após a execução dessa operação propagarão o contexto transacional.
- `commit`: efetiva a transação associada à linha de execução chamadora. Após a execução dessa operação, a linha de execução chamadora não estará associada a nenhuma transação.
- `rollback`: aborta a transação associada à linha de execução chamadora. Após a execução dessa operação, a linha de execução chamadora não estará associada a nenhuma transação.

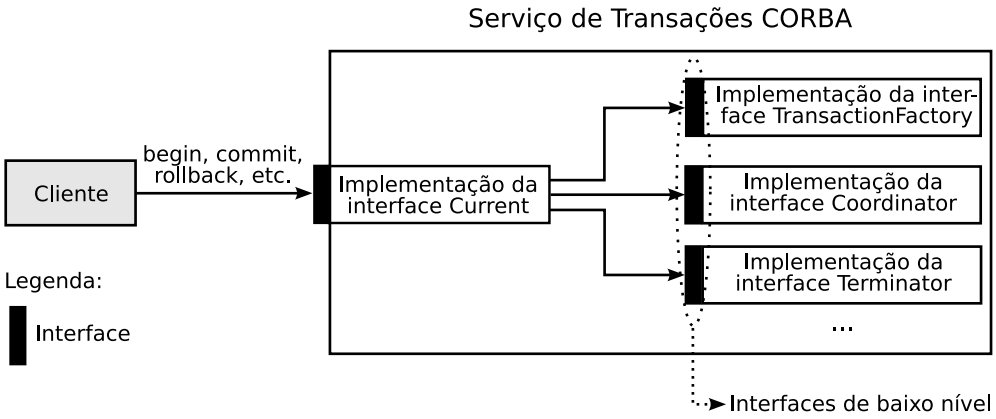



Figura 2.1: O cliente interage com o serviço de transações *CORBA* através da interface `Current`.

A interface `Coordinator` representa o coordenador de uma transação distribuída, provendo operações que são chamadas pelos participantes durante a execução de uma transação. Essa interface define, por exemplo, a operação `register_resource`, que é utilizada para registrar recursos transacionais numa transação distribuída. Já a interface `Resource` representa um

recurso transacional, definindo operações como `prepare`, `commit` e `rollback`. Tais operações são chamadas pelo coordenador da transação durante a execução do protocolo de efetivação em duas fases (*two-phase commit*)[4] [28]. A Figura 2.2 ilustra a comunicação entre as entidades que implementam as interfaces `Coordinator` e `Resource`. Por fim, a interface `Terminator` provê os métodos `commit` e `rollback`, que permitem ao iniciador da transação encerrá-la.

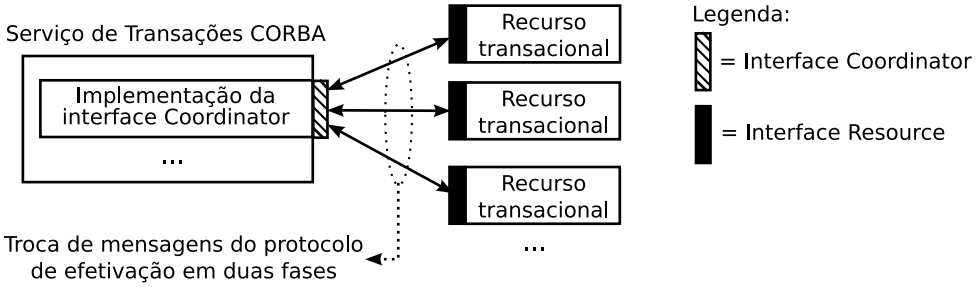



Figura 2.2: As interfaces `Coordinator` e `Resource` permitem a comunicação entre o coordenador e os participantes de uma transação distribuída.

### 2.1.2 OTS e Java Enterprise Edition

O *OTS* desempenha hoje um importante papel não apenas na arquitetura *CORBA*, mas também na plataforma *Java EE*. Isso porque a especificação do *Java Transaction Service* (*JTS*) [63], ao invés de definir um novo gerenciador de transações, optou por implementar o mapeamento para Java do *OTS*. O *JTS* define um gerenciador de transações que: *(i)* disponibiliza a *Java Transaction API* (*JTA*) [65] para interações de alto nível e *(ii)* realiza as interações de baixo nível através de uma implementação do mapeamento para Java do *OTS*. Isso significa que o *JTS* é definido como uma camada sobre o *OTS*, e portanto utiliza as interfaces do *OTS* e o *Internet Inter-ORB Protocol* (*IIOP*) [**?**] para coordenar transações distribuídas.

A fundamentação do *JTS* no *OTS* faz com que implementações dessas especificações possam interoperar entre si. Isso permite que um gerenciador de transações compatível com a especificação do *JTS* possa interagir com objetos transacionais *CORBA* que potencialmente não foram implementados em Java. Em outras palavras, o *JTS* se beneficia de toda interoperabilidade e portabilidade oferecida pela arquitetura *CORBA*, pois na realidade uma implementação do *JTS* é também uma implementação do *OTS*.

[4] O protocolo de efetivação em duas fases será abordado no Capítulo 3.

| CORBA OTS | WS-Coordination/WS-AtomicTransaction |
|---|---|
| TransactionFactory | ActivationCoordinatorPortType |
| Terminator | CompletionCoordinatorPortType |
| Coordinator | CoordinatorPortType |
| RecoveryCoordinator | CoordinatorPortType |
| Resource | ParticipantPortType |
| Synchronization | ParticipantPortType |

Tabela 2.1: Comparação entre as interfaces do *CORBA OTS* e os *port types* de *WS-Coordination* e *WS-AtomicTransaction*. Uma interface e um *port type* presentes na mesma linha apresentam finalidade semelhante.

### 2.1.3 Relação com o nosso trabalho

As especificações *WS-Coordination* e *WS-AtomicTransaction*, que são o foco do nosso trabalho, apresentam muitas semelhanças com a especificação do *OTS*. Por exemplo, assim como o *OTS*, *WS-Coordination* e *WS-AtomicTransaction* definem: *(i)* um formato padrão para o contexto transacional e *(ii)* um conjunto de *port types*[5] para a comunicação entre os membros de uma transação distribuída.

Embora os contextos transacionais definidos pelas especificações sejam diferentes, ambos possuem um mesmo conjunto fundamental de informações. Por exemplo, os dois contextos contêm o identificador da transação e o endereço de um coordenador com o qual é possível registrar recursos transacionais. Logo, pode-se dizer que os contextos definidos pelas especificações são bastante similares.

Além disso, as interfaces e os *port types* definidos pelas especificações também são muito semelhantes e, em muitos casos, suas funções são análogas. Por exemplo, para possibilitar a criação de novas transações o *OTS* define a interface `TransactionFactory`, enquanto *WS-Coordination* define o *port type* `ActivationCoordinatorPortType`[6]. Do mesmo modo, a interface `Resource` do *OTS* representa um recurso transacional, e é equivalente ao *port type* `ParticipantPortType` de *WS-AtomicTransaction*. E as equivalências não acabam por aí: é possível relacionar cada um dos *port types* de *WS-Coordination* e *WS-AtomicTransaction* a alguma interface do *OTS* (Tabela 2.1). Embora estes relacionamentos de equivalência não sejam perfeitos, eles refletem bem as semelhanças entre as especificações.

Há também, evidentemente, diferenças importantes entre o *OTS* e a dupla *WS-Coordination* e *WS-AtomicTransaction*. Primeiramente, no caso do *OTS*, a comunicação entre os membros

[5] Um *port type* consiste num grupo de operações, sendo que um *Web service* pode implementar um ou mais *port types*.

[6] No Capítulo 3 veremos em detalhe quais os *port types* definidos por *WS-Coordination* e *WS-AtomicTransaction*.

de uma transação distribuída ocorre através de chamadas síncronas que utilizam o protocolo *IIOP*. Já no caso de *WS-Coordination* e *WS-AtomicTransaction*, a comunicação ocorre através de chamadas assíncronas que utilizam o protocolo *SOAP*.

Uma segunda diferença importante é que as especificações para *Web services* não contemplam a efetivação em uma só fase, uma otimização bastante comum no âmbito das transações distribuídas [2]. Conforme veremos no Capítulo 3, o protocolo de efetivação em duas fases é o mecanismo mais utilizado para efetivar uma transação distribuída, sendo empregado tanto pelo *OTS* quanto por *WS-AtomicTransaction*. Como o próprio nome sugere, o protocolo possui duas fases, denominadas fase de votação e fase de efetivação. Entretanto, em condições especiais (quando há apenas um recurso transacional envolvido na transação distribuída), a fase de votação pode ser suprimida, possibilitando um processo de efetivação mais rápido. Quando isso ocorre, dizemos que a transação foi efetivada em somente uma fase. A implementação dessa simplificação é exigida pelo *OTS*, mas não por *WS-AtomicTransaction*. Logo, para que uma implementação de *WS-AtomicTransaction* seja interoperável, ela não deve pressupor que outras implementações sejam capazes de executar a efetivação em uma só fase.

Outra dessemelhança relevante entre as especificações diz respeito ao identificador da transação, que é propagado como parte do contexto transacional. No caso do *OTS*, um *Xid* [75] é propagado no contexto transacional. Um *Xid* possui três partes: *(i)* um identificador global, que consiste num vetor de bytes, *(ii)* um identificador de formato, específico de cada implementação do *OTS*, representado por um inteiro, e *(iii)* um identificador do ramo da transação (*branch qualifier*), que também consiste num vetor de bytes. A rigor, o que de fato precisa ser propagado num contexto transacional são os identificadores global e de formato (a propagação do identificador do ramo da transação é dispensável). Como o contexto definido no *OTS* propaga um *Xid*, esses dois identificadores podem ser facilmente conduzidos como parte desse *Xid*.

Já em *WS-AtomicTransaction*, um vetor de bytes, e não um *Xid*, deve conduzir o identificador da transação. Desse modo, apenas o identificador global da transação pode ser propagado de modo interoperável. Embora seja possível agrupar os identificadores global e de formato num único vetor de bytes, a especificação *WS-AtomicTransaction* não descreve como fazer isso. Conseqüentemente, fazê-lo resultaria numa implementação não interoperável. Logo, no caso de *WS-AtomicTransaction*, não há como propagar o identificador de formato de maneira interoperável. Uma conseqüência disso é que quando implementações distintas de *WS-AtomicTransaction* utilizam um mesmo recurso transacional no escopo de uma certa transação, tais acessos ocorrem com o mesmo identificador global, mas com identificadores de formato diferentes. Isso faz com que o recurso transacional não considere os dois acessos

como pertencentes à mesma transação. Conseqüentemente, o recurso transacional fica impedido de executar algumas otimizações durante a efetivação da transação distribuída. Para sanar essa deficiência, a especificação *WS-AtomicTransaction* precisaria definir como propagar o identificador de formato dentro do contexto transacional.

## 2.2 O Arjuna Transaction Service

Os esforços em torno do *Arjuna Transaction Service* (*ArjunaTS*) [45] se iniciaram na segunda metade da década de 80. Naquela época, o *ArjunaTS* era um projeto acadêmico na Universidade de *Newcastle* e seu principal objetivo era avaliar o uso de técnicas de orientação a objetos no desenvolvimento de aplicações tolerantes a falhas.

A primeira versão do *ArjunaTS*, escrita em C++, foi lançada em 1990. Já em 1997, o *ArjunaTS* tornou-se compatível com a especificação do *OTS* e deixou de ser um projeto acadêmico para se transformar num produto comercial da *Arjuna Solutions*, empresa criada por membros do projeto acadêmico. A disseminação de Java como plataforma para o desenvolvimento de aplicações distribuídas fez com que o *ArjunaTS* fosse portado para essa linguagem, tornando-se em 1998 o primeiro sistema de processamento de transações escrito totalmente em Java [45].

Nos anos seguintes, a *Arjuna Solutions* passou por uma série de aquisições, finalmente se tornando parte da *Hewlett-Packard* (*HP*) em 2001. Dentro da *HP* o *ArjunaTS* continuou a evoluir, passando a oferecer soluções transacionais para *Web services* e computação móvel. No final de 2005, o *ArjunaTS* foi adquirido pelo grupo *JBoss*, que o renomeou para *JBoss Transactions* [39] e o converteu em código aberto. Apesar do produto não se chamar mais *ArjunaTS*, utilizamos esse nome no decorrer do texto por três motivos: *(i)* para distinguir o *ArjunaTS* do gerenciador de transações que utilizamos como base para o nosso trabalho, *(ii)* porque boa parte da literatura utiliza o nome *ArjunaTS* e *(iii)* porque a mudança de nome é um acontecimento recente.

### 2.2.1 Avaliação

O *ArjunaTS* foi idealizado tendo modularidade em mente. O seu principal componente é denominado *ArjunaCore* e consiste num gerenciador transacional que comporta exclusivamente transações locais (isto é, não distribuídas). Para contemplar transações distribuídas, o *ArjunaCore* provê pontos de extensão para que outros módulos complementem a sua funcionalidade. Por exemplo, o *ArjunaJTS* e o *XTS* são dois módulos que possibilitam, juntamente com o *ArjunaCore*, a coordenação de transações distribuídas via *CORBA/IIOP* e *Web services/SOAP*. O *ArjunaTS* possui ainda uma série de outros módulos: o *Transactional Objects*,

que permite a criação de aplicações transacionais utilizando *Plain Old Java Objects* (*POJOs*), o *ORB Portability*, que permite ao *ArjunaJTS* utilizar qualquer *ORB* compatível com a especificação *CORBA*, o SOAP Portability, que é utilizado pelo *XTS* para o envio e o recebimento de mensagens *SOAP*, entre outros. A Figura 2.3 mostra os módulos do *ArjunaTS*. Segundo seus autores [45], a modularização do sistema foi um dos fatores que mais contribuíram para a sua longevidade e o seu sucesso.

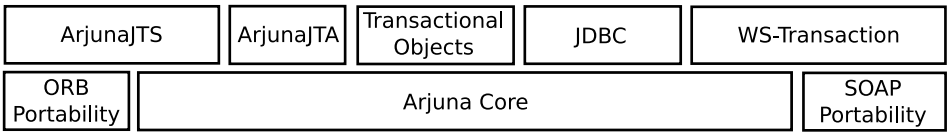



Figura 2.3: Os módulos presentes no *ArjunaTS*.

Além da modularidade, outra característica marcante do *ArjunaTS* é o seu comprometimento com os padrões da indústria. No início de 1997, ele era compatível apenas com o *OTS*. Depois disso, a disseminação de Java fez com que ele passasse a oferecer compatibilidade também com as especificações *JTA* e *JTS*. Em seguida, o *ArjunaTS* passou também a oferecer suporte transacional para *Web services*, tornando-se compatível com as especificações *WS-Coordination*, *WS-AtomicTransaction* e *WS-BusinessActivity*.

O sistema da *Arjuna* implementa também mecanismos avançados de alta disponibilidade. Ele permite, por exemplo, a criação de réplicas dos objetos que participam de uma transação distribuída. Esse recurso é útil quando alguma máquina envolvida numa transação distribuída apresenta um problema. Ao invés de simplesmente abortar a transação devido à máquina com problema, é possível utilizar as réplicas dos objetos para dar continuidade à execução da transação. Esse mecanismo permite que transações sejam efetivadas mesmo na presença de falhas.

O *ArjunaTS* implementa ainda vários mecanismos de otimização que aprimoram o seu desempenho. Entre eles, podemos mencionar o aborto presumido, a efetivação numa só fase e uma otimização para participantes que fazem apenas leituras. Há ainda um outro fator que torna o *ArjunaTS* bastante eficiente: devido à sua arquitetura modular, o gerenciador de transações é um componente leve e compacto, que contém apenas as funcionalidades básicas. Funcionalidades adicionais são incorporadas ao gerenciador de transações apenas quando necessário.

Atualmente, o sistema oferece dois modos de execução: ele pode ser executado integrado a um servidor de aplicações *Java EE* ou como um serviço independente. Para viabilizar o segundo caso, é imprescindível que o *ArjunaTS* seja um sistema completo e independente de

qualquer servidor de aplicações. Por esse motivo, o *ArjunaTS* possui, por exemplo, a sua própria implementação do protocolo *SOAP*.

Por um lado, o fato do *ArjunaTS* possuir a sua própria implementação do protocolo *SOAP* faz com que ele independa de qualquer servidor de aplicações. Além disso, a implementação do protocolo *SOAP* presente no *ArjunaTS* é mais simples e eficiente do que as implementações presentes nos servidores de aplicações. Isso ocorre porque o *ArjunaTS* implementa apenas as funcionalidades do protocolo exigidas por *WS-Coordination*, *WS-AtomicTransaction* e *WS-BusinessActivity*.

Por outro lado, o uso de uma implementação independente do protocolo *SOAP* faz com que existam duas implementações desse protocolo quando o *ArjunaTS* é integrado a um servidor de aplicações. Na realidade, isso não chega a ser uma desvantagem na maioria dos casos. Como desempenho é um requisito fundamental em qualquer sistema gerenciador de transações, o ganho de eficiência proporcionado por uma implementação simplificada do protocolo *SOAP* supera as desvantagens geradas pela existência de duas implementações do protocolo.

## 2.3 O Apache Kandula

O *Apache Kandula*, um projeto da *Apache Software Foundation*, é uma implementação em código aberto das especificações *WS-Coordination* e *WS-AtomicTransaction*[7]. Seu principal objetivo é oferecer uma implementação dessas especificações que seja interoperável com outras implementações, em particular com as da *Microsoft* (*.NET*) e da *IBM* (*WebSphere*). O *Kandula* é implementado em Java e foi desenvolvido utilizando o *Apache Axis* [34] como plataforma para o desenvolvimento e a execução de *Web services*.

O *Apache Axis* é uma plataforma bastante popular para o desenvolvimento e a execução de *Web services*. A sua primeira versão, denominada *Axis 1.x*, é compatível com a especificação *Java API for XML-based RPC* (*JAX-RPC*) [68]. Isso significa que *Web services* desenvolvidos em conformidade com a especificação podem ser implantados no *Axis 1.x* ou em qualquer outro servidor de aplicações compatível com a especificação. Em outras palavras, tais *Web services* são portáveis.

Como a *JAX-RPC* é parte integrante da plataforma *Java EE*, muitos servidores de aplicações utilizam o *Axis* como a implementação padrão dessa *API*. Este é o caso, por exemplo, do *JBoss* (até a versão 4.0.3)[8], do *Apache Geronimo* [49] e do *JOnAS* [35].

[7] O projeto também tem como objetivo implementar a especificação *WS-BusinessActivity*, mas os esforços nessa direção estão apenas começando.

[8] A partir da versão 4.0.4 o *JBoss* não mais utiliza o *Axis* como implementação padrão da *JAX-RPC*.

Infelizmente, o comprometimento do *Axis* com os padrões Java para *Web services* ficou limitado à sua versão 1.x. Para a segunda geração do *Axis*, denominada *Axis 2.x*, a equipe de desenvolvimento decidiu que a implementação não seria influenciada por nenhuma *API* ou especificação. Isso quer dizer que o *Axis 2.x* não é compatível com a *JAX-RPC* e nem com a *Java API for XML Web Services* (*JAX-WS*) [71], que é a especificação sucessora da *JAX-RPC* e parte da plataforma *Java EE 5*. Ao invés disso, o *Axis 2.x* definiu uma *API* própria. Portanto, *Web services* desenvolvidos utilizando o *Axis 2.x* não são portáveis, nem mesmo para o *Axis 1.x*.

Assim sendo, existem atualmente duas versões do *Kandula*: o *Kandula1*, que executa no *Axis 1.x*, e o *Kandula2*, que executa no *Axis 2.x*. Inicialmente havia apenas uma versão do projeto, que executava sobre o *Axis 1.x*. Entretanto, devido ao surgimento do *Axis 2.x* e sua incompatibilidade com a versão 1.x, foi necessário o desenvolvimento de uma nova versão do *Kandula*, compatível com a segunda geração do *Axis*. Essa nova versão do *Kandula* foi denominada *Kandula2*, e é onde hoje se concentram os esforços de desenvolvimento.

### 2.3.1 Avaliação

O *Kandula* apresenta uma implementação razoavelmente completa das especificações *WS-Coordination* e *WS-AtomicTransaction*. Entretanto, como a própria página do projeto na Internet[9] anuncia, a implementação ainda não contempla todos os cenários descritos nas especificações. Comparado a vários outros projetos da *Apache Software Foundation*, pode-se dizer que o *Kandula* se situa num estágio inicial de desenvolvimento, não estando ainda maduro para uso em ambientes de produção. Não há, por exemplo, nenhuma versão estável pré-compilada disponível para *download*. O único modo de se obter o *Kandula* é através do repositório de desenvolvimento. Apesar disso, o *Kandula* já apresenta um razoável conjunto de funcionalidades e é capaz de interoperar em alguns cenários com o *IBM WebSphere*.

A escolha do *Axis* como plataforma de desenvolvimento e execução de *Web services* trouxe algumas conseqüências negativas para o *Kandula*. O *Axis 2.x* não tem mais como um de seus principais objetivos ser compatível com as especificações Java relacionadas a *Web services*, o que resulta em perda de portabilidade para os *Web services* desenvolvidos utilizando o *Axis 2.x*. Uma conseqüência disso é a existência de duas versões do *Kandula*, que utilizam versões distintas do *Axis*. Outra conseqüência é uma maior dificuldade na integração do *Kandula* com servidores de aplicações que não usam o *Axis*. Esse é o caso, por exemplo, do *JBoss*, que recentemente (a partir da versão 4.0.4) abandonou o *Axis* em favor de uma implementação própria focada na compatibilidade com as especificações Java para *Web services*.

[9] `http://ws.apache.org/kandula/`

Apesar de ter sido desenvolvido com o intuito de ser compatível com vários gerenciadores de transações, atualmente o *Kandula* pode ser integrado apenas ao gerenciador de transações presente no servidor de aplicações *Apache Geronimo*, denominado *Java Open Transaction Manager* (*JOTM*) [36]. Contudo, essa integração é ainda limitada. Por exemplo, o *Kandula* registra um recurso transacional com o coordenador da transação prematuramente, no momento em que um *Web service* recebe uma requisição transacional. Esse processo de registro ocorre mesmo quando o *Web service* alvo da requisição não executa nenhum trabalho como parte da transação. O correto seria que o registro do recurso transacional fosse postergado até que o *Web service* executasse algum trabalho como parte da transação — fazendo acesso a um banco de dados, por exemplo. Essa medida permitiria que recursos transacionais fossem registrados com o coordenador apenas quando necessário, garantindo um melhor desempenho durante a execução e o encerramento da transação. Entretanto, para que isso ocorresse com o *Kandula*, ele deveria ser melhor integrado ao *JOTM*.

Outra deficiência do *Kandula* é que ele contempla transações distribuídas envolvendo somente recursos acessíveis como *Web services*. Desse modo, ele não é capaz de coordenar, por exemplo, transações híbridas envolvendo recursos *XA* [75] e recursos acessíveis como *Web services*. Conseqüentemente, podemos dizer que o *Kandula* consiste basicamente numa implementação do protocolo de efetivação em duas fases que contempla exclusivamente recursos acessíveis como *Web services*.

Há ainda uma última limitação do *Kandula*: ele não oferece suporte a recuperação de falhas. Isso ocorre porque o coordenador transacional do *Kandula* trabalha somente com dados em memória volátil e, portanto, na eventualidade de uma queda, tais dados são perdidos. Essa é, sem dúvida, uma das maiores limitações do *Kandula*, pois qualquer gerenciador de transações se torna muito menos atraente quando não oferece suporte a recuperação de falhas. Afinal, o que tipicamente se espera de um gerenciador de transações é que ele mantenha um sistema num estado consistente, inclusive na eventualidade de falhas.

Para que o *Kandula* pudesse sobrepujar as deficiências apresentadas nos três últimos parágrafos, ele precisaria ser intimamente integrado ao *JOTM*. Como parte desse processo de integração, a implementação do protocolo de efetivação em duas fases presente no *JOTM* deveria ser modificada para levar em consideração também os recursos acessíveis como *Web services*. Além disso, o suporte a recuperação de falhas do *JOTM*, que hoje contempla apenas recursos *XA*, precisaria ser estendido para contemplar também os recursos acessíveis como *Web services*. Porém, caso isso ocorresse, o *Kandula* deveria ser visto como um componente do *JOTM* e não como um projeto independente.

Sem uma profunda integração com o *JOTM*, o *Kandula* pode ser apenas o que ele é hoje:

um simples gerenciador de transações que contempla exclusivamente recursos acessíveis como *Web services* e que não oferece suporte a recuperação de falhas. Veremos, no Capítulo 4, que o gerenciador de transações do *JBoss*, juntamente com a extensão desenvolvida como parte desta dissertação, consiste numa solução transacional muito mais completa que o *Kandula*, não apresentando nenhuma das limitações aqui discutidas.

# 3 Transações distribuídas em ambientes Web services

O suporte a transações distribuídas em ambientes *Web services* é definido num conjunto de três especificações, publicadas em agosto de 2005 por um grupo de três empresas (*BEA*, *IBM* e *Microsoft*) e endossadas por outras empresas fornecedoras de sistemas de processamento de transações. Estas especificações são: *WS-Coordination*, *WS-AtomicTransaction* e *WS-BusinessActivity*. Juntas, elas definem um alicerce sobre o qual aplicações confiáveis e robustas baseadas em *Web services* podem ser construídas. Veremos mais detalhes sobre cada uma dessas especificações nas próximas seções. Antes disso, porém, apresentaremos alguns conceitos básicos.

## 3.1 Conceitos básicos

### 3.1.1 Transações

Podemos definir uma transação como um grupo de operações que possui as seguintes características, conhecidas como propriedades *ACID*:

- Atomicidade: ou todas as operações de uma transação são executadas, ou nenhuma delas é executada;
- Consistência: a execução de uma transação mantém o sistema num estado consistente;
- Isolamento: os estados intermediários de uma transação não são visíveis a outras transações;
- Durabilidade: as modificações efetuadas por uma transação bem sucedida são persistentes e sobrevivem a possíveis falhas do sistema.

Uma transação pode ser encerrada de apenas dois modos:

- Ela pode ser efetivada (*committed*): neste caso, todas as ações executadas dentro do escopo da transação tornam-se permanentes;
- Ela pode ser abortada (*aborted* ou *rolled back*): neste caso, todas as ações executadas dentro do escopo da transação são desfeitas.

Informalmente, pode-se pensar numa transação como um bloco atômico de operações que leva o sistema de um estado consistente para outro estado também consistente. O exemplo clássico de transação consiste na transferência de um valor (digamos, X unidades monetárias) entre duas contas-correntes. Tal operação é composta por duas etapas: *(i)* sacar X da primeira conta e *(ii)* depositar X na segunda conta. Se por alguma eventualidade a operação *(i)* for bem sucedida e em seguida (isto é, antes que a operação *(ii)* seja executada) o sistema bancário sofrer uma falha, o valor sacado da primeira conta será perdido. Isso ocorre porque quando o sistema se recuperar da falha, ele não vai "lembrar" que o valor X não foi depositado na segunda conta. O efeito final será o seguinte: o dinheiro foi sacado da primeira conta e simplesmente desapareceu do sistema. Para evitar esse tipo de problema, as operações em questão devem ser agrupadas numa transação.

### 3.1.2 Transações distribuídas

Uma transação pode envolver recursos presentes em um ou mais computadores. Chamamos de transação distribuída uma transação que envolve recursos transacionais (bancos de dados e filas de mensagens, por exemplo) dispersos por vários computadores. No exemplo discutido na seção anterior, caso a transferência fosse feita entre contas-correntes de bancos distintos, ela exigiria a interação com duas bases de dados (uma de cada banco), as quais estariam sendo executadas em computadores distintos. Tratar-se-ia, portanto, de um exemplo de transação distribuída.

Como uma transação distribuída atua sobre recursos transacionais presentes em vários computadores, cada um desses computadores controla apenas um ramo da transação (*transaction branch*). Isso resulta num problema: como fazer com que todos os ramos da transação, que estão espalhados por vários computadores, executem de modo atômico[1]? Não queremos, por exemplo, que alguns ramos sejam abortados e outros efetivados. Para resolver esse problema, precisamos de um mecanismo para garantir que todos os ramos de uma transação distribuída sejam efetivados ou abortados de maneira consistente. O mecanismo mais utilizado para essa finalidade é o protocolo de efetivação em duas fases (*two-phase commit*, ou *2PC*) [28].

[1] Estamos pressupondo que cada computador pode executar o seu ramo da transação de modo atômico. Sem essa suposição não faz sentido tentar executar a transação distribuída de maneira atômica.

Antes de descrever como o *2PC* funciona, é importante ressaltar alguns pontos fundamentais sobre ele:

- O *2PC* pressupõe que cada computador participante da transação distribuída possui um *log* local, cujos registros contêm informações suficientes para colocar o participante num estado consistente na eventualidade de falhas ou quedas do sistema. As informações presentes no *log* são utilizadas pelo procedimento de recuperação, que é executado após uma falha ou queda do participante. Dentre os vários algoritmos que o participante pode usar para se recuperar de falhas ou quedas, destaca-se o *ARIES* [48], que utiliza informações presentes no *log* local do participante para colocá-lo num estado consistente.

- O *2PC* pressupõe também que cada computador participante da transação é capaz de tornar duráveis, porém de modo não definitivo, quaisquer alterações de dados sob sua responsabilidade (atualizações em tabelas de um banco de dados, por exemplo). "Durável, porém de modo não definitivo" significa que, além da alteração ser registrada em memória durável, são também guardadas em memória durável (tipicamente o *log* local do participante) informações suficientes para desfazer a alteração, pois esta ainda não é considerada definitiva.

- Finalmente, o protocolo pressupõe que um dos computadores que participam da transação distribuída desempenha um papel especial, sendo denominado coordenador. Ele é o responsável por decidir se a transação distribuída deve ser efetivada ou abortada.

Podemos agora descrever o protocolo *2PC*. Como seu próprio nome sugere, o protocolo possui duas fases:

1. Fase de votação: o coordenador inicia a efetivação de uma transação distribuída T. Ele grava no seu *log* uma entrada `[Prepare T]` e em seguida envia mensagens `Prepare T` a cada um dos participantes (recursos transacionais) da transação T. Um participante que receba a mensagem `Prepare T` deve decidir se deseja efetivar ou abortar a transação T. Caso o participante deseje efetivar a transação T, ele torna duráveis, porém de modo não definitivo, as alterações de dados sob sua responsabilidade efetuadas como parte dessa transação. Após fazer isso, ele adiciona ao seu *log* local uma entrada `[Ready T]` e informa ao coordenador que ele vota "sim" para a efetivação de T. Por outro lado, se o participante decidir abortar a transação T ou se, por qualquer motivo, ele não puder tornar duráveis, mas não definitivas, as alterações de dados sob sua responsabilidade efetuadas como parte de T, então o participante desfaz essas alterações. Após isso, ele

escreve em seu *log* local uma entrada `[Don't commit T]` e informa ao coordenador que ele vota "não" para a efetivação de T.

2. Fase de efetivação: o coordenador faz a apuração dos votos. Se todos os participantes votaram "sim", então a transação distribuída T será efetivada, caso contrário ela será abortada. No caso de uma decisão pela efetivação da transação, o coordenador grava em seu *log* uma entrada `[Commit T]` e envia mensagens `Commit T` para todos os participantes. Ao receber uma mensagem `Commit T`, o participante deve efetivar o seu ramo da transação T (tornando definitivas as alterações de dados sob sua responsabilidade) e registrar em seu *log* uma entrada `[Commit T]`. Já no caso de uma decisão por abortar a transação, o coordenador grava em seu *log* uma entrada `[Rollback T]` e envia mensagens `Rollback T` para os participantes que votaram "sim". Ao receber uma mensagem `Rollback T`, o participante deve abortar o seu ramo da transação (desfazendo as alterações de dados sob sua responsabilidade) e registrar em seu *log* uma entrada `[Rollback T]`.

A Figura 3.1 ilustra as mensagens trocadas entre o coordenador e os participantes de uma transação distribuída nas duas fases do *2PC*. É importante ressaltar que, durante a execução desse protocolo, ocorrem escritas nos *logs* do coordenador e dos participantes. Os dados escritos nesses *logs* são a chave para a atomicidade e a durabilidade das transações distribuídas. Em particular, o "ponto de não retorno" de uma transação distribuída T é o momento em que ocorre a escrita do registro `[Commit T]` no *log* do coordenador. Isso significa que, após a escrita desse registro no *log* do coordenador, o desfecho da transação distribuída tem de ser a efetivação, mesmo que ocorram quedas ou falhas do coordenador e/ou dos participantes.

## 3.2 WS-Coordination

É bastante comum, em aplicações distribuídas, que a execução de certas atividades exija interações complexas entre os componentes de um sistema ou até mesmo entre sistemas distintos. Tomemos como exemplo uma atividade fictícia de compra, realizada entre um cliente e um fornecedor (Figura 3.2). A primeira ação que o sistema do cliente deve tomar é requisitar ao sistema do fornecedor um orçamento dos produtos desejados. Em seguida, caso deseje prosseguir com a compra, o sistema do cliente deve enviar uma solicitação de compra ao sistema do fornecedor. Por fim, o sistema cliente deve interagir com um sistema bancário para efetuar o pagamento e então confirmar o pagamento ao sistema do fornecedor. O processo de compra se encerra com essa última interação.

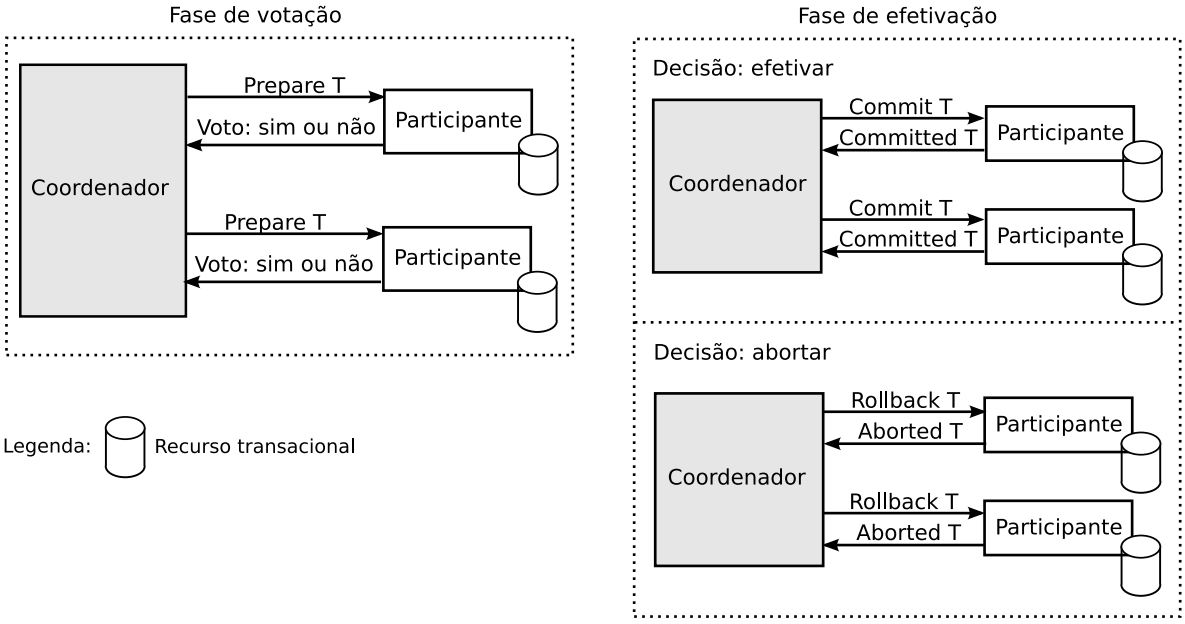



Figura 3.1: As mensagens trocadas entre o coordenador e os participantes de uma transação nas duas fases do *2PC*.

Cabe ressaltar que, na atividade mencionada, a ordem em que as requisições são efetuadas é importante. Por exemplo, não faz sentido contactar o sistema bancário para efetuar um pagamento se um pedido de compra não foi feito anteriormente. Nesse caso, um mecanismo de coordenação é necessário para assegurar que as requisições sejam executadas numa ordem correta. Trata-se, portanto, de um exemplo de atividade que precisa ser coordenada.

Atividades coordenadas envolvem três abstrações: o coordenador, o participante e o protocolo de coordenação (Figura 3.3). O primeiro atua como um mediador [27] entre os participantes. Já o último define: *(i)* as mensagens que podem ser trocadas entre o coordenador e os

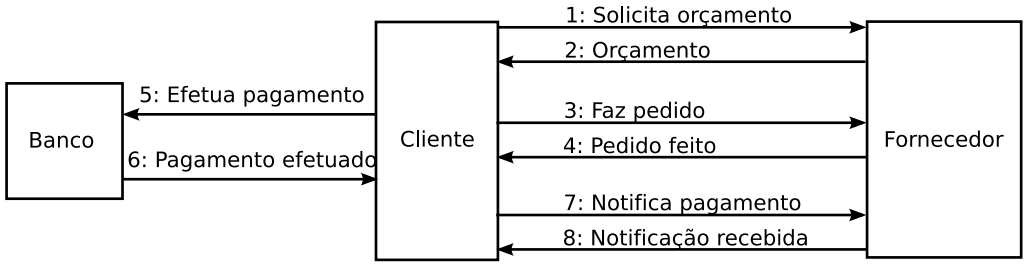



Figura 3.2: As operações envolvidas numa atividade fictícia de compra.

participantes, *(ii)* as regras que tais trocas de mensagens devem obedecer (por exemplo, uma certa ordem) e *(iii)* a semântica das mensagens trocadas entre o coordenador e os participantes.

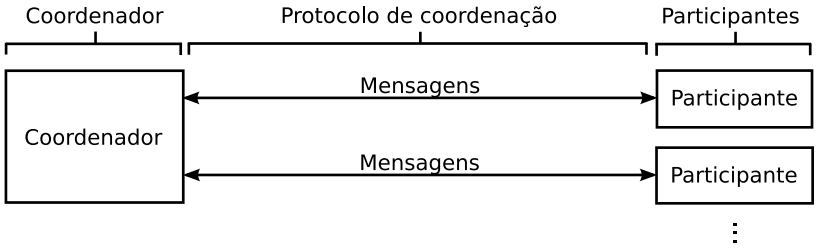



Figura 3.3: As abstrações envolvidas numa atividade coordenada.

Um exemplo bastante conhecido de protocolo de coordenação é o *2PC*, empregado no desfecho de transações atômicas distribuídas. O protocolo define: *(i)* as mensagens que podem ser trocadas entre o coordenador e os participantes de uma transação distribuída (`Prepare`, `Commit`, `Rollback`, etc.), *(ii)* uma ordem para a troca de tais mensagens (por exemplo, uma mensagem `Commit` só pode ser enviada a um participante após uma mensagem `Prepare`) e *(iii)* a finalidade (semântica) de cada uma dessas mensagens (por exemplo, uma mensagem `Commit` indica a um participante que ele deve efetivar o seu ramo da transação distribuída).

*WS-Coordination* define um coordenador de atividades genérico e extensível. Esse coordenador é na realidade um arcabouço, que só é útil quando a sua funcionalidade é estendida. A funcionalidade do coordenador pode ser ampliada através de protocolos de coordenação, que são definidos em outras especificações (ou seja, *WS-Coordination* não define nenhum protocolo de coordenação). Por exemplo, *WS-AtomicTransaction* define protocolos de coordenação que habilitam o coordenador genérico a lidar com transações atômicas. Do mesmo modo, *WS-BusinessActivity* define protocolos de coordenação que habilitam o coordenador genérico a lidar com transações distribuídas de longa duração. Portanto, através da definição de novos protocolos de coordenação, esse coordenador genérico pode ser utilizado para orquestrar virtualmente qualquer tipo de atividade. Informalmente, podemos dizer que a funcionalidade do coordenador pode ser expandida plugando-se novos protocolos de coordenação a ele (Figura 3.4).

Resumidamente, a especificação *WS-Coordination* define os seguintes itens:

- Um formato para o contexto de coordenação, que é utilizado para transportar informações relevantes aos potenciais[2] participantes de uma atividade coordenada.

[2] O termo "potenciais" se faz necessário porque nem todos os *Web services* que recebem o contexto transacional precisam se registrar com o coordenador e participar da atividade coordenada.

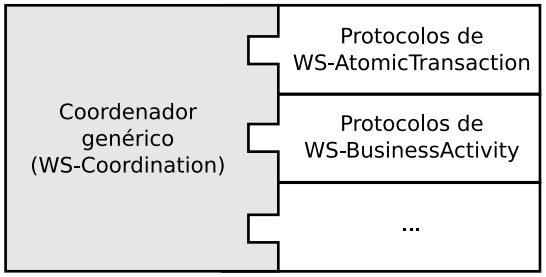



Figura 3.4: A arquitetura extensível do coordenador de atividades definido em *WS-Coordination*.

- Um serviço de ativação (*activation service*), que possibilita a criação de novos contextos de coordenação.
- Um serviço de registro (*registration service*), através do qual *Web services* podem se registrar como participantes de uma atividade coordenada.

Os serviços de ativação e registro, que são implementados utilizando *Web services*, compõem o coordenador genérico definido em *WS-Coordination*. É importante mencionar que a interação com esses dois serviços é feita através de mensagens assíncronas, sendo o suporte mensagens síncronas opcional. Por esse motivo, abordaremos nesta dissertação apenas as interações assíncronas com tais serviços. Vale ressaltar, contudo, que o caso síncrono funciona de modo análogo.

### 3.2.1 O contexto de coordenação

*WS-Coordination* define um contexto de coordenação para a propagação de informações aos potenciais participantes de uma atividade coordenada. Esse contexto contém informações tais como o identificador da atividade, o tipo de coordenação (por exemplo, uma transação atômica) e o endereço de um serviço com o qual os participantes da atividade podem se registrar (o serviço de registro). Desse modo, um *Web service* que recebe tal contexto possui todas as informações necessárias para poder participar da atividade representada pelo contexto. Pode-se dizer, portanto, que é a propagação de tal contexto que viabiliza a participação de vários *Web services* numa mesma atividade coordenada.

O contexto de coordenação é propagado no cabeçalho das mensagens *SOAP*. As requisições *SOAP* efetuadas dentro do escopo de uma atividade coordenada devem obrigatoriamente incluir o contexto de coordenação. Por exemplo, se um *Web service* inicia uma atividade coordenada e em seguida efetua uma chamada a um outro *Web service*, a mensagem *SOAP*

destinada a este *Web service* deverá incluir em seu cabeçalho o contexto de coordenação (Figura 3.5).

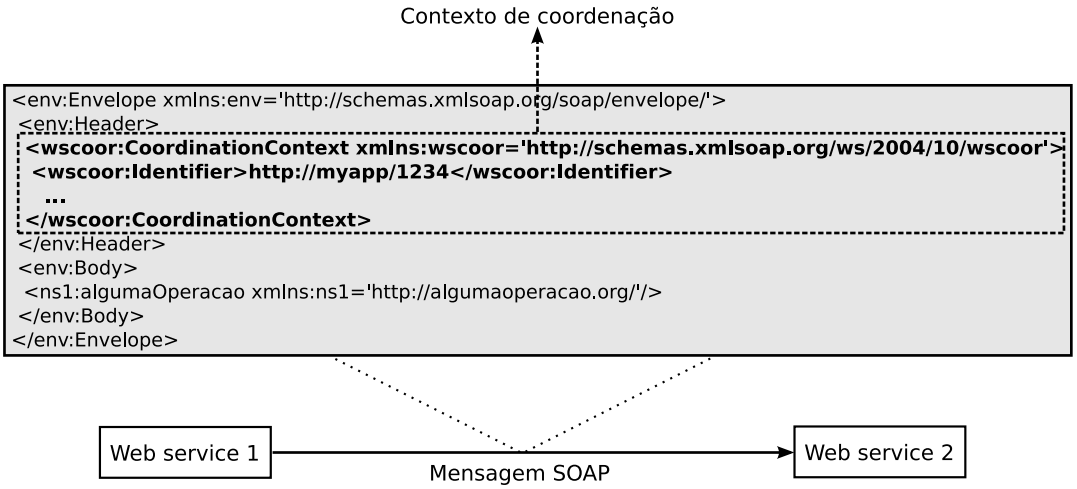



Figura 3.5: O *Web service* 1 inicia uma atividade coordenada e a seguir envia uma mensagem ao *Web service* 2. Como a requisição ocorre dentro do escopo da atividade coordenada, o contexto de coordenação é propagado no cabeçalho da mensagem *SOAP*.

O contexto de coordenação é uma estrutura de dados definida em *XML Schema* [11, 76] e, portanto, suas instâncias são representadas por excertos de documentos *XML*. A Figura 3.6 mostra um exemplo de contexto de coordenação. Vejamos com mais detalhe quais são as informações presentes num contexto de coordenação:

- O identificador da atividade (`Identifier`): uma *URI* [9], única no contexto de um coordenador, que identifica a atividade.
- O tempo de expiração da atividade (`Expires`): um inteiro positivo que especifica o tempo, em milésimos de segundo, de validade do contexto de coordenação. Após transcorrido esse tempo a atividade expira e o contexto torna-se inválido.
- O tipo de coordenação (`CoordinationType`): uma *URI* que identifica o tipo de coordenação, isto é, o tipo de atividade que o contexto representa. Esse campo pode conter, por exemplo, a *URI* `http://schemas.xmlsoap.org/ws/2004/10/wsat`[3] definida em *WS-AtomicTransaction*, indicando que a atividade descrita pelo contexto é uma transação atômica. Alternativamente, esse campo pode conter a *URI* `http://schemas.`

[3]Essa *URI* é utilizada apenas como um identificador, ou seja, ela não necessariamente representa um recurso acessível através da *Web*.

`xmlsoap.org/ws/2004/10/wsba/MixedOutcome` definida em *WS-BusinessActivity*, explicitando que a atividade descrita pelo contexto é uma transação de longa duração.

- Uma referência para o serviço de registro (`RegistrationService`): os participantes podem utilizar tal referência para se registrar como participantes de um ou mais protocolos de coordenação (mais detalhes na Seção 3.2.3).

- Elementos de extensão: o contexto de coordenação definido por *WS-Coordination* é extensível, ou seja, ele pode conter outros elementos além dos já citados. Isso quer dizer que outras especificações podem adicionar as informações que julgarem relevantes ao contexto e que, portanto, o conteúdo de um contexto de coordenação não fica limitado aos elementos básicos definidos em *WS-Coordination* (identificador, tempo de vida, tipo de coordenação e endereço do serviço de registro). Cabe ressaltar que o exemplo da Figura 3.6 não contém nenhum elemento de extensão.

```
<wscoor:CoordinationContext mustUnderstand='true'
 xmlns:wscoor='http://schemas.xmlsoap.org/ws/2004/10/wscoor'>
 <wscoor:Identifier formatID='257'>gid:192.168.1.1:1099/3</wscoor:Identifier>
 <wscoor:Expires>299952</wscoor:Expires>
 <wscoor:CoordinationType>http://schemas.xmlsoap.org/ws/2004/10/wsat</wscoor:CoordinationType>
 <wscoor:RegistrationService xmlns:wsa='http://schemas.xmlsoap.org/ws/2004/08/addressing'>
  <wsa:Address>http://192.168.1.1:8080/jbosssws/wscoor/RegistrationCoordinator</wsa:Address>
  <wsa:ReferenceProperties xmlns:jbosswsc='http://www.jboss.org/wscoor/extension'>
   <jbosswsc:CoordinatedActivityID>3</jbosswsc:CoordinatedActivityID>
  </wsa:ReferenceProperties>
 </wscoor:RegistrationService>
</wscoor:CoordinationContext>
```

Figura 3.6: Um exemplo de contexto de coordenação.

É importante ressaltar que o contexto de coordenação definido por *WS-Coordination* contém um tipo de coordenação, e não um protocolo de coordenação. Este último define a semântica e as regras para as trocas de mensagens entre o coordenador e os participantes de uma atividade coordenada (Figura 3.3), ou seja, ele define quais mensagens podem ser trocadas entre o coordenador e os participantes num dado momento. Já um tipo de coordenação (que é o que um contexto de coordenação contém) representa um conjunto de protocolos de coordenação logicamente relacionados. Por exemplo, conforme veremos adiante, há dois protocolos de coordenação envolvidos na execução de uma transação atômica, denominados *completion* e *2PC*. O conjunto desses dois protocolos (que estão logicamente relacionados, pois ambos são essenciais para a execução de uma transação atômica) define o tipo de coordenação "transação

atômica". O contexto de coordenação precisa conduzir o tipo de coordenação, ou seja, o tipo da atividade que o contexto representa. Assim, se um contexto especifica que seu tipo de coordenação é `http://schemas.xmlsoap.org/ws/2004/10/wsat` (esse é o tipo de coordenação definido em *WS-AtomicTransaction*), pode-se deduzir que: *(i)* o contexto representa uma transação atômica e *(ii)* a comunicação entre o coordenador e os participantes da atividade ocorre através dos protocolos *completion* e *2PC*. A Figura 3.7 apresenta os tipos de coordenação e os protocolos de coordenação definidos em *WS-AtomicTransaction* e *WS-BusinessActivity*.

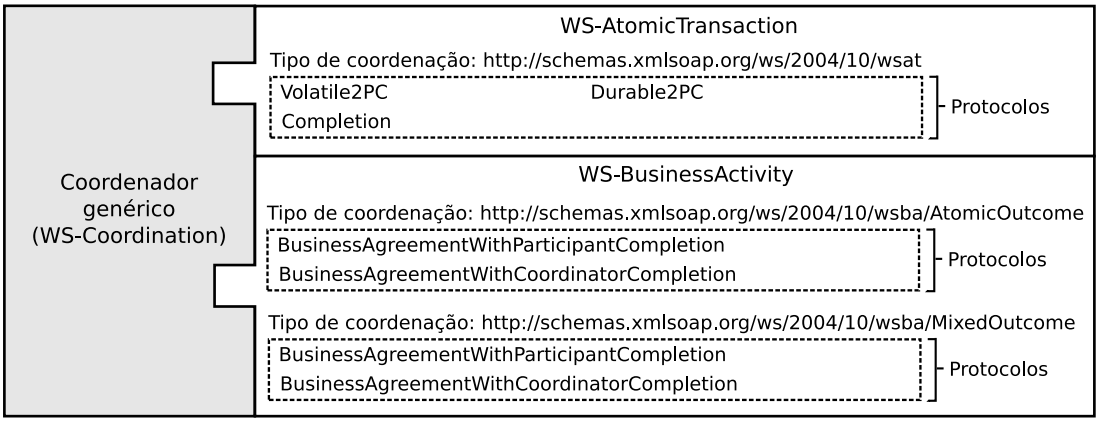



Figura 3.7: Os tipos de coordenação e os protocolos de coordenação definidos em *WS-AtomicTransaction* e *WS-BusinessActivity*.

Um *Web service* que recebe uma requisição *SOAP* contendo um contexto de coordenação deve: *(i)* identificar e processar o contexto de coordenação, *(ii)* verificar se o *Web service* deve participar da atividade coordenada e, em caso afirmativo, *(iii)* registrar-se com o coordenador da atividade como participante de algum protocolo de coordenação. Dado que a identificação e o processamento do contexto de coordenação são tarefas obrigatórias para o destinatário de uma mensagem *SOAP*, o elemento `CoordinationContext` propagado no cabeçalho da mensagem *SOAP* deve possuir o atributo `mustUnderstand="true"` (vide Figura 3.6). Caso o *Web service* que receba a requisição não seja capaz de lidar com o contexto de coordenação, ele deve devolver ao remetente uma indicação de erro.

### 3.2.2 O serviço de ativação

O serviço de ativação possibilita a criação de contextos de coordenação. Ele define dois papéis: o coordenador, responsável pela criação de novos contextos, e o requerente, que solicita a criação de novos contextos. *WS-Coordination* define *port types* para cada um desses pa-

péis, sendo tais *port types* denominados `ActivationCoordinatorPortType` e `ActivationRequesterPortType`. O primeiro *port type* deve ser implementado pelo coordenador e possui uma única operação denominada `CreateCoordinationContext`. Já o segundo deve ser implementado pelo requerente e possui uma única operação denominada `CreateCoordinationContextResponse`.

O fato do coordenador implementar a operação `CreateCoordinationContext` presente no `ActivationCoordinatorPortType` é esperado, pois ele deve permitir a criação de novos contextos de coordenação. O fato do requerente também precisar implementar um *port type*, contudo, pode parecer estranho à primeira vista. Se o requerente vai apenas solicitar ao coordenador a criação de um novo contexto de coordenação, por que ele precisa implementar o `ActivationRequesterPortType`? O motivo é que as interações com os serviços definidos em *WS-Coordination* ocorrem através de mensagens assíncronas. Para criar um novo contexto de coordenação, o requerente envia uma mensagem `CreateCoordinationContext` ao `ActivationCoordinatorPortType` do coordenador. Todavia, o requerente não recebe a resposta de modo síncrono. Ao invés disso, o coordenador responde assincronamente enviando uma mensagem `CreateCoordinationContextResponse` ao `ActivationRequesterPortType` do requerente. Portanto, o requerente precisa implementar o `ActivationRequesterPortType` para receber a resposta assíncrona do coordenador. A Figura 3.8 ilustra a interação entre um requerente e o coordenador do serviço de ativação.

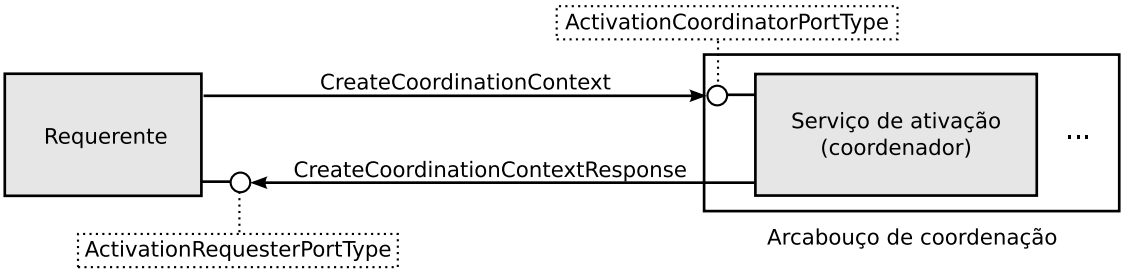



Figura 3.8: Criação de um contexto de coordenação através do serviço de ativação.

A mensagem `CreateCoordinationContext` que o requerente envia ao coordenador do serviço de ativação deve especificar um tipo de coordenação. Em outras palavras, essa mensagem deve especificar qual o tipo do contexto a ser criado (transação atômica, transação distribuída de longa duração, etc.). A Figura 3.9 mostra uma mensagem `CreateCoordinationContext` [4]

[4] Omitiremos a partir desse ponto, por questão de brevidade, as declarações dos espaços de nomes *XML*. Cabe lembrar, portanto, que o prefixo `wscoor` está associado ao espaço de nomes `http://schemas.xmlsoap.org/ws/2004/10/wscoor`, que o prefixo `wsa` está associado ao espaço de nomes

que especifica o tipo de coordenação `http://schemas.xmlsoap.org/ws/2004/10/wsat`. A presença de tal *URI* indica ao coordenador que ele deve criar um contexto de coordenação para uma transação atômica.

```
<wscoor:CreateCoordinationContext>
 <wscoor:CoordinationType>
  http://schemas.xmlsoap.org/ws/2004/10/wsat
 </wscoor:CoordinationType>
</wscoor:CreateCoordinationContext>
```

Figura 3.9: Um exemplo de mensagem `CreateCoordinationContext`.

Após enviar uma mensagem `CreateCoordinationContext` ao coordenador, o requerente recebe como resposta uma mensagem `CreateCoordinationContextResponse`, que contém o contexto de coordenação recém-criado. Um exemplo de mensagem `CreateCoordination-ContextResponse` é mostrado na Figura 3.10. Vale ressaltar que essa mensagem contém o endereço do serviço de registro (elemento `RegistrationService`), o qual é apresentado na próxima seção. Portanto, após criar criar um novo contexto de coordenação, o requerente pode, se desejar, contactar o serviço de registro.

```
<wscoor:CreateCoordinationContextResponse>
 <wscoor:CoordinationContext>
  <wscoor:Identifier formatID='257'>gid:192.168.1.1:1099/3</wscoor:Identifier>
  <wscoor:Expires>299952</wscoor:Expires>
  <wscoor:CoordinationType>
   http://schemas.xmlsoap.org/ws/2004/10/wsat
  </wscoor:CoordinationType>
  <wscoor:RegistrationService>
   <wsa:Address>http://192.168.1.1:8080/jbossws/wscoor/RegistrationCoordinator</wsa:Address>
   <wsa:ReferenceProperties>
    <jbosswsc:CoordinatedActivityID>3</jbosswsc:CoordinatedActivityID>
   </wsa:ReferenceProperties>
  </wscoor:RegistrationService>
 </wscoor:CoordinationContext>
</wscoor:CreateCoordinationContextResponse>
```

Figura 3.10: Um exemplo de mensagem `CreateCoordinationContextResponse`.

`http://schemas.xmlsoap.org/ws/2004/08/addressing` e que o prefixo `jbosswsc` está associado ao espaço de nomes `http://www.jboss.org/wscoor/extension`.

### 3.2.3 O serviço de registro

O serviço de registro permite que *Web services* se registrem como participantes de uma atividade coordenada. Assim como o serviço de ativação, esse serviço define dois papéis: o coordenador, com quem os participantes se registram, e o participante, que solicita o registro numa atividade coordenada. *WS-Coordination* define *port types* para cada um desses papéis, sendo tais *port types* denominados `RegistrationCoordinatorPortType` e `RegistrationRequesterPortType`. O primeiro deve ser implementado pelo coordenador e possui uma única operação denominada `Register`. Já o segundo deve ser implementado pelo participante e possui uma única operação denominada `RegisterResponse`. O participante precisa implementar o `RegistrationRequesterPortType` para receber as respostas assíncronas enviadas pelo coordenador.

Um participante que deseja se registrar numa atividade coordenada deve enviar uma mensagem `Register` ao `RegistrationCoordinatorPortType` do coordenador. O coordenador então responde assincronamente enviando uma mensagem `RegisterResponse` ao `RegistrationRequesterPortType` do participante. A Figura 3.11 apresenta as mensagens envolvidas no registro de um participante numa atividade coordenada.

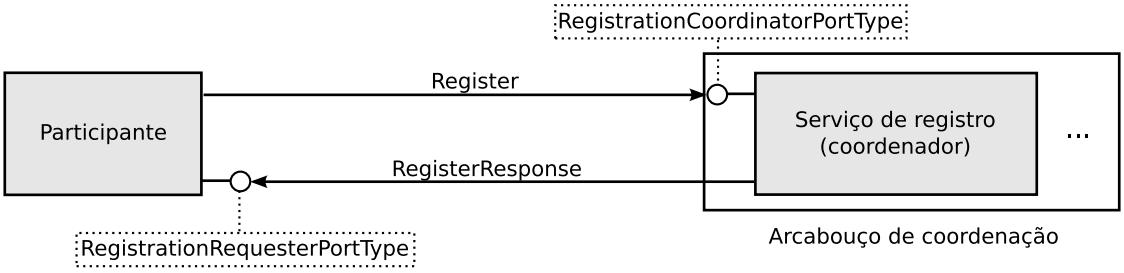



Figura 3.11: O processo de registro de um participante numa atividade coordenada.

Cabe ressaltar que, para poder se registrar numa atividade coordenada, o participante precisa obter o contexto de coordenação de tal atividade. Ele pode fazer isso de dois modos: *(i)* via serviço de ativação ou *(ii)* extraindo o contexto do cabeçalho de uma requisição *SOAP*. O contexto de coordenação é necessário pois é nele que o participante encontra o endereço do serviço de registro.

A mensagem `Register` que o participante envia ao coordenador deve conter:

1. Um identificador do protocolo de coordenação: o participante deve informar de qual protocolo de coordenação ele pretende participar. Tal protocolo determinará quais as mensagens que serão trocadas entre o participante e o coordenador do protocolo. Por exemplo,

se a mensagem `Register` do participante especificar o protocolo de coordenação *2PC*, o coordenador desse protocolo e o participante trocarão mensagens específicas do *2PC* (`Prepare`, `Commit`, `Rollback`, etc.).

2. O endereço de um *port type* específico do protocolo: esse *port type* não é definido em *WS-Coordination*, mas sim em outras especificações, como por exemplo *WS-AtomicTransaction* e *WS-BusinessActivity*. Quando um participante se registra em um protocolo de coordenação, ele deve informar o endereço de um *port type* capaz de receber as mensagens enviadas pelo coordenador do protocolo. Por exemplo, no caso do *2PC*, o participante deve fornecer o endereço de seu `ParticipantPortType`, que pode receber as mensagens `Prepare`, `Commit` e `Rollback`. Tais mensagens são enviadas pelo coordenador do protocolo *2PC* durante o processo de encerramento da transação.

A Figura 3.12 apresenta um exemplo de mensagem `Register`. Nesse exemplo, a mensagem `Register` especifica que o participante deseja se registrar no protocolo *2PC*, e que o coordenador desse protocolo deve enviar as mensagens ao *port type* localizado no endereço `http://192.168.1.3:8080/jbossws/wsat/Participant`.

```
<wscoor:Register>
 <wscoor:ProtocolIdentifier>
  http://schemas.xmlsoap.org/ws/2004/10/wsat/Durable2PC
 </wscoor:ProtocolIdentifier>
 <wscoor:ParticipantProtocolService>
  <wsa:Address>http://192.168.1.3:8080/jbossws/wsat/Participant</wsa:Address>
  <wsa:ReferenceProperties>
   <jbosswsc:CoordinatedActivityID>3</jbosswsc:CoordinatedActivityID>
  </wsa:ReferenceProperties>
 </wscoor:ParticipantProtocolService>
</wscoor:Register>
```

Figura 3.12: Um exemplo de mensagem `Register`.

Após enviar uma mensagem `Register`, o participante recebe como resposta uma mensagem assíncrona `RegisterResponse`. Essa mensagem contém o endereço do coordenador do protocolo no qual o participante se registrou. O participante utiliza esse endereço para enviar mensagens ao coordenador do protocolo. No caso do *2PC*, esse endereço especifica a localização de um `CoordinatorPortType`, que permite ao participante enviar ao coordenador do *2PC* as seguintes mensagens: `Prepared`, `Aborted`, `ReadOnly`, `Committed` e `Replay`. Um exemplo de mensagem `RegisterResponse` é apresentado na Figura 3.13.

É importante notar que, após o registro de um participante num protocolo de coordenação, tal participante conhece o endereço do coordenador do protocolo (que veio na mensagem

```
<wscoor:RegisterResponse>
 <wscoor:CoordinatorProtocolService>
  <wsa:Address>http://192.168.1.1:8080/jbossws/wsat/Coordinator</wsa:Address>
  <wsa:ReferenceProperties>
   <jbosswsc:ParticipantID>00000000-0000-0005-46a8-040f00000001</jbosswsc:ParticipantID>
  </wsa:ReferenceProperties>
 </wscoor:CoordinatorProtocolService>
</wscoor:RegisterResponse>
```

Figura 3.13: Um exemplo de mensagem `RegisterResponse`.

`RegisterResponse`). Este último, por sua vez, conhece o endereço do participante (presente na mensagem `Register`). Desse modo, o processo de registro de um participante permite que este e o coordenador do protocolo informem seus endereços um ao outro. A partir daí, eles podem trocar as mensagens definidas pelo protocolo de coordenação (por exemplo, o *2PC*). A Figura 3.14 ilustra esse processo.

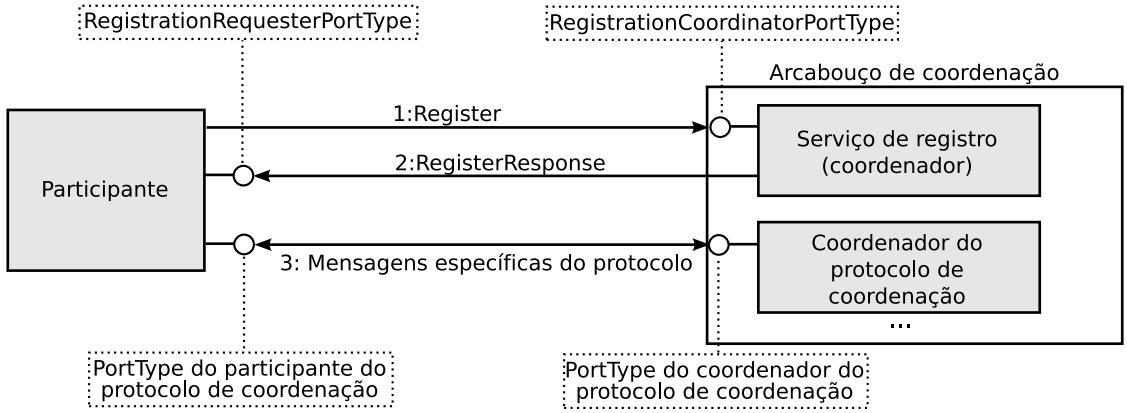



Figura 3.14: A troca de endereços durante o processo de registro permite que o participante e o coordenador troquem mensagens específicas do protocolo no qual aquele se registrou.

### 3.2.4 O que não é especificado em WS-Coordination

*WS-Coordination* define apenas o formato do contexto de coordenação e um coordenador genérico, composto pelos serviços de ativação e registro. Apesar desse coordenador permitir sua extensão através de protocolos de coordenação, a especificação não define nenhum desses protocolos. Tal papel é desempenhado por outras especificações, como por exemplo *WS-AtomicTransaction* e *WS-BusinessActivity*. *WS-Coordination* permite, portanto, a criação de contextos e o registro de participantes com um coordenador, e nada mais. A partir daí, as

mensagens trocadas entre coordenador e participante são dependentes de um protocolo de coordenação, e fogem do escopo de *WS-Coordination*.

Podemos concluir, portanto, que devido a não especificação de protocolos de coordenação, *WS-Coordination* é uma especificação (propositadamente) incompleta. Em outras palavras, uma implementação de *WS-Coordination* não faz sentido por si só, dado que ela só permite a criação de contextos e o registro de participantes. Isso está longe de ser um problema, pois o coordenador definido em *WS-Coordination* tem como objetivo ser extensível e delega a responsabilidade de definir protocolos de coordenação a outras especificações. Assim sendo, para compreender como uma atividade coordenada prossegue após o registro dos participantes, precisamos analisar alguma outra especificação que estenda *WS-Coordination*. Para esse fim, examinaremos a seguir a especificação *WS-AtomicTransaction*.

## 3.3 WS-AtomicTransaction

*WS-AtomicTransaction* foi o primeiro serviço definido sobre a infra-estrutura oferecida por *WS-Coordination*. A especificação define um tipo de coordenação "transação atômica", representado pela *URI* `http://schemas.xmlsoap.org/ws/2004/10/wsat`. Esse tipo de coordenação habilita o coordenador genérico de *WS-Coordination* a lidar com transações atômicas distribuídas. Resumidamente, *WS-AtomicTransaction* define:

- Um tipo de coordenação, representado pela *URI* `http://schemas.xmlsoap.org/ws/2004/10/wsat`.
- Dois protocolos de coordenação: o *completion*, utilizado para encerrar uma transação, e o *2PC*. Este último possibilita que múltiplos participantes cheguem a um consenso sobre o desfecho de uma transação. Na realidade, *WS-AtomicTransaction* define duas variações do *2PC*, denominadas *2PC* durável e *2PC* volátil. Essas duas variações serão apresentadas mais adiante.
- Um conjunto de *port types*, que devem ser implementados pelos participantes e coordenadores dos protocolos *completion* e *2PC*.

Esses itens representam praticamente tudo o que *WS-AtomicTransaction* define, pois outras funcionalidades, como a criação de contextos e o registro de participantes, são contempladas pelos serviços definidos em *WS-Coordination*.

### 3.3.1 O contexto transacional

O contexto transacional definido em *WS-AtomicTransaction* é o contexto de coordenação definido em *WS-Coordination*, com as seguintes especificações adicionais:

- O tempo de expiração da atividade (`Expires`) representa o tempo a partir do qual uma transação pode ser abortada por duração excessiva.
- O tipo de coordenação (`CoordinationType`) deve conter a seguinte *URI*, que representa o tipo de coordenação "transação atômica":

```
http://schemas.xmlsoap.org/ws/2004/10/wsat
```

O exemplo da Figura 3.6 é, na realidade, um contexto transacional compatível com as regras descritas em *WS-AtomicTransaction*. O contexto transacional deve ser propagado juntamente com todas as requisições *SOAP* efetuadas dentro do escopo de uma transação.

### 3.3.2 Os protocolos de coordenação definidos por WS-AtomicTransaction

*WS-AtomicTransaction* define dois protocolos de coordenação, denominados *completion* e *2PC*, sendo que este último apresenta duas variações: o *2PC* durável e o *2PC* volátil. Vamos analisar a seguir os protocolos *completion* e *2PC*.

#### 3.3.2.1 O protocolo completion

O protocolo *completion* é utilizado para encerrar (efetivar ou abortar) uma transação. Assim como os serviços de ativação e registro de *WS-Coordination*, esse protocolo também define dois papéis: o coordenador, a quem os participantes solicitam o encerramento de uma transação, e o participante (também denominado cliente ou iniciador), que é quem solicita o encerramento da transação. *WS-AtomicTransaction* define *port types* para cada um desses papéis, sendo tais *port types* denominados `CompletionCoordinatorPortType` e `CompletionInitiatorPortType`. O primeiro possui as operações `Commit` e `Rollback` e deve ser implementado pelo coordenador. O segundo possui as operações `Committed` e `Aborted` e deve ser implementado pelo participante. O participante precisa implementar o *port type* `CompletionInitiatorPortType` para poder receber a resposta assíncrona enviada pelo coordenador.

O protocolo *completion* é utilizado por um participante do seguinte modo: primeiramente, o participante envia ao coordenador uma mensagem `Commit` ou uma mensagem `Rollback`. A mensagem enviada determina se o coordenador irá tentar efetivar a transação ou se ele

irá abortá-la. Após receber uma dessas mensagens, o coordenador tomará as providências para encerrar a transação. Tais providências incluirão, por exemplo, a execução do protocolo *2PC*. Depois de encerrar a transação, o coordenador devolve ao participante uma mensagem `Committed` ou `Aborted`, indicando se a transação foi efetivada ou abortada. Cabe ressaltar que no caso em que o coordenador recebe uma mensagem `Commit`, a resposta enviada ao participante pode ser `Committed` ou `Aborted`. Já no caso de uma mensagem `Rollback`, a resposta só pode ser `Aborted`. Outro ponto a ser destacado é que as providências tomadas pelo coordenador para encerrar a transação são irrelevantes para o participante. Este último está interessado apenas no desfecho da transação, e não no modo como tal desfecho é alcançado. A Figura 3.15 mostra as mensagens envolvidas no protocolo *completion*.

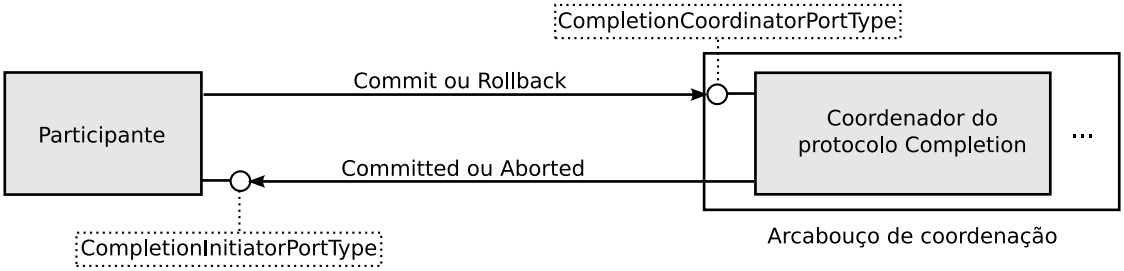



Figura 3.15: As mensagens envolvidas no protocolo *completion*.

#### 3.3.2.2 O protocolo de efetivação em duas fases

O protocolo de efetivação em duas fases (*2PC*) define um mecanismo para que múltiplos participantes cheguem a um consenso sobre o desfecho de uma transação. Assim como o protocolo *completion*, o *2PC* também define dois papéis: o coordenador, responsável por decidir se a transação distribuída deve ser efetivada ou abortada, e o participante, que representa um recurso transacional. *WS-AtomicTransaction* define *port types* para cada um desses papéis, sendo tais *port types* denominados `CoordinatorPortType` e `ParticipantPortType`. O primeiro possui cinco operações: `Prepared`, `Aborted`, `ReadOnly`, `Committed` e `Replay`, e deve ser implementado pelo coordenador. Já o segundo possui três operações: `Prepare`, `Commit` e `Rollback`, e deve ser implementado pelo participante.

Vimos, na Seção 3.1.2, uma descrição abstrata do protocolo *2PC*. Em termos dos *port types* e das operações definidas por *WS-AtomicTransaction*, estas são as interações que ocorrem em cada uma das duas fases do protocolo[5]:

[5] As operações de escrita nos *logs*, já detalhadas na Seção 3.1.2, foram omitidas desta descrição.

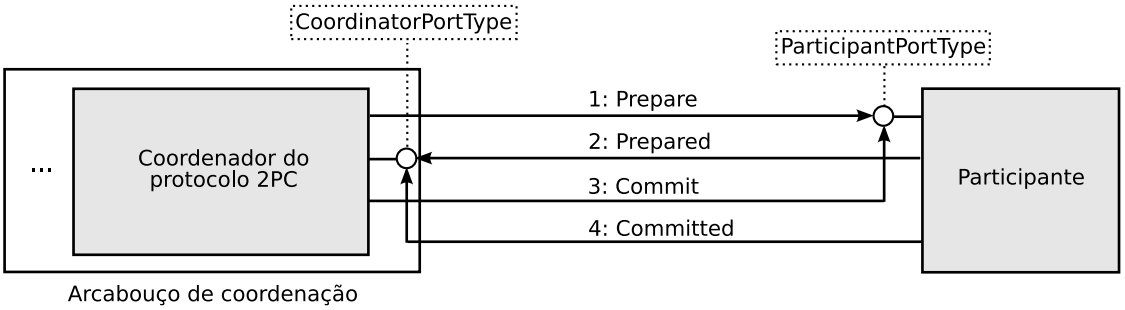
1. Fase de votação: o coordenador inicia a efetivação de uma transação atômica distribuída e envia mensagens `Prepare` ao `ParticipantPortType` de cada um dos participantes que se registraram no protocolo *2PC*. Tais participantes devem então responder ao `CoordinatorPortType` do coordenador com uma destas mensagens:
   - `Prepared`: indica que o participante tornou duráveis, porém de modo não definitivo, quaisquer alterações de dados sob sua responsabilidade e que ele vota "sim" para a efetivação da transação distribuída.
   - `ReadOnly`: indica que o participante vota "sim" para a efetivação da transação distribuída e que ele não deseja participar da segunda etapa do *2PC*.
   - `Aborted`: indica que o participante abortou o seu ramo da transação e portanto vota "não" para a efetivação da transação distribuída.
2. Fase de efetivação: o coordenador faz a apuração dos votos. Se todos os participantes responderam com mensagens `Prepared` ou `ReadOnly`, então a transação será efetivada, caso contrário ela será abortada. No caso de uma decisão pela efetivação da transação, o coordenador envia uma mensagem `Commit` para o `ParticipantPortType` dos participantes que responderam com a mensagem `Prepared` na primeira fase do protocolo. Cada um desses participantes deve responder ao `CoordinatorPortType` do coordenador com uma mensagem `Committed`, indicando que o seu ramo da transação foi efetivado. Já no caso de uma decisão por abortar a transação, o coordenador envia uma mensagem `Rollback` para o `ParticipantPortType` dos participantes que responderam com a mensagem `Prepared` na primeira fase do protocolo. Tais participantes então respondem ao `CoordinatorPortType` do coordenador com a mensagem `Aborted`, indicando que eles abortaram os seus ramos da transação.

A Figura 3.16 mostra três cenários possíveis durante a execução do protocolo *2PC*.

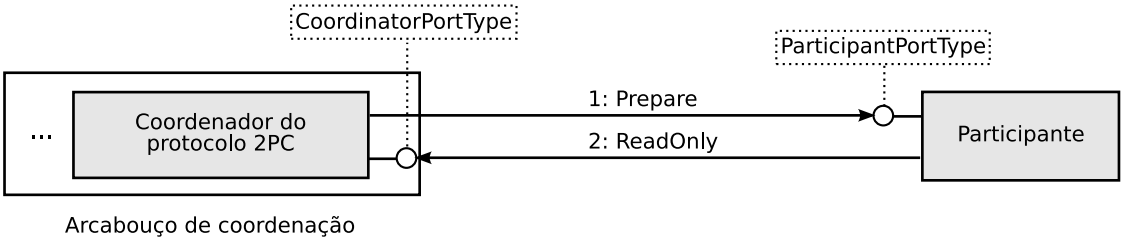
*WS-AtomicTransaction* define duas variações do 2PC. São elas:

- 2PC volátil: participantes cujos recursos transacionais são voláteis (por exemplo, um *cache*) devem se registrar nesse protocolo;
- 2PC durável: participantes cujos recursos transacionais são duráveis (por exemplo, um banco de dados ou uma fila de mensagens) devem se registrar nesse protocolo.

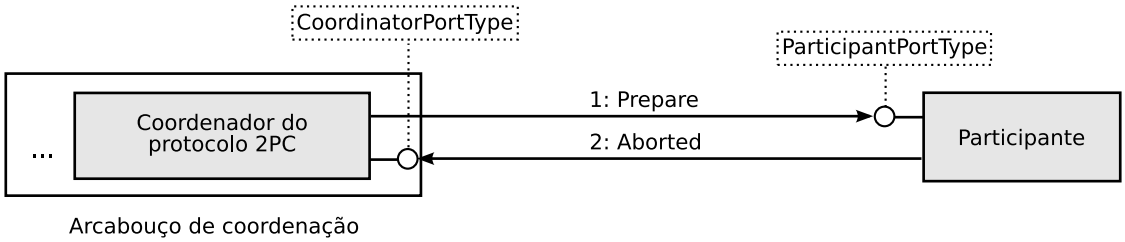
Há uma justificativa para a existência de duas variações do protocolo *2PC*. Alguns participantes de uma transação, por interagirem intensamente com algum meio de armazenamento persistente (por exemplo o disco rígido), podem impor uma sobrecarga considerável sobre o




(a) O participante responde `Prepared` na primeira fase; o coordenador envia uma mensagem `Commit` na segunda fase.




(b) O participante responde `ReadOnly` na primeira fase; não há segunda fase.




(c) O participante responde `Aborted` na primeira fase; como só há um participante não há segunda fase.

Figura 3.16: Três cenários possíveis durante a execução do protocolo *2PC*.

tempo de execução da transação. Uma solução natural para esse problema seria fazer um *cache* em memória dos dados presentes no meio de armazenamento persistente. Durante a execução da transação, os participantes trabalhariam apenas com os dados presentes nesse *cache*. Isso funciona muito bem, desde que os dados presentes no *cache* sejam transferidos para o meio de armazenamento persistente antes da transação iniciar o seu processo de encerramento. É por esse motivo que existe o *2PC* volátil: participantes que se registram nesse protocolo recebem a mensagem `Prepare` antes dos participantes que se registram no protocolo *2PC* durável. Isso dá aos participantes do *2PC* volátil a chance de transferir os dados de um meio volátil (um *cache*, por exemplo) para um meio persistente (um banco de dados, por exemplo), antes que qualquer notificação seja enviada aos participantes que gerenciam recursos duráveis.

Após enviar as mensagens `Prepare` a todos os participantes do 2PC volátil, o coordenador

da transação inicia a execução do protocolo *2PC* para os recursos que se registraram no *2PC* durável. Assim que o coordenador terminar de executar o *2PC* para todos os recursos duráveis, os participantes do *2PC* volátil serão informados sobre o desfecho da transação. Vale ressaltar, porém, que essas notificações aos participantes do *2PC* volátil são apenas uma cortesia, pois a transação já terá sido encerrada quando tais notificações forem enviadas (todos os recursos duráveis já terão recebido as mensagens do *2PC* antes de qualquer recurso volátil receber a notificação do desfecho da transação). Como as notificações aos participantes do *2PC* volátil são apenas uma cortesia, elas podem eventualmente não ocorrer (em caso de falhas, por exemplo).

### 3.3.3 Um exemplo de transação atômica envolvendo Web services

Tendo visto os serviços de ativação e registro de *WS-Coordination* e os protocolos *completion* e *2PC* de *WS-AtomicTransaction*, podemos agora mostrar a execução de uma transação atômica distribuída envolvendo *Web services*. A Figura 3.17 apresenta a execução de uma transação que envolve: um cliente (que inicia a transação), dois participantes do protocolo *2PC* (recursos transacionais acessíveis como *Web services*) e um coordenador transacional.

As mensagens presentes na Figura 3.17 são descritas a seguir:

1. O *Web service* cliente solicita ao coordenador a criação de um novo contexto transacional. Ele faz isso enviando uma mensagem `CreateCoordinationContext` ao `ActivationCoordinatorPortType` do coordenador.

2. O coordenador inicia uma nova transação, cria um contexto para a transação e o envia ao cliente através de uma mensagem `CreateCoordinationContextResponse`.

3. O cliente se registra com o coordenador no protocolo *completion*. Ele faz isso enviando uma mensagem `Register` ao `RegistrationCoordinatorPortType` do coordenador. Note que o cliente tem acesso ao endereço desse *port type* através do contexto transacional presente na mensagem `CreateCoordinationContextResponse` recebida anteriormente. O procedimento de registro é necessário para que o cliente seja notificado sobre o desfecho da transação.

4. O coordenador envia uma resposta ao cliente notificando que ele foi registrado no protocolo *completion*. A resposta enviada ao cliente inclui o endereço do `CompletionCoordinatorPortType` do coordenador, que o cliente utilizará futuramente (mensagem 13) para encerrar a transação.

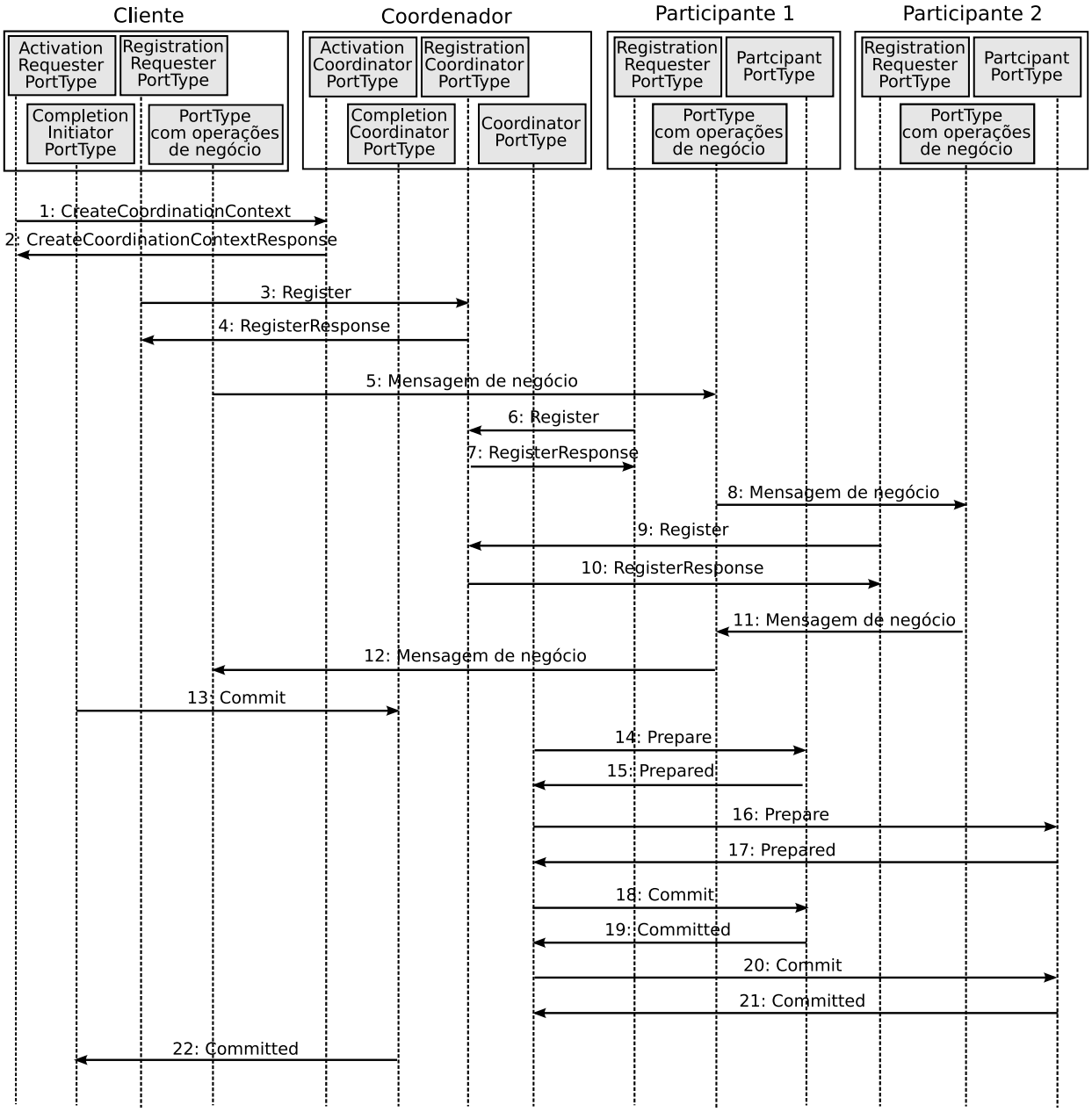



Figura 3.17: Mensagens trocadas durante a execução de uma transação atômica distribuída.

5. O cliente envia uma mensagem de negócio ao participante 1. Como essa mensagem é enviada dentro do escopo de uma transação, o contexto transacional é propagado juntamente com ela.

6. Durante a execução da operação de negócio, o participante 1 utiliza um recurso transacional (por exemplo, um banco de dados). Nesse momento o participante 1 envia uma mensagem `Register` ao `RegistrationCoordinatorPortType` do coordenador para se registrar no protocolo *2PC* durável (o endereço do `RegistrationCoordinatorPortType` do coordenador está presente no contexto transacional propagado juntamente com a requisição). É importante ressaltar que essa mensagem `Register` enviada ao coordenador contém o endereço do `ParticipantPortType` do participante 1.

7. O coordenador registra o participante 1 no protocolo *2PC* durável e o notifica desse fato. Juntamente com a resposta, o coordenador envia o endereço do seu `Coordinator-PortType`.

8. A operação de negócio do participante 1 chama uma operação de um segundo *Web service* (participante 2). Novamente o contexto transacional é propagado juntamente com a requisição.

9. Durante a execução da operação de negócio, o participante 2 utiliza um recurso transacional (por exemplo, um outro banco de dados). Nesse momento o participante 2 envia uma mensagem `Register` ao `RegistrationCoordinatorPortType` do coordenador para se registrar no protocolo *2PC* durável (o endereço do `RegistrationCoordinatorPortType` do coordenador está presente no contexto transacional propagado juntamente com a requisição). É importante ressaltar que essa mensagem `Register` enviada ao coordenador contém o endereço do `ParticipantPortType` do participante 2.

10. O coordenador registra o participante 2 no protocolo *2PC* durável e o notifica desse fato. Juntamente com a resposta, o coordenador envia o endereço do seu `Coordinator-PortType`.

11. O participante 2 envia a resposta da sua operação de negócio ao participante 1.

12. O participante 1 envia a resposta da sua operação de negócio ao cliente.

13. O cliente decide efetivar a transação e envia a mensagem `Commit` ao `Completion-CoordinatorPortType` do coordenador (o endereço desse *port type* foi obtido na mensagem 4).

14. O coordenador inicia a execução do protocolo *2PC* e envia uma mensagem `Prepare` ao `ParticipantPortType` do participante 1. O endereço desse *port type* foi fornecido ao coordenador quando o participante 1 se registrou no protocolo *2PC* durável (mensagem 6).

15. O participante 1 envia uma mensagem `Prepared` ao `CoordinatorPortType` do coordenador, indicando que ele está preparado e vota "sim" para a efetivação da transação. O endereço desse *port type* foi obtido pelo participante 1 quando ele se registrou no protocolo 2PC durável (mensagem 7).

16. O coordenador continua com a execução do *2PC* e envia uma mensagem `Prepare` ao `ParticipantPortType` do participante 2. O endereço desse *port type* foi fornecido ao coordenador quando o participante 2 se registrou no protocolo *2PC* durável (mensagem 9).

17. O participante 2 envia uma mensagem `Prepared` ao `CoordinatorPortType` do coordenador, indicando que ele está preparado e vota "sim" para a efetivação da transação. O endereço desse *port type* foi obtido pelo participante 2 quando ele se registrou no protocolo 2PC durável (mensagem 10).

18. O coordenador faz a apuração dos votos: os dois participantes votaram "sim" para a efetivação da transação. O coordenador passa então para a segunda fase do protocolo *2PC* e envia uma mensagem `Commit` ao `ParticipantPortType` do participante 1.

19. O participante 1 efetiva o seu ramo da transação e envia uma mensagem `Committed` ao `CoordinatorPortType` do coordenador.

20. O coordenador continua com a segunda fase do protocolo *2PC* e envia uma mensagem `Commit` ao `ParticipantPortType` do participante 2.

21. O participante 2 efetiva o seu ramo da transação e envia uma mensagem `Committed` ao `CoordinatorPortType` do coordenador.

22. O protocolo *2PC* foi completado com sucesso e a transação distribuída foi efetivada. O coordenador notifica o cliente desse fato enviando-lhe uma mensagem `Committed`.

É importante ressaltar que muitas dessas trocas de mensagens são efetuadas pelo middleware, ocorrendo transparentemente para o cliente. Este último tipicamente utiliza uma *API* de alto nível para a demarcação de transações, que disponibiliza operações tais como `begin`, `commit` e

`rollback`. Desse modo, o cliente geralmente informa apenas *(i)* o início da transação, *(ii)* quais requisições a outros *Web services* devem ser feitas como parte da transação e *(iii)* o fim da transação, salientando se ela deve ser efetivada ou abortada. Logo, as mensagens destinadas ao registro de recursos transacionais e à execução do protocolo *2PC* são geralmente enviadas e recebidas implicitamente pelo middleware. Além disso, o middleware também é o responsável pela propagação do contexto transacional.

## 3.4 WS-BusinessActivity

Há situações onde o modelo tradicional de transações atômicas não é adequado. É o caso, por exemplo, das transações de longa duração, cuja execução pode levar horas, dias ou até semanas. Para esse tipo de transação, muitas vezes é inviável manter travas (*locks*) sobre os recursos transacionais durante toda a execução da transação. Além disso, quando uma transação de longa duração é abortada, muito trabalho pode ser desfeito.

Outro cenário no qual o modelo tradicional de transações atômicas não é adequado é o das interações negócio-a-negócio. Nesse tipo de interação os recursos transacionais estão espalhados por várias empresas e, portanto, a obtenção de travas sobre recursos alheios pode ser (e geralmente é) inviável. Muito raramente uma empresa permitirá que outra empresa coloque travas arbitrariamente sobre seus recursos, seja por razões técnicas (segurança, por exemplo) ou não técnicas (violação de normas da empresa, por exemplo).

Para evitar os problemas enfrentados pelas transações atômicas nos cenários acima descritos, *WS-BusinessActivity* define um modelo transacional menos restritivo. Nesse modelo, as operações de uma transação são executadas e tornam-se duráveis imediatamente. Caso ocorra alguma falha durante a execução da transação, operações de compensação são acionadas para "compensar" a falha. Isso evita que todo o trabalho executado por uma transação de longa duração tenha que ser desfeito devido a ocorrência de alguma falha.

# 4 A implementação de um serviço de transações atômicas para Web services

Como parte do nosso estudo, desenvolvemos um serviço de transações atômicas voltado para *Web services* e compatível com as especificações *WS-Coordination* e *WS-AtomicTransaction*. Tal serviço foi integrado ao servidor de aplicações *JBoss*, estendendo a funcionalidade do gerenciador de transações desse servidor.

A decisão de utilizar o *JBoss* como ponto de partida para o nosso trabalho foi fundamentada nas seguintes razões:

- Trata-se de um servidor de aplicações de código aberto.
- Já possuíamos uma certa experiência com esse servidor de aplicações, o que auxiliaria o processo de desenvolvimento.
- O gerenciador de transações do *JBoss* possuía uma arquitetura extensível e bastante favorável ao desenvolvimento do nosso trabalho.
- O gerenciador de transações do *JBoss* não oferecia suporte a transações distribuídas envolvendo *Web services*. A adição desse recurso seria desejável e constituiria um significativo ganho de funcionalidade.

Como o nosso trabalho consiste numa extensão do gerenciador de transações presente no *JBoss*, convém apresentá-lo antes de fornecermos maiores detalhes sobre o nosso projeto.

## 4.1 O gerenciador de transações distribuídas do JBoss

Um gerenciador de transações é a entidade responsável por gerenciar e coordenar todo o trabalho envolvido numa transação. O *JBoss* possui um gerenciador de transações distribuídas denominado *Distributed Transaction Manager* (*DTM*), que contempla transações distribuídas envolvendo múltiplos servidores de aplicações e múltiplos gerenciadores de recursos (*resource*

*managers*)[1]. Ele é capaz de coordenar transações que envolvem dois tipos de recursos transacionais:

- Recursos XA: são recursos transacionais (bancos de dados, filas de mensagens, etc.) acessíveis ao *DTM* via adaptadores *JCA* [67] com suporte a *XA*[2] [75], como por exemplo *drivers JDBC* [73] e provedores *JMS* [64]. *Oracle*, *Sybase*, *IBM* e *BEA* são exemplos de empresas que fornecem gerenciadores de recursos transacionais e oferecem *drivers JDBC* ou provedores *JMS* com suporte a *XA* para seus produtos. Embora os recursos transacionais *XA* sejam vistos pelo *DTM* como objetos locais, eles não precisam executar no mesmo computador que o gerenciador de transações. Por exemplo, o *DTM* pode utilizar um *driver JDBC* para interagir com um sistema gerenciador de banco de dados remoto.

- Recursos remotos: são recursos transacionais acessíveis ao *DTM* através de algum mecanismo de chamada remota (por exemplo, *CORBA*). Esses recursos geralmente estão localizados em outras instâncias de servidores de aplicações.

O *DTM* é capaz de atuar como coordenador e conduzir a execução do protocolo *2PC* em transações distribuídas que envolvam esses dois tipos de recursos. Em particular, no caso dos recursos remotos, o *DTM* provê suporte a dois mecanismos de chamada remota:

- CORBA: as chamadas aos recursos remotos são feitas utilizando o *IIOP*. O suporte a esse mecanismo é importante, entre outros motivos, para permitir a execução de transações distribuídas envolvendo componentes *Enterprise JavaBeans* (*EJB*) [66] implantados em múltiplos servidores de aplicações. Isso porque a especificação *EJB*, com o intuito de promover a interoperabilidade, determina que a comunicação entre componentes *EJB* implantados em diferentes servidores de aplicações deve ocorrer através do *IIOP*.

- JBoss Remoting: as chamadas aos recursos remotos são feitas através do *JBoss Remoting*, um arcabouço para chamadas remotas específico do *JBoss*.

Vemos que, na realidade, o *DTM* comporta transações distribuídas envolvendo três tipos de recursos transacionais: recursos *XA*, recursos remotos acessíveis via *CORBA* e recursos remotos acessíveis via *JBoss Remoting* (Figura 4.1). Além disso, o *DTM* permite que uma

[1] Um gerenciador de recurso permite que uma aplicação tenha acesso a um recurso transacional, implementando uma interface que representa tal recurso. Um exemplo de gerenciador de recurso é um *driver JDBC*, que provê acesso a um banco de dados.

[2] A especificação *XA* define uma *API* para a interação com gerenciadores de recursos que oferecem suporte transacional. Os gerenciadores de recursos que implementam essa *API* são ditos compatíveis com a especificação *XA* (ou, em outras palavras, possuem suporte a *XA*).

transação distribuída envolva esses três tipos de recursos simultaneamente. Isso significa que o *DTM* pode coordenar, por exemplo, uma transação que envolve dois recursos *XA*, dois recursos remotos acessíveis via *CORBA* e um recurso remoto acessível via *JBoss Remoting*.

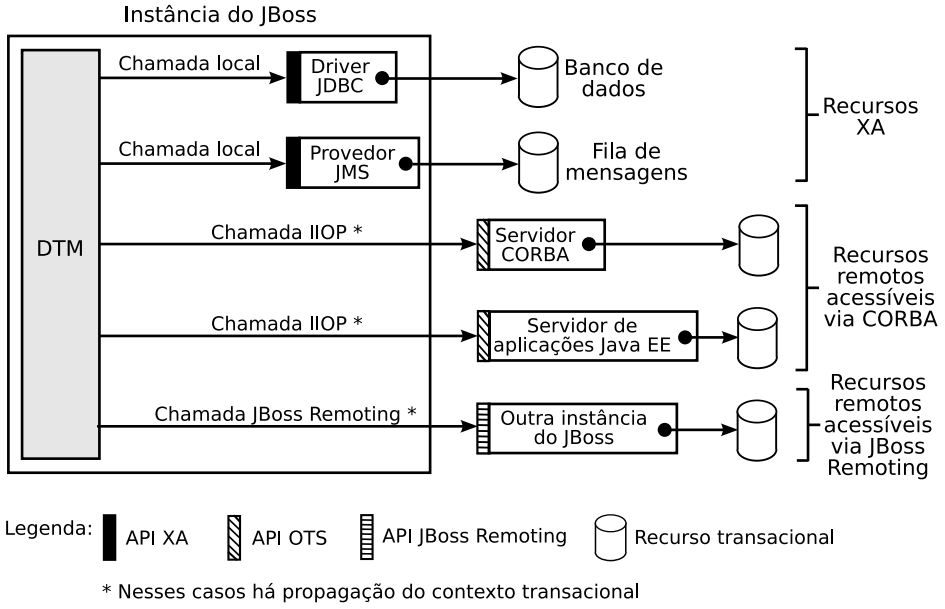



Figura 4.1: Os três tipos de recursos transacionais comportados pelo *DTM*.

A propagação do contexto transacional é feita automaticamente pelo *DTM* para todas as requisições *CORBA* ou *JBoss Remoting* que ocorram dentro do escopo de uma transação. No caso de uma requisição *CORBA*, o contexto transacional é propagado no formato definido pela especificação do *OTS*. Já no caso de uma requisição *JBoss Remoting*, o contexto transacional é propagado num formato específico do *JBoss*.

As requisições transacionais entre instâncias do servidor de aplicações *JBoss* podem ocorrer via *IIOP* ou via *JBoss Remoting*. Já as requisições transacionais entre o *JBoss* e algum outro servidor de aplicações devem ocorrer compulsoriamente via *IIOP*. Por exemplo, um *EJB* implantado numa instância do *JBoss* pode efetuar uma chamada transacional para outro *EJB* via *IIOP* ou *JBoss Remoting*, desde que esse segundo *EJB* também esteja implantado numa instância do *JBoss*. Se algum desses dois *EJBs* não estiver implantado numa instância do *JBoss*, a requisição transacional deverá ser feita obrigatoriamente via *IIOP*.

Durante a execução do protocolo *2PC*, o *DTM* interage com os recursos *XA* através da *API XA*. Já para os recursos remotos acessíveis via *CORBA*, a interação ocorre através das interfaces definidas pela especificação do *OTS*, apresentadas na Seção 2.1. Por fim, no caso

dos recursos remotos acessíveis via *JBoss Remoting*, a interação com o *DTM* ocorre através de interfaces que são específicas do *JBoss* e que foram definidas tendo como base as interfaces do *OTS* (Figura 4.1). Na realidade, conforme veremos adiante, internamente o *DTM* conduz a execução do *2PC* distinguindo apenas dois tipos de recursos: recursos *XA* e recursos remotos. A lógica para a interação com os recursos remotos através de um mecanismo de chamada remota específico fica confinada dentro de invólucros que implementam uma interface bem definida. Interagindo com esses invólucros, o *DTM* pode tratar uniformemente todos os recursos remotos.

Além de contemplar transações distribuídas envolvendo múltiplos tipos de recursos, o *DTM* também oferece suporte a recuperação de falhas. Para isso, o *DTM* faz uso de um *log* local, cujos registros são mantidos em meio de armazenamento persistente. Nesse *log* são registradas as ocorrências de eventos relevantes do *2PC* (um exemplo de evento relevante do *2PC* é a decisão de efetivar a transação). As informações presentes em tal *log* são utilizadas pelo procedimento de recuperação, que é executado após uma queda do coordenador ou do participante.

## 4.2 Visão geral da implementação do serviço de transações para Web services

A implementação do nosso serviço de transações atômicas para *Web services* pode ser dividida nas seguintes partes principais:

1. Extensão do *DTM*: nosso trabalho parte de um gerenciador de transações que comporta dois mecanismos de chamada remota (*CORBA* e *JBoss Remoting*) e o estende com o suporte a um terceiro mecanismo. Essa extensão permite que o *DTM* lide com recursos remotos acessíveis via *SOAP* e é discutida na Seção 4.3.

2. Implementação dos *port types* definidos em *WS-Coordination* e *WS-AtomicTransaction*: as implementações de tais *port types* viabilizam a comunicação entre o cliente, o *DTM* e os recursos remotos acessíveis como *Web services*. Maiores detalhes são apresentados na Seção 4.4.

3. Propagação e importação do contexto transacional: utilizamos interceptadores [59] para propagar e importar os contextos transacionais. Informações adicionais podem ser encontradas na Seção 4.5.

Juntas, essas partes permitem que o *DTM* gerencie também transações distribuídas envolvendo recursos remotos acessíveis como *Web services*. Embora não faça parte do serviço de

transações propriamente dito, uma parte considerável dos esforços de desenvolvimento foi concentrada na elaboração de testes. Elaboramos mais de uma centena de testes de unidade que verificam o correto funcionamento do *DTM* na coordenação de transações envolvendo recursos remotos acessíveis como *Web services*. Implementamos, inclusive, testes de unidade que envolvem múltiplos servidores de aplicações e que forçam a queda de um deles em momentos cruciais da execução do protocolo *2PC*. Os testes de unidade são discutidos na Seção 4.7.

## 4.3 A extensão do gerenciador de transações do JBoss

### 4.3.1 O suporte a recursos remotos acessíveis via SOAP

O *DTM* foi projetado tendo em mente que o suporte a novos mecanismos de chamada remota (por exemplo, *Web services/SOAP*) poderiam ser adicionados no futuro. Para viabilizar tal extensibilidade, o *DTM* interage com os recursos remotos apenas através de uma interface bem definida, denominada `org.jboss.tm.remoting.interfaces.Resource`. A comunicação do *DTM* com um recurso remoto ocorre através de classes que implementam essa interface e delegam as chamadas ao recurso através de algum mecanismo de chamada remota (Figura 4.2). Tais classes são apenas invólucros [27] que envolvem um recurso remoto e expõem ao *DTM* uma interface comum e independente do mecanismo de chamada remota. O uso de invólucros permite ao *DTM* tratar todos os recursos remotos uniformemente, mesmo quando múltiplos mecanismos de chamada remota são empregados numa mesma transação.

A Figura 4.2 mostra três invólucros: *JBoss Remoting*, *IIOP* e *SOAP*. Os dois primeiros fazem parte do *DTM*. O último, entretanto, é fruto do nosso projeto. As chamadas destinadas ao invólucro *SOAP* são delegadas ao `ParticipantPortType` de um *Web service* que representa o recurso transacional (Figura 4.3). Cabe lembrar que o `ParticipantPortType` é definido em *WS-AtomicTransaction* e possui três operações, denominadas `Prepare`, `Commit` e `Rollback`.

Nosso trabalho também adiciona ao *DTM* a capacidade de importar os contextos transacionais presentes nas requisições *SOAP* (a Seção 4.5 discute a implementação dessa funcionalidade). Quando o contexto de uma transação externa (isto é, uma transação distribuída iniciada em outra instância de servidor de aplicações) é importado pelo *DTM*, uma nova transação local é criada. Essa transação local é na realidade um ramo da transação distribuída (global) e, portanto, ela deve ser registrada com o coordenador da transação distribuída. Quando um ramo da transação se registra com o coordenador da transação distribuída, ele o faz assumindo o papel de um recurso remoto. Vale ressaltar, contudo, que esse processo de registro só será necessário quando algum recurso transacional (um recurso *XA*, por exemplo) for utilizado den-

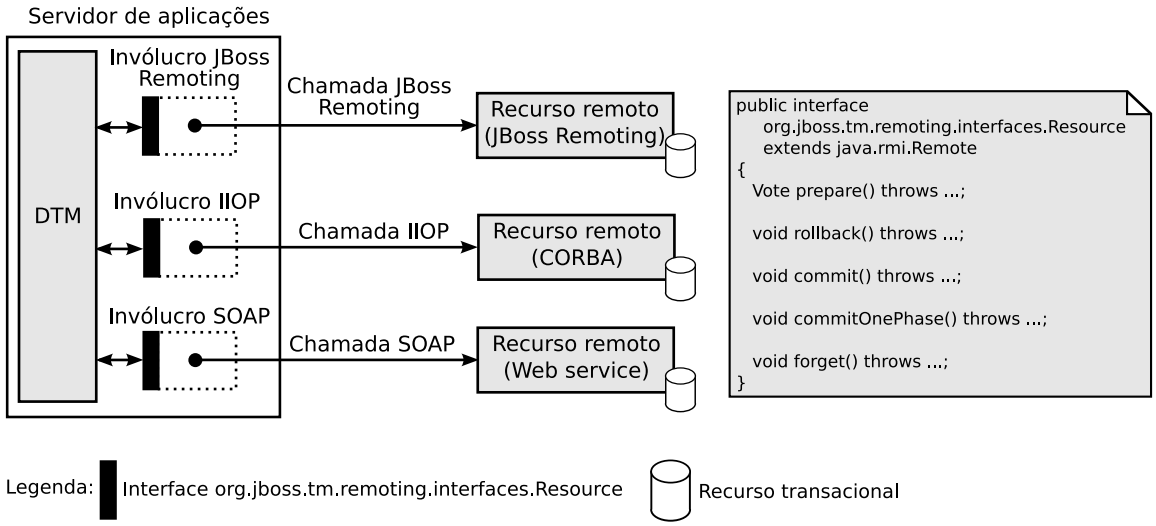



Figura 4.2: O *DTM* interage com os recursos remotos apenas através de invólucros que implementam a interface `org.jboss.tm.remoting.interfaces.Resource`. As chamadas a um invólucro são delegadas ao recurso remoto através de algum mecanismo de chamada remota.

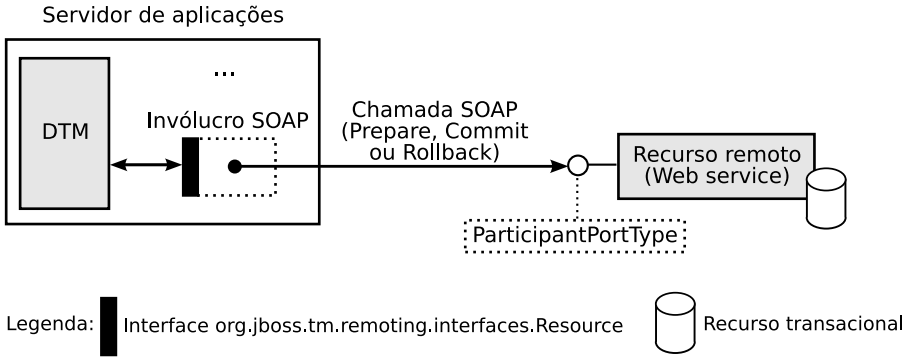



Figura 4.3: As chamadas feitas pelo *DTM* ao invólucro *SOAP* são delegadas ao `Participant-PortType` de um *Web service* que representa um recurso transacional.

tro do escopo da transação local. Caso isso não ocorra, o ramo da transação não precisa se registrar com o coordenador.

De modo análogo ao caso da interação com os recursos remotos, a comunicação do *DTM* com o coordenador da transação distribuída pode ocorrer através de vários protocolos de comunicação. Para lidar com essa situação, o *DTM* utiliza a mesma estratégia adotada na interação com os recursos remotos: ele interage com o coordenador da transação distribuída apenas através de uma interface bem definida, denominada `org.jboss.tm.remoting.interfaces.Coordinator`. A comunicação do *DTM* com o coordenador da transação distribuída ocorre somente através de classes que implementam essa interface e delegam as chamadas ao coordenador através de algum mecanismo de chamada remota (Figura 4.4). Tais classes são meros invólucros que envolvem o coordenador e expõem ao *DTM* uma interface comum e independente do mecanismo de chamada remota utilizado para acessar tal coordenador. O uso de invólucros permite ao *DTM* tratar todos os coordenadores de transações distribuídas de modo uniforme, independentemente do mecanismo de chamada remota utilizado para se comunicar com tais coordenadores.

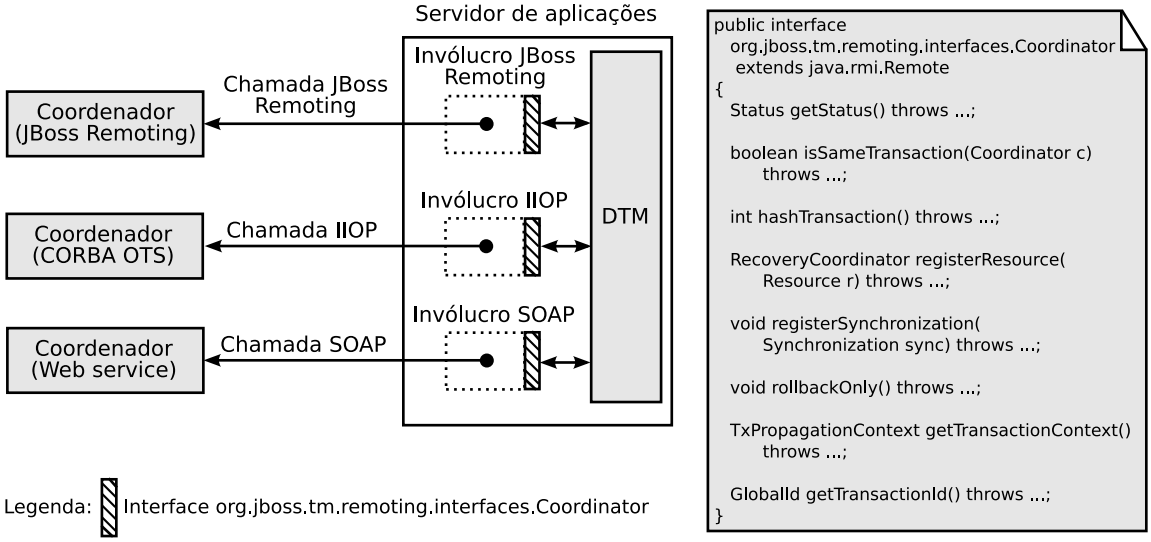



Figura 4.4: O *DTM* interage com o coordenador da transação distribuída apenas através de invólucros que implementam a interface `org.jboss.tm.remoting.interfaces.Coordinator`. Esses invólucros delegam as suas chamadas ao coordenador da transação distribuída através de algum mecanismo de chamada remota.

A Figura 4.4 exibe três invólucros: *JBoss Remoting*, *IIOP* e *SOAP*. Os dois primeiros fazem parte do *DTM*. O último, entretanto, é fruto do nosso projeto. As chamadas destinadas ao in-

vólucro *SOAP* são delegadas ao `RegistrationCoordinatorPortType` de um *Web service* que representa o coordenador da transação distribuída (Figura 4.5). Cabe lembrar que o `RegistrationCoordinatorPortType` é definido em *WS-Coordination* e possui uma única operação denominada `Register`.

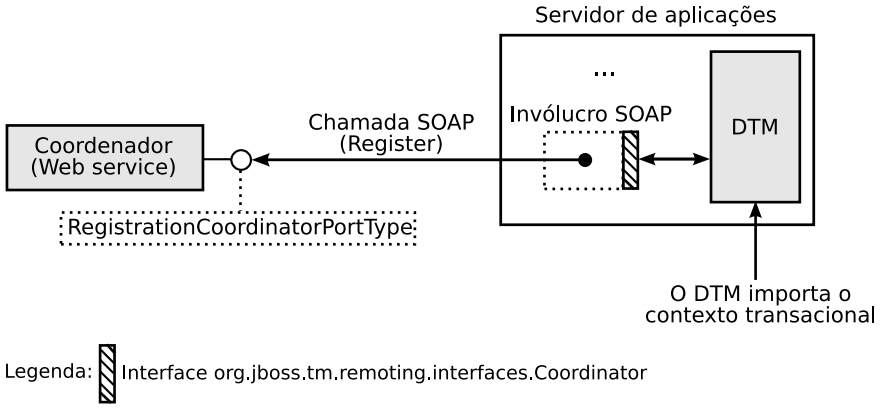



Figura 4.5: As chamadas feitas pelo *DTM* ao invólucro *SOAP* são delegadas ao `RegistrationCoordinatorPortType` de um *Web service* que representa o coordenador da transação distribuída.

Além das interfaces `Resource` e `Coordinator`, o pacote `org.jboss.tm.remoting.interfaces` possui mais duas importantes interfaces: `Synchronization` e `RecoveryCoordinator`. A primeira delas é utilizada para representar recursos remotos voláteis (por exemplo, um *cache*), enquanto que a segunda é utilizada no processo de recuperação de falhas. A mesma estratégia de invólucros utilizada para as interfaces `Resource` e `Coordinator` é empregada também para essas interfaces.

Além da implementação dos invólucros, a adição do suporte a recursos remotos acessíveis via *SOAP* exigiu algumas modificações na implementação do protocolo *2PC* existente no *DTM*. Uma dessas modificações foi a supressão da otimização de efetivação numa só fase quando um recurso remoto acessível como *Web service* se registrar com o coordenador. Tal medida foi necessária porque *WS-AtomicTransaction* não contempla a efetivação em uma só fase, exigindo, portanto, que a efetivação ocorra sempre em duas fases, mesmo quando há apenas um recurso transacional envolvido na transação. Foi necessário também modificar o gerenciador de transações para que ele informasse o desfecho da transação aos *Web services* que se registraram no protocolo *completion*. Com essa alteração, após a execução do protocolo *2PC* o gerenciador de transações envia mensagens `Commit` ou `Rollback` aos participantes do

protocolo *completion*[3], informando o desfecho da transação.

### 4.3.2 A implementação dos invólucros SOAP

Todos os invólucros descritos na seção anterior expõem ao *DTM* operações síncronas. Por outro lado, as operações definidas nos *port types* de *WS-Coordination* e *WS-AtomicTransaction* são assíncronas. Devido a essa discrepância, todos os invólucros *SOAP* implementam as suas operações síncronas empregando duas mensagens assíncronas (uma de requisição e uma de resposta).

A troca de mensagens assíncronas ocorre de maneira semelhante em todos os invólucros *SOAP*. Ao receber uma chamada local do *DTM*, o invólucro cria uma requisição *SOAP* assíncrona e nela injeta um identificador. A seguir, um *callback* é criado e associado a esse identificador. A requisição é então enviada ao *Web service* alvo e a linha de execução do invólucro fica bloqueada até que uma resposta chegue ao *callback*. O *Web service* alvo da requisição deve enviar uma resposta assíncrona que contenha o mesmo identificador presente na requisição. Tal resposta é recebida pelo servidor de aplicações que enviou a requisição através de um *servlet* [69], que extrai o identificador da resposta e a direciona ao *callback* correspondente. Após isso, a linha de execução do invólucro é liberada e a resposta da requisição é retornada ao *DTM* (Figura 4.6). Note que, do ponto de vista do *DTM*, tudo o que ocorre é uma chamada local.

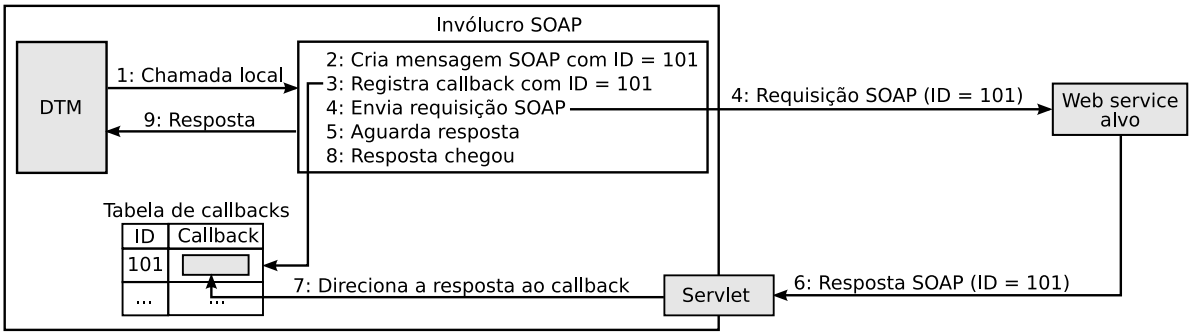



Figura 4.6: O procedimento executado quando a operação de um invólucro *SOAP* é chamada.

Para ilustrar como as mensagens *SOAP* assíncronas carregam o identificador de um *callback*, a Figura 4.7 mostra uma mensagem `Prepare` e a correspondente mensagem `Prepared`. Essas mensagens são trocadas entre o coordenador da transação e um recurso remoto acessível como

[3]Tipicamente há apenas um participante: o iniciador da transação.

um *Web service* quando o *DTM* do coordenador chama a operação `prepare` do invólucro *SOAP* que implementa a interface `org.jboss.tm.remoting.interfaces.Resource`.

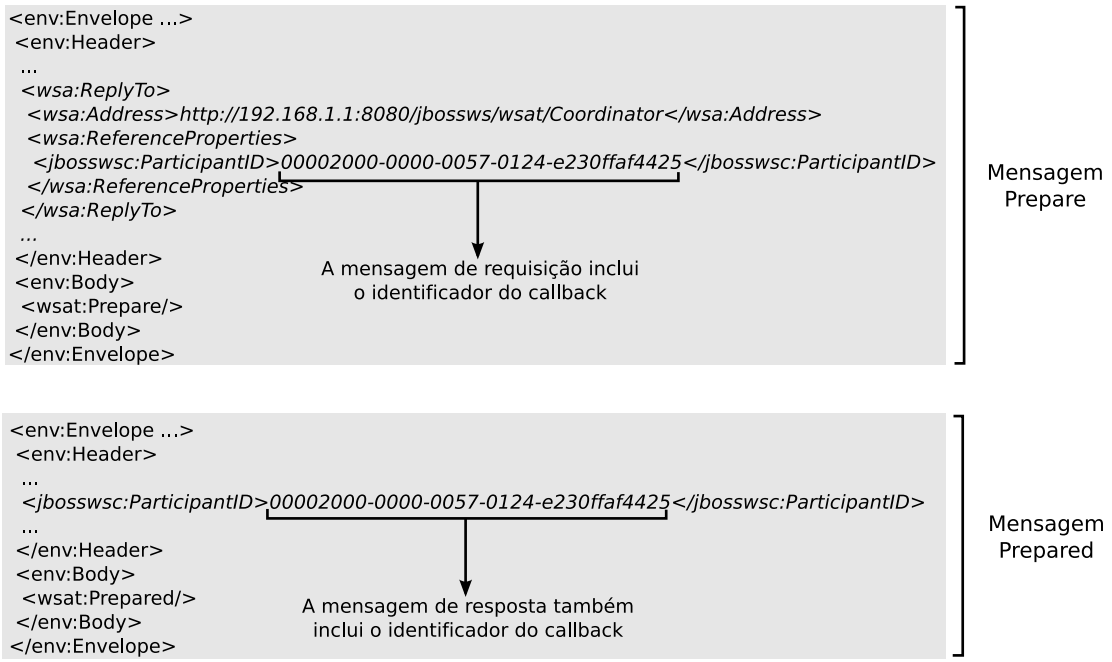



Figura 4.7: O identificador do *callback* está presente tanto nas mensagens de requisição quanto nas de resposta.

As respostas assíncronas são recebidas pelos mesmos *servlets* que implementam os *port types* de *WS-Coordination* e *WS-AtomicTransaction*. Tais *servlets* serão apresentados em detalhe na Seção 4.4. Porém, podemos discutir agora a capacidade que esses *servlets* têm de receber as respostas assíncronas e encaminhá-las para o *callback* apropriado.

Analisando uma mensagem *SOAP*, os *servlets* em questão conseguem distinguir se esta consiste ou não numa resposta a uma mensagem assíncrona enviada anteriormente. Por exemplo, as mensagens `Prepared`, `Committed` e `Aborted` são sempre respostas de requisições assíncronas e, portanto, devem ser encaminhadas a seus respectivos *callbacks*. Já as mensagens `Prepare`, `Commit` e `Rollback` nunca são respostas de requisições assíncronas e, como será mostrado na Seção 4.4, elas devem ser encaminhadas ao *DTM*. Sempre que esses *servlets* identificarem que uma mensagem é uma resposta a uma requisição assíncrona, tal resposta será direcionada ao *callback* apropriado.

Há ainda um último ponto importante sobre a implementação dos invólucros *SOAP*. Para atender aos requisitos das especificações *WS-Coordination* e *WS-AtomicTransaction*, as men-

sagens assíncronas enviadas pelos invólucros precisam incluir em seus cabeçalhos algumas informações de endereçamento padronizadas pela especificação *WS-Addressing* [13]. A seguir listamos um subconjunto das informações de endereçamento que podem estar presentes nas mensagens assíncronas geradas pelos invólucros *SOAP*:

- Identificador da mensagem (`MessageID`): uma *URI* que identifica unicamente a mensagem no espaço e no tempo.
- Destino (`To`): o endereço para o qual a mensagem deve ser encaminhada.
- Ação (`Action`): uma *URI* que identifica a semântica da mensagem.
- Endereço de resposta (`ReplyTo`): presente nas mensagens de requisição, esse campo especifica um endereço para o qual as respostas podem ser encaminhadas.
- Mensagem relacionada (`RelatesTo`): presente nas mensagens de resposta, esse campo relaciona uma mensagem de resposta com a sua respectiva mensagem de requisição.

Algumas dessas informações de endereçamento, como por exemplo o endereço de resposta, são imprescindíveis para a comunicação assíncrona. As demais informações, embora não sejam fundamentais, precisam ser incluídas para que as mensagens sejam compatíveis com a especificação. A Figura 4.8 mostra um exemplo de mensagem *SOAP* com informações de endereçamento[4].

### 4.3.3 A extensão do suporte a recuperação de falhas

Para viabilizar o suporte a recuperação de falhas, o *DTM* grava a ocorrência dos eventos relevantes do *2PC* nos *logs* locais do coordenador e dos participantes da transação. Tais *logs* são mantidos em meios de armazenamento persistentes e suas informações são utilizadas pelo procedimento de recuperação que o *DTM* executa após uma queda do coordenador ou do participante. Podemos dizer, portanto, que são as informações presentes nesses *logs* que possibilitam ao procedimento de recuperação recolocar o sistema num estado consistente.

Existem sete tipos de registros que podem ser gravados no *log* transacional pelo *DTM*. Cada um desses tipos representa a ocorrência de um evento relevante da execução do *2PC*. Ao invés de apresentar cada um dos tipos de registro, vamos nos ater aos dois mais importantes para o desenvolvimento deste trabalho.

[4] As mensagens *SOAP* geralmente contêm apenas um subconjunto das informações de endereçamento definidas em *WS-Addressing*. As mensagens de requisição, por exemplo, não incluem o elemento `RelatesTo` (mensagem relacionada), pois este só faz sentido em mensagens de resposta.

```
<env:Envelope>
 <env:Header>
  <wsa:MessageID>uuid:197020fa-a896-44f9-b28d-0ec827603ab3</wsa:MessageID>
  <wsa:To>http://192.168.1.3:8080/jbossws/wsat/Participant</wsa:To>
  <wsa:Action>http://schemas.xmlsoap.org/ws/2004/10/wsat/Prepare</wsa:Action>
  <wsa:ReplyTo>
   <wsa:Address>http://192.168.1.1:8080/jbossws/wsat/Coordinator</wsa:Address>
   <wsa:ReferenceProperties>
    <jbosswsc:ParticipantID>00000000-0000-0005-46ae-1eb300000001</jbosswsc:ParticipantID>
   </wsa:ReferenceProperties>
  </wsa:ReplyTo>
 </env:Header>
 <env:Body>
  ...
 </env:Body>
</env:Envelope>
```

Figura 4.8: Um exemplo de mensagem *SOAP* que contém um subconjunto das informações de endereçamento definidas em *WS-Addressing*.

#### 4.3.3.1 Os registros `MULTI_TM_TX_COMMITTED`

O primeiro tipo de registro de nosso interesse é o `MULTI_TM_TX_COMMITTED`. Ele é utilizado por transações criadas localmente pelo *DTM* que envolvam pelo menos um recurso remoto (ou seja, ele é utilizado por transações que envolvam mais de um gerenciador de transações, daí o prefixo `MULTI_TM`). Para transações que não envolvam recursos remotos, o *DTM* cria registros do tipo `TX_COMMITTED`, que são uma versão simplificada do `MULTI_TM_TX_COMMITTED`. Um registro do tipo `MULTI_TM_TX_COMMITTED` é gravado no *log* do coordenador da transação distribuída ao final da fase de votação do *2PC*, caso nenhum dos participantes tenha votado "não" para a efetivação da transação. A gravação bem sucedida desse registro no *log* do coordenador define que o desfecho da transação distribuída é a efetivação. Em outras palavras, a partir do momento em que o registro é escrito no *log* do coordenador, a transação distribuída pode ser considerada efetivada.

Um registro `MULTI_TM_TX_COMMITTED` contém, entre outras informações, referências para os recursos remotos que participam da transação. A presença dessas referências nesse tipo de registro é importante para garantir que, caso ocorra uma queda do coordenador, as referências para os recursos remotos envolvidos na transação estejam salvas num meio de armazenamento persistente. O procedimento de recuperação executado após a queda do coordenador pode obter as referências para os recursos remotos a partir do registro no *log* e dar prosseguimento à execução do protocolo *2PC*. Se as referências para os recursos remotos não fizessem parte do registro `MULTI_TM_TX_COMMITTED`, o procedimento de recuperação que executa no coordenador

não poderia prosseguir com a execução do *2PC*, pois tal procedimento desconheceria os recursos remotos envolvidos na transação.

Devido à diversidade de tipos de recursos remotos contemplados pelo *DTM*, é importante descrever como as referências para tais recursos são armazenadas nos registros do *log*. Cada referência para um recurso remoto é gravada como uma cadeia de caracteres (*string*), cujo formato varia de acordo com o mecanismo de chamada remota empregado para se comunicar com o recurso remoto. Por exemplo, no caso de um recurso remoto acessível via *JBoss Remoting*, uma referência para o recurso poderia ser representada pela cadeia de caracteres exibida na Figura 4.9. O formato da cadeia é o seguinte: um caractere que representa uma interface (o `R` indica que se trata de um `Resource`, ou seja, um recurso transacional remoto), seguido pela representação hexadecimal do identificador local da transação (um inteiro de precisão dupla), o qual é seguido por uma lista de *URIs*[5] separadas por barras verticais (`|`).

```
R3d00000004c75,socket://server4.acme.com:3873/
```

Figura 4.9: Uma cadeia de caracteres que representa um recurso remoto acessível via *JBoss Remoting*.

No caso dos recursos remotos acessíveis via *CORBA*, uma referência para um recurso remoto é armazenada no *log* como uma *Interoperable Object Reference* (*IOR*) [53]. Um exemplo de *IOR* é apresentado na Figura 4.10.

```
IOR:000000000000002B49444C3A6F6D672E6F72672F436F734E616D696E672F4E616D696E67436F6E74657
8744578743A312E3000000000000020000000000000E8000102000000000C3139322E3136382E312E31000D
C80000000000114A426F73732F4E616D696E672F726F6F740000000000000500000000000000008000000004
A414300000000010000001C0000000000010001000000010501000100010109000000010501000100000021
0000006000000000000000001000000000000000240000001E0000007E0000000000000001000000C3139322
E3136382E312E31000DC900400000000000000000000000100401000806066781020101010000000000000
0000000000000000000000000000000020000000400000000000001F0000000400000003000000010000
020000000000000000200000020000000040000000000000001F0000000400000003
```

Figura 4.10: Um exemplo de *IOR* que representa um recurso remoto acessível via *CORBA*.

No caso dos recursos remotos acessíveis como *Web services*, uma referência para um recurso remoto é armazenada no *log* na forma de uma *endpoint reference*[6]. Apenas essa informação, contudo, não é suficiente para que o coordenador restabeleça contato com o recurso remoto

[5] Na realidade, o arcabouço *JBoss Remoting* utiliza o conceito de `InvokerLocator` para identificar um serviço remoto. Embora um `InvokerLocator` não seja estritamente compatível com a definição de uma *URI*, essas duas entidades são muito semelhantes. Logo, dentro do escopo desta dissertação, podemos considerar os termos `InvokerLocator` e *URI* como intercambiáveis.

[6] Uma *endpoint reference* é, na tecnologia *Web services*, o equivalente a uma *IOR* de *CORBA*.

após uma queda. Dado que a comunicação com o recurso remoto acessível como *Web service* ocorre de modo assíncrono, o coordenador deve incluir um elemento `ReplyTo` (Figura 4.11) no cabeçalho das mensagens *SOAP* destinadas ao recurso remoto. Dentro desse elemento, o coordenador envia o identificador do recurso remoto (`ParticipantID`), que é um *Universally Unique Identifier* (*UUID*) [43] gerado quando o recurso foi registrado com o coordenador. O recurso remoto inclui esse identificador nas suas mensagens de resposta, permitindo ao coordenador identificar qual o recurso remoto que enviou uma determinada resposta. Como o coordenador precisa do *UUID* de um recurso remoto para criar o elemento de endereçamento `ReplyTo`, o *UUID* também é armazenado no *log*.

```
<wsa:ReplyTo>
 <wsa:Address>http://192.168.1.1:8080/jbossws/wsat/Coordinator</wsa:Address>
 <wsa:ReferenceProperties>
  <jbosswsc:ParticipantID>00000000-0000-0005-46a8-040f00000001</jbosswsc:ParticipantID>
 </wsa:ReferenceProperties>
</wsa:ReplyTo>
```

Figura 4.11: Um elemento `ReplyTo` que referencia o coordenador da transação.

Para cada recurso remoto acessível como um *Web service* envolvido na transação, uma *endpoint reference* e um *UUID* deveriam ser gravados no *log*. Cada *UUID*, contudo, é composto por dois inteiros longos: o identificador local da transação e um identificador do recurso remoto dentro do escopo da transação. O primeiro desses inteiros é o mesmo para todos os *UUIDs* dos recursos envolvidos na mesma transação. Logo, ao invés de armazenar um *UUID* completo, apenas o segundo inteiro longo é armazenado juntamente com a *endpoint reference* de um recurso remoto. O identificador local da transação é salvo apenas uma vez no registro do *log*. Desse modo, para cada recurso remoto acessível como um *Web service* envolvido na transação, uma cadeia de caracteres semelhante à da Figura 4.12[7] é gravada no *log*. Note que todas as linhas da figura, exceto a última, correspondem à *endpoint reference*. A última linha, por sua vez, contém o segundo inteiro longo do *UUID* (em representação hexadecimal).

Como é possível observar na Figura 4.12, uma *endpoint reference* consiste num excerto *XML* e, conforme o seu próprio nome sugere, ela é uma espécie de referência para um *Web service*. Em outras palavras, uma *endpoint reference* agrega as informações necessárias para que um cliente interaja com o *Web service*. Ela contém, por exemplo, o endereço do *Web service* (elemento `Address`) e um conjunto de dados que devem ser enviados juntamente com as requisições feitas ao *Web service* (elemento `ReferenceProperties`). Assim sendo, um

[7] As quebras de linha e a indentação foram introduzidas na Figura 4.12 somente para melhorar a legibilidade.

```
<wscoor:ParticipantProtocolService
 xmlns:wsa='http://schemas.xmlsoap.org/ws/2004/08/addressing'
 xmlns:wscoor='http://schemas.xmlsoap.org/ws/2004/10/wscoor'
 xmlns:jbosswsc='http://www.jboss.org/wscoor/extension'>
 <wsa:Address>http://192.168.1.3:8080/jbossws/wsat/Participant</wsa:Address>
 <wsa:ReferenceProperties>
  <jbosswsc:CoordinatedActivityID>3</jbosswsc:CoordinatedActivityID>
 </wsa:ReferenceProperties>
</wscoor:ParticipantProtocolService>
46a8040f00000001
```

Figura 4.12: Para cada recurso remoto acessível como um *Web service* envolvido em uma transação, uma cadeia de caracteres semelhante a esta é armazenada no registro `MULTI_TM_TX_COMMITTED`.

cliente que utiliza a *endpoint reference* presente na Figura 4.12 irá incluir nas requisições feitas ao *Web service* o elemento `CoordinatedActivityID` com o valor `3`. Maiores detalhes sobre as *endpoint references* podem ser vistos na especificação *WS-Addressing* [13].

#### 4.3.3.2 Os registros `TX_PREPARED`

O segundo tipo de registro de nosso interesse é o `TX_PREPARED`. Ele é utilizado nas transações externas que ingressam na máquina virtual do *DTM* através de contextos transacionais propagados juntamente com as chamadas remotas. Nessa situação, o *DTM* participante, que foi envolvido nessa transação iniciada externamente, atua como subcoordenador da transação. Um registro do tipo `TX_PREPARED` é gravado no *log* de um participante da transação quando ele tiver tornado duráveis, porém de modo não definitivo, todas as alterações de dados sob sua responsabilidade. Isso irá ocorrer quando o participante, atuando como subcoordenador da transação, encerrar a fase de votação do *2PC* para todos os recursos (*XA* ou remotos) sob sua responsabilidade. Cabe ressaltar, entretanto, que caso algum desses recursos vote "não" para a efetivação da transação, não haverá a criação de um registro `TX_PREPARED` no *log* do participante e a transação distribuída será abortada.

Um registro `TX_PREPARED` contém, entre outras informações, uma referência para um `RecoveryCoordinator`. A presença dessa referência num registro `TX_PREPARED` é importante pois, caso ocorra uma queda do participante, o procedimento de recuperação do participante pode utilizar essa referência para perguntar ao `RecoveryCoordinator` qual foi o desfecho da transação distribuída. O participante faz isso enviando uma mensagem `Replay` ao `RecoveryCoordinator`, que deve então reenviar ao participante a última mensagem apropriada do protocolo *2PC* (`Commit` ou `Rollback`). É importante ressaltar dois pontos sobre a operação `Replay`:

1. Ela só é utilizada quando ocorre algum contratempo na execução da transação (uma queda ou o *timeout* de um dos participantes, por exemplo).

2. Após o registro de uma entrada `TX_PREPARED` no *log* do participante, caso este sofra algum tipo de falha, é dele a responsabilidade de contactar o `RecoveryCoordinator` (através da execução da operação `Replay` em seu procedimento de recuperação) para averiguar qual foi o desfecho da transação.

O formato da cadeia de caracteres utilizado para representar um `RecoveryCoordinator` é bastante similar ao utilizado para representar recursos remotos. Por exemplo, uma referência para um `RecoveryCoordinator` acessível via *JBoss Remoting* pode ser representada pela cadeia mostrada na Figura 4.13. Vale ressaltar que a primeira letra da cadeia é um `V`, indicando se tratar de uma referência para um `RecoveryCoordinator`. O restante da cadeia segue o mesmo formato da cadeia apresentada na Figura 4.9. No caso de um `RecoveryCoordinator` acessível via *CORBA*, a referência é representada nos registros do *log* como uma *IOR*. Por fim, no caso de um `RecoveryCoordinator` acessível como um *Web service*, a referência é representada nos registros do *log* por uma *endpoint reference*, como mostra a Figura 4.14.

```
V1b6,socket://zee.acme.com:3873/|rmi://zee.acme.com:5678/
```

Figura 4.13: Uma cadeia de caracteres que representa um `RecoveryCoordinator` acessível via *JBoss Remoting*.

```
<wscoor:CoordinatorProtocolService
 xmlns:wsa='http://schemas.xmlsoap.org/ws/2004/08/addressing'
 xmlns:wscoor='http://schemas.xmlsoap.org/ws/2004/10/wscoor'
 xmlns:jbosswsc='http://www.jboss.org/wscoor/extension'>
 <wsa:Address>http://192.168.1.1:8080/jbossws/wsat/Coordinator</wsa:Address>
 <wsa:ReferenceProperties>
  <jbosswsc:ParticipantID>00000000-0000-0005-46a8-040f00000001</jbosswsc:ParticipantID>
 </wsa:ReferenceProperties>
</wscoor:CoordinatorProtocolService>
```

Figura 4.14: Uma cadeia de caracteres que representa um `RecoveryCoordinator` acessível como um *Web service*.

Um registro `TX_PREPARED` pode conter também referências para recursos remotos. Isso ocorre quando há interposição de coordenadores e um participante atua como o subcoordenador de um ramo da transação que engloba recursos remotos. Tal cenário é interessante quando o coordenador se comunica com os participantes da transação através de um enlace de baixa velocidade. Nesse caso, alguns participantes podem se registrar com um subcoordenador acessível

através de um enlace mais rápido, sendo que apenas este último se registra com o coordenador através do enlace de baixa velocidade (Figura 4.15). O registro `TX_PREPARED` armazenado num subcoordenador utiliza o mesmo formato de cadeia de caracteres para recursos remotos definido anteriormente. A presença dessas referências no registro `TX_PREPARED` é importante para garantir que o subcoordenador, na eventualidade de uma queda, seja capaz de restabelecer a comunicação com os recursos remotos que ele coordena.

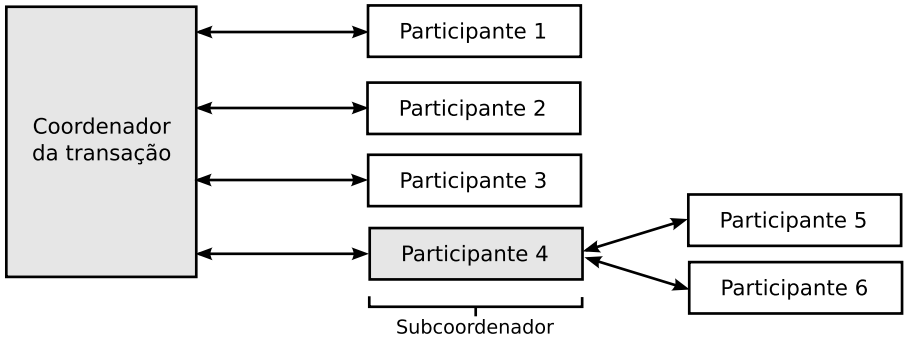



Figura 4.15: Interposição de coordenadores. O quarto participante atua como um subcoordenador da transação.

Para ilustrar o uso dos registros `MULTI_TM_TX_COMMITTED` e `TX_PREPARED`, tomemos como exemplo uma transação distribuída que envolve três recursos remotos: um acessível via *JBoss Remoting*, outro via *CORBA/IIOP* e o último via *Web services/SOAP* (Figura 4.16). Na primeira fase do *2PC*, o coordenador envia mensagens *Prepare* a todos os participantes. Cada participante torna então duráveis, mas de modo não definitivo, todas as alterações de dados sob sua responsabilidade. Após isso, cada participante armazena em seu *log* local um registro `TX_PREPARED` e, em seguida, envia uma mensagem *Prepared* ao coordenador. Concluída a apuração dos votos, o coordenador verifica que todos os participantes votaram "sim" para a efetivação da transação e escreve em seu *log* local uma entrada `MULTI_TM_TX_COMMITTED`. Dá-se início então à segunda fase do *2PC*, que não é mostrada na Figura 4.16.

Para finalizar a discussão sobre o suporte a recuperação de falhas no *DTM*, vamos descrever como ele converte referências para entidades remotas em cadeias de caracteres e vice-versa. A interface `org.jboss.tm.StringRemoteRefConverter`, exibida na Figura 4.17, define as operações que devem ser implementadas por um conversor bidirecional de referências em cadeias de caracteres. Para cada mecanismo de chamada remota contemplado pelo *DTM*, existe uma classe que implementa essa interface. O gerenciador de transações delega a essas classes a tarefa de efetuar a conversão de referências remotas em cadeias de caracteres e vice-versa.

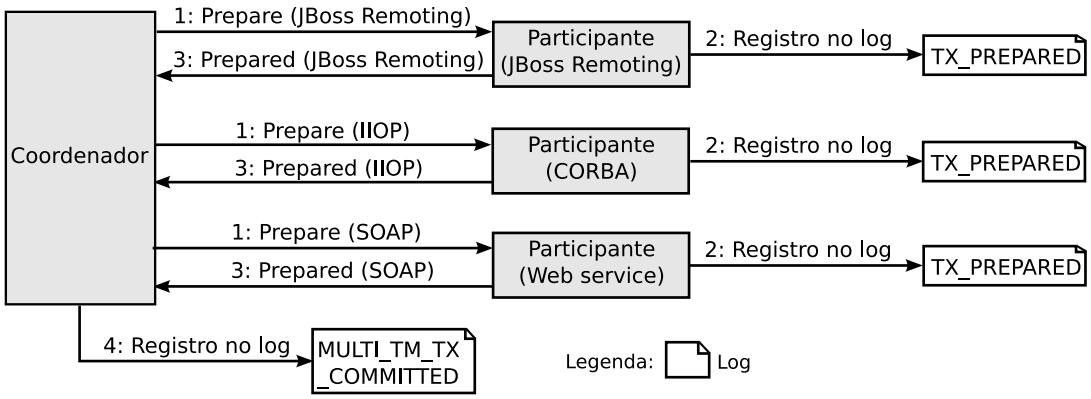



Figura 4.16: O uso dos registros `MULTI_TM_TX_COMMITTED` e `TX_PREPARED` no decorrer de uma transação distribuída.

```
public interface StringRemoteRefConverter
{
   Resource stringToResource(String strResource);
   RecoveryCoordinator stringToRecoveryCoordinator(String strRecCoordinator);
   String resourceToString(Resource res);
   String recoveryCoordinatorToString(RecoveryCoordinator recoveryCoord);
}
```

Figura 4.17: A interface `org.jboss.tm.StringRemoteRefConverter`.

## 4.4 A implementação dos port types

As especificações *WS-Coordination* e *WS-AtomicTransaction* definem uma série de *port types* que devem ser implementados para possibilitar a comunicação entre os membros de uma transação distribuída. *WS-Coordination* define os *port types* `ActivationCoordinatorPortType` e `RegistrationCoordinatorPortType`, os quais permitem, respectivamente, a criação de novos contextos de coordenação e o registro de participantes numa atividade coordenada (que pode ser, por exemplo, uma transação). Já *WS-AtomicTransaction* define os *port types* `Completion-CoordinatorPortType`, `CoordinatorPortType` e `ParticipantPortType`, que permitem, respectivamente, iniciar o processo de encerramento de uma transação e a troca de mensagens do protocolo *2PC* entre o coordenador e os participantes. Todos os *port types* mencionados recebem requisições *SOAP* e as delegam ao *DTM* (Figura 4.18). Eles atuam, portanto, como uma espécie de adaptador [27] que converte requisições *SOAP* em chamadas locais ao *DTM*.

Nossa implementação inicial dos *port types* de *WS-Coordination* e *WS-AtomicTransaction* foi baseada na *Java API for XML-based RPC* (*JAX-RPC*) [68]. Embora essa implementação inicial funcionasse corretamente, ela não era eficiente, pois o caminho que uma requisição

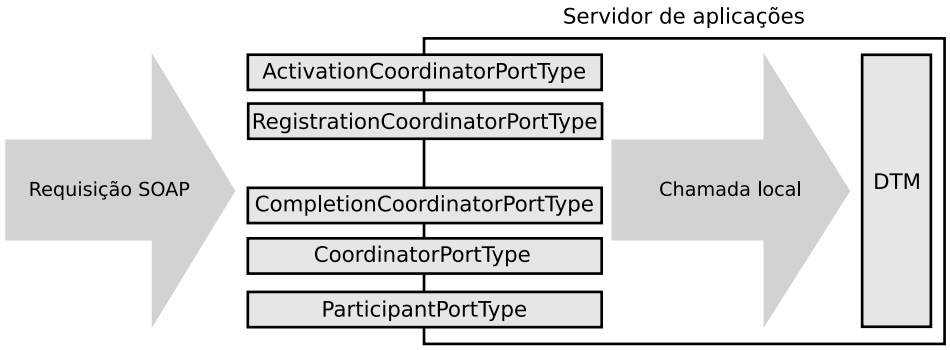



Figura 4.18: Os *port types* de *WS-Coordination* e *WS-AtomicTransaction* delegam as requisições *SOAP* ao *DTM*.

*SOAP* percorre num ambiente de execução *JAX-RPC* é bastante longo e envolve operações onerosas.

Na primeira implementação dos *port types*, a requisição *SOAP* era recebida por um *servlet* que faz parte do ambiente de execução *JAX-RPC* e desempenha o papel de tratador genérico de requisições *SOAP*. Em seguida, a requisição era convertida para um conjunto de objetos definidos na *SOAP with Attachments API for Java* (*SAAJ*)[8] [70]. Após isso, a requisição *SOAP* atravessava uma cadeia de interceptadores, sendo que estes podiam manipulá-la também através da *SAAJ*. A seguir, a requisição era despachada, via reflexão Java, para a implementação do *Web service*. Por fim, essa implementação delegava a requisição ao *DTM* (Figura 4.19).

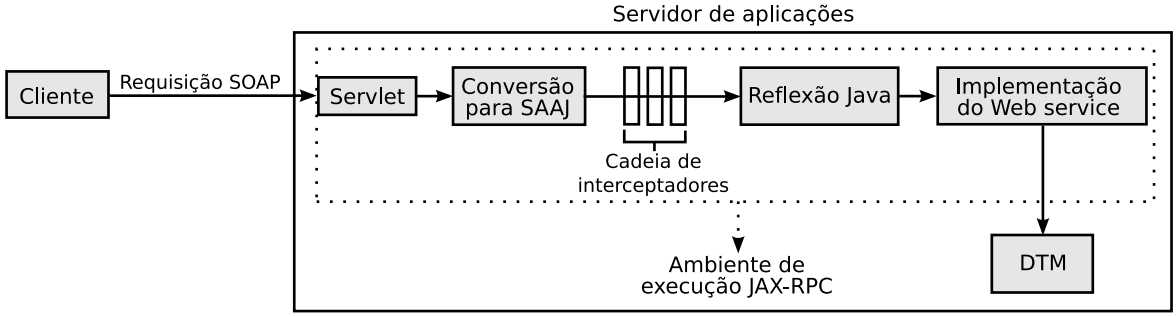



Figura 4.19: O trajeto percorrido por uma requisição *SOAP* na nossa implementação inicial dos *port types* de *WS-Coordination* e *WS-AtomicTransaction*

No trajeto apresentado na Figura 4.19, há duas etapas bastante onerosas: *(i)* a conversão da requisição *SOAP* para elementos da *SAAJ* e *(ii)* o uso de reflexão Java para determinar

[8] Essa *API* consiste numa extensão do *Document Object Model* (*DOM*) [6], sendo voltada para a manipulação de mensagens *SOAP*.

qual operação da implementação do *Web service* deve ser chamada.

Visando maior eficiência, os *port types* de *WS-Coordination* e *WS-AtomicTransaction* foram reimplementados como dois *servlets* independentes do ambiente de execução *JAX-RPC*. O *servlet* `org.jboss.ws.wstx.coordination.WSCoorServletEndpoint` trata todas as requisições destinadas aos *port types* definidos em *WS-Coordination*. Similarmente, o *servlet* `org.jboss.ws.wstx.atomictx.WSATServletEndpoint` trata todas as requisições destinadas aos *port types* definidos em *WS-AtomicTransaction*. Esses dois *servlets* delegam as requisições *SOAP* quase que diretamente ao *DTM* (Figura 4.20), evitando assim boa parte do processamento que ocorreria num ambiente *JAX-RPC*. Além disso, os *servlets* em questão fazem a manipulação das mensagens *SOAP* através de uma *API* muito mais eficiente do que a *SAAJ*: a *Streaming API for XML* (*StAX*) [8]. Todos esses fatores contribuíram para um ganho de desempenho de aproximadamente uma ordem de magnitude no tratamento das requisições *SOAP*.

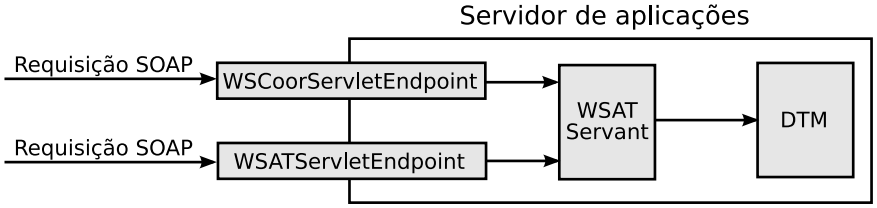



Figura 4.20: Os *port types* de *WS-Coordination* e *WS-AtomicTransaction* foram reimplementados e passaram a delegar as requisições *SOAP* quase que diretamente ao *DTM*.

As requisições destinadas aos *servlets* `WSCoorServletEndpoint` e `WSATServletEndpoint` percorrem caminhos ligeiramente diferentes até chegarem ao *DTM*. Veremos a seguir maiores detalhes sobre esses caminhos.

### 4.4.1 Os port types de WS-Coordination

Como *WS-Coordination* define um coordenador genérico, a especificação não descreve quais ações devem ser executadas pelas operações dos seus *port types*. Isso é ditado pelos protocolos de coordenação, que estendem a funcionalidade do coordenador. Para refletir tal fato, a nossa implementação de *WS-Coordination* consiste numa infra-estrutura extensível onde tipos de coordenação[9] podem ser adicionados ou removidos em tempo de execução.

O *servlet* `WSCoorServletEndpoint` delega as requisições a um repositório de implementações de tipos de coordenação. Esse repositório localiza a implementação de tipo de coordenação

[9]Como vimos na Seção 3.2.1, um tipo de coordenação consiste num conjunto de protocolos de coordenação logicamente relacionados.

apropriada para tratar a requisição. No caso do tipo de coordenação "transação atômica", quem trata as requisições é uma instância da classe `org.jboss.ws.wstx.atomictx.WSATCoordinationType`. A única função dessa classe é delegar as chamadas à classe `org.jboss.ws.wstx.atomictx.WSATServant`, que por sua vez as repassa ao *DTM* (Figura 4.21).

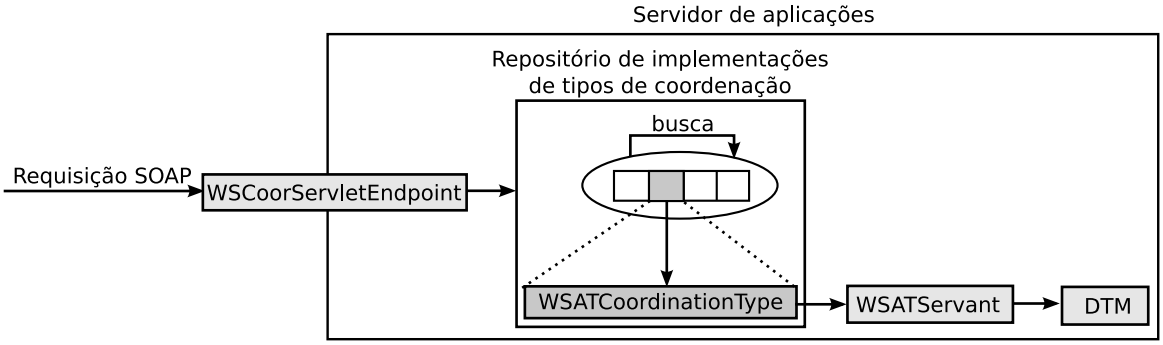



Figura 4.21: O caminho percorrido por uma requisição *SOAP* enviada a um dos *port types* de *WS-Coordination*.

O repositório de tipos de coordenação é implementado pela classe `org.jboss.ws.wstx.coordination.PluggableCoordinationTypeRepository`, apresentada na Figura 4.22. Como é possível ver, a classe possui quatro métodos públicos. Os dois primeiros deles servem, respectivamente, para adicionar e remover implementações de tipos de coordenação. Já os dois últimos métodos possuem nomes idênticos aos das operações definidas nos *port types* `ActivationCoordinatorPortType` e `RegistrationCoordinatorPortType` de *WS-Coordination*. Quando o `WSCoorServletEndpoint` recebe uma mensagem `CreateCoordinationContext` ou `Register`, ele chama o método correspondente do repositório. Os dois últimos métodos do repositório funcionam de modo semelhante: eles procuram no repositório um de tipo de coordenação apropriado para tratar a requisição e delegam tal requisição.

```
public class PluggableCoordinationTypeRepository
{
   public void addCoordinationType(PluggableCoordinationType handler) { ... }
   public void removeCoordinationType(PluggableCoordinationType handler) { ... }
   public CreateCoordinationContextResponse createCoordinationContext(
          CreateCoordinationContext create) { ... }
   public RegisterResponse register(String coordinatedActivityID,
          Register register) { ... }
}
```

Figura 4.22: A classe `org.jboss.ws.wstx.coordination.PluggableCoordinationTypeRepository`.

Para localizar o tipo de coordenação correto para tratar uma requisição *SOAP*, a classe `PluggableCoordinationTypeRepository` utiliza os dados presentes na própria requisição. Por exemplo, uma mensagem `CreateCoordinationContext` destinada ao `ActivationCoordinatorPortType` deve conter um tipo de coordenação (representado por uma *URI*). Já uma mensagem `Register` destinada ao `RegistrationCoordinatorPortType` deve conter um protocolo de coordenação (também representado por uma *URI*), a partir do qual é possível determinar o tipo de coordenação. Portanto, a própria requisição contém os dados que possibilitam ao `PluggableCoordinationTypeRepository` selecionar a implementação de tipo de coordenação apropriada para tratá-la.

Uma implementação de um tipo de coordenação consiste numa classe que implementa a interface exibida na Figura 4.23. As duas primeiras operações dessa interface permitem ao repositório indagar qual é tipo e os protocolos de coordenação que uma classe implementa. São essas informações que o repositório utiliza para descobrir a qual classe ele deve delegar uma requisição recebida pelo `WSCoorServletEndpoint`. Já as duas últimas operações possuem exatamente as mesmas assinaturas dos dois últimos métodos do repositório. Tal fato torna trivial a delegação de requisições por parte do repositório quando um tipo de coordenação apropriado é encontrado. Desse modo, para estender a funcionalidade de nossa implementação de *WS-Coordination*, basta registrar com o repositório, em tempo de execução, instâncias de classes que implementam a interface `PluggableCoordinationType`.

```
interface org.jboss.ws.wstx.coordination.PluggableCoordinationType
{
   URI getURI();
   boolean supportsCoordinationProtocol(URI coordinationProtocol);
   CreateCoordinationContextResponse createCoordinationContext(
                                     CreateCoordinationContext create);
   RegisterResponse register(String coordinatedActivityID, Register register);
}
```

Figura 4.23: A interface `org.jboss.ws.wstx.coordination.PluggableCoordinationType`.

### 4.4.2 Os port types de WS-AtomicTransaction

Vimos que *WS-AtomicTransaction* define os *port types* `CompletionCoordinatorPortType`, `CoordinatorPortType` e `ParticipantPortType`, os quais permitem o encerramento de uma transação e a troca de mensagens do protocolo *2PC* entre o coordenador e os participantes. Todos esses *port types* foram implementados através do *servlet* `WSATServletEndpoint`.

Diferentemente do *servlet* que trata as requisições de *WS-Coordination*, o *servlet* `WSATServ-`

`letEndpoint` não precisa delegar as requisições para um repositório que seleciona dinamicamente qual classe vai tratar determinada requisição. Ao invés disso, ele delega as requisições diretamente ao `WSATServant`, como mostra a Figura 4.20.

## 4.5 A propagação e importação do contexto transacional

A propagação e a importação do contexto transacional são efetuadas por intermédio de interceptadores. Do lado cliente, um interceptador acoplado ao representante (*proxy*) do *Web service* é o responsável por injetar o contexto transacional nas requisições *SOAP* que ocorrem dentro do escopo de uma transação. Do lado servidor, há um interceptador que trata as requisições *SOAP* recebidas pelo *Web service*. Esse interceptador verifica se existe um contexto transacional na requisição *SOAP* e, em caso afirmativo, ele extrai e repassa esse contexto ao *DTM*.

Nós desenvolvemos dois conjuntos de interceptadores. O primeiro conjunto é compatível com a especificação *JAX-RPC* e é empregado por *Web services* implementados utilizando componentes *EJB* 2.1 ou *POJOs* sem anotações. Já o segundo conjunto de interceptadores é compatível com a especificação *JAX-WS* (parte da plataforma *Java EE* 5) e é empregado por *Web services* implementados utilizando componentes *EJB* 3.0 ou *POJOs* com anotações. A especificação *Web Services Metadata for the Java Platform* [72] pode ser consultada para maiores informações sobre o uso de anotações em *Web services* na plataforma Java.

Em ambientes *Java EE*, o cliente tipicamente interage com um *Web service* utilizando um representante obtido por meio de uma busca num serviço de nomes através do *Java Naming and Directory Interface* (*JNDI*) [62]. Nesse contexto, um cliente pode ser uma aplicação Java *stand-alone*, um componente *EJB*, um *servlet*, um *Web service* ou qualquer outro componente *Java EE*. Tipicamente, a busca por tal representante ocorre no escopo do *Enterprise Naming Context* (*ENC*) [66] do cliente. Em outras palavras, a obtenção do representante de um *Web service* ocorre geralmente com um código bastante similar ao da Figura 4.24.

```
(1)   javax.naming.Context ctx = new javax.naming.InitialContext();
(2)   javax.xml.rpc.Service service = (javax.xml.rpc.Service)
(3)                                   ctx.lookup("java:comp/env/service/MyWebService");
(4)   MyServicePort port = service.getPort(MyServicePort.class);
```

Figura 4.24: Um cliente *Java EE* tipicamente obtém o representante de um *Web service* por meio de uma busca num serviço de nomes.

Na Figura 4.24, estamos pressupondo que o *Web service* implementa um *port type* que é

representado na linguagem Java pela interface `MyServicePort`. Para que a busca efetuada nas linhas 2 e 3 seja concluída com sucesso, o cliente precisa declarar um `service-ref` em seu descritor de implantação (no caso de *Java EE 5* isso também pode ser feito através de anotações). Um `service-ref`, como o próprio nome sugere, descreve uma referência para um *Web service*. Ele é análogo ao elemento `ejb-ref` do modelo *EJB*.

Caso o cliente deseje que o representante obtido via *JNDI* disponibilize suporte transacional, ele deve especificar, como parte desse `service-ref`, num descritor de implantação específico do *JBoss* (`jboss.xml`, `jboss-web.xml`, `jboss-client.xml`, etc.), um elemento `config-name` com o valor `Standard Transaction Client`, como mostrado na Figura 4.25.

```
<service-ref>
  <service-ref-name>service/MyWebService</service-ref-name>
  <config-name>Standard Transaction Client</config-name>
</service-ref>
```

Figura 4.25: Para que o representante de um *Web service* disponibilize suporte transacional, o cliente deve especificar como parte do `service-ref` um elemento `config-name` com o valor `Standard Transaction Client`.

Em outras palavras, para que o representante de um *Web service* passe a incluir suporte transacional basta alterar o descritor de implantação do cliente e incluir um elemento `config-name` com o valor `Standard Transaction Client`. Nenhuma modificação no código fonte é necessária. Todos os clientes configurados desse modo têm acesso, via *JNDI*, a um representante que possui um interceptador capaz de injetar o contexto transacional nas requisições *SOAP* efetuadas dentro do escopo de uma transação (Figura 4.26). É importante ressaltar que a cadeia de interceptadores presente no representante de um *Web service* é totalmente configurável. O elemento `config-name` é apenas uma das maneiras de incluir o interceptador transacional na cadeia de interceptadores de um representante.

Visto o lado cliente, passemos agora para o lado do servidor de aplicações. Neste último também há um interceptador, cujo papel é tratar as requisições *SOAP* recebidas, extraindo e importando o contexto transacional quando este é propagado juntamente com as requisições (Figura 4.27).

O procedimento para que um *Web service* inclua o interceptador transacional em sua cadeia de interceptadores depende do modo como tal *Web service* é implementado. Caso ele seja implementado utilizando um componente *EJB*, o descritor de implantação `jboss.xml` deve incluir um elemento `config-name` com o valor `Standard Transaction Endpoint`, como ilustrado na Figura 4.28. No caso de um componente *EJB* compatível com a versão 3.0 da especificação, há

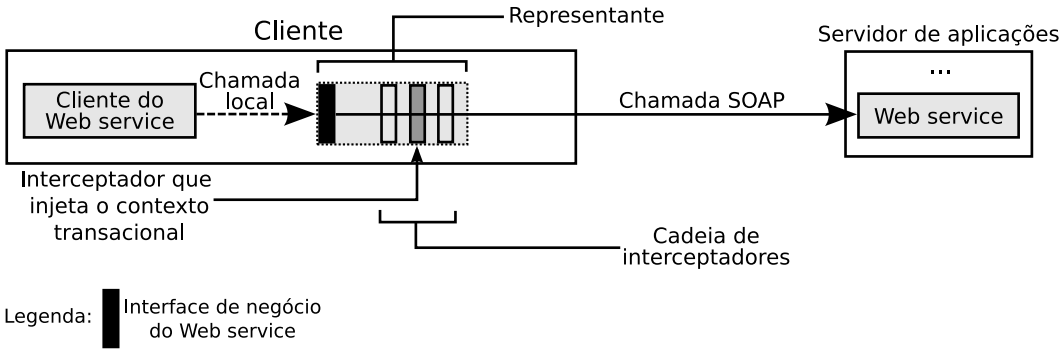



Figura 4.26: Uso de um interceptador do lado cliente para propagar transparentemente o contexto de coordenação.

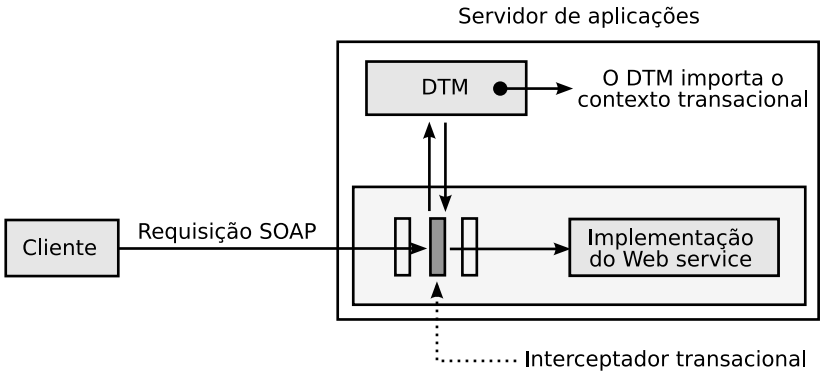



Figura 4.27: Do lado servidor há um interceptador que trata as requisições *SOAP* recebidas, importando o contexto transacional quando este é recebido juntamente com a requisição.

também a possibilidade de se utilizar anotações, conforme exibido na Figura 4.29. Por outro lado, caso o *Web service* seja implementado utilizando um *POJO*, o descritor de implantação `web.xml` deve incluir o excerto apresentado na Figura 4.30.

Qualquer uma dessas alterações nos descritores de implantação ou a adição da anotação `EndpointConfig` é suficiente para habilitar o suporte transacional num *Web service*. O suporte transacional para *Web services* pode, portanto, ser configurado de modo bastante simples tanto do lado cliente quanto do lado servidor.

Além do papel dos interceptadores, é importante descrever como o *DTM* importa o contexto transacional propagado juntamente com uma requisição. Conforme ilustrado na Figura 4.27, o interceptador do lado servidor delega ao *DTM* a tarefa de importar o contexto transacional. Mais precisamente, a tarefa em questão é delegada ao método `importTransactionPropa-`

```
<webservice-description>
  <webservice-description-name>MyWebService</webservice-description-name>
  <config-name>Standard Transaction Endpoint</config-name>
</webservice-description>
```

Figura 4.28: Para habilitar o suporte transacional, um elemento `config-name` com o valor `Standard Transaction Endpoint` deve ser incluído no descritor de implantação do componente *EJB*.

```
@EndpointConfig(configName = "Standard WSAtomicTransaction Endpoint")
public class ...
```

Figura 4.29: No caso de componentes *EJB* 3.0, há também a possibilidade de utilizar anotações para habilitar o suporte transacional.

`gationContext` da classe `org.jboss.tm.TxManager`. Esse método cria uma nova transação local (uma instância de `org.jboss.tm.TransacionImpl`) apenas quando não há no servidor de aplicações uma transação com mesmo o identificador global presente no contexto transacional. Em outras palavras, o método em questão cria uma transação local somente quando o contexto transacional é importado pela primeira vez. Chamadas subseqüentes com o mesmo contexto transacional não implicam na criação de uma nova transação local.

Quando uma transação local é criada através da importação de um contexto transacional, essa transação consiste num ramo de uma transação distribuída. Entretanto, o *DTM* não registra imediatamente o ramo da transação com o coordenador presente no contexto transacional. Ao invés disso, o *DTM* posterga a operação de registro até o momento em que ela se torna imprescindível. Isso ocorre, por exemplo, quando algum recurso *XA* é utilizado dentro do escopo do ramo da transação. Este último deve então ser registrado, na forma de um recurso remoto, com o coordenador presente no contexto transacional. Desse modo, o ramo da transação passa a receber as notificações do protocolo *2PC* quando o coordenador iniciar o procedimento de encerramento da transação distribuída. Vale ressaltar, entretanto, que em alguns casos o registro com o coordenador não ocorre. É o que acontece, por exemplo, quando um contexto transacional é importado, mas nenhum recurso transacional é utilizado dentro do

```
<context-param>
  <param-name>jbossws-config-name</param-name>>
  <param-value>Standard Transaction Endpoint</param-value>
</context-param>
```

Figura 4.30: *Web services* implementados utilizando *POJOs* devem incluir esse elemento no descritor de implantação `web.xml` para habilitar o suporte transacional.

escopo do ramo da transação.

## 4.6 A interação com o serviço de transações

As aplicações que necessitam de suporte transacional interagem com o *DTM* através da *Java Transaction API* (*JTA*) ou delegam a um contêiner *EJB* a tarefa de gerenciar transações. A *JTA* define interfaces de alto nível que permitem a uma aplicação demarcar explicitamente os limites de uma transação. Portanto, uma aplicação que deseja executar uma transação distribuída envolvendo *Web services*, demarcando explicitamente o escopo dessa transação, não precisa interagir diretamente com o *DTM* ou com os *port types* definidos em *WS-Coordination* e *WS-AtomicTransaction*. Ao invés disso, a aplicação interage somente com as interfaces de alto nível da *JTA*.

Para permitir a demarcação de transações, a *JTA* define as interfaces `javax.transaction.TransactionManager` e `javax.transaction.UserTransaction`. A primeira é utilizada pelo servidor de aplicações para efetuar a demarcação de transações em nome dos componentes que ele gerencia. Por exemplo, o contêiner *EJB* realiza a demarcação de transações nos componentes configurados para usar transações gerenciadas pelo contêiner (*Container-Managed Transactions*, ou *CMT*). Ele faz isso interagindo com o *DTM* através da interface `TransactionManager`. Já a segunda interface é empregada por componentes que fazem a demarcação explícita dos limites de uma transação. Esse é o caso dos *servlets*, dos componentes *EJB* configurados para usar transações gerenciadas pelo componente (*Bean-Managed Transactions*, ou *BMT*), dos *Web services* implementados utilizando *POJOs* e das aplicações Java *stand-alone*[10] (Figura 4.31).

A interface `javax.transaction.UserTransaction` é exibida na Figura 4.32. Suas operações sempre atuam sobre a transação vinculada à linha de execução que chama a operação. Por exemplo, o método `begin` cria uma nova transação e a associa à linha de execução chamadora. Os métodos `commit` e *rollback*, por sua vez, fazem respectivamente a efetivação e o aborto da transação associada à linha de execução chamadora. A operação `setRollbackOnly` coloca a transação vinculada à linha de execução chamadora num estado onde ela somente pode ser abortada. Por fim, as operações `getStatus` e `setTransactionTimeout` permitem obter o *status* e definir o tempo de expiração da transação associada à linha de execução chamadora. Como é possível observar, a interface `UserTransaction` possibilita a demarcação dos limites de uma transação de modo bastante simples.

[10] A especificação *Java EE* não exige que haja suporte transacional para aplicações *stand-alone*. O *JBoss*, contudo, oferece esse suporte.

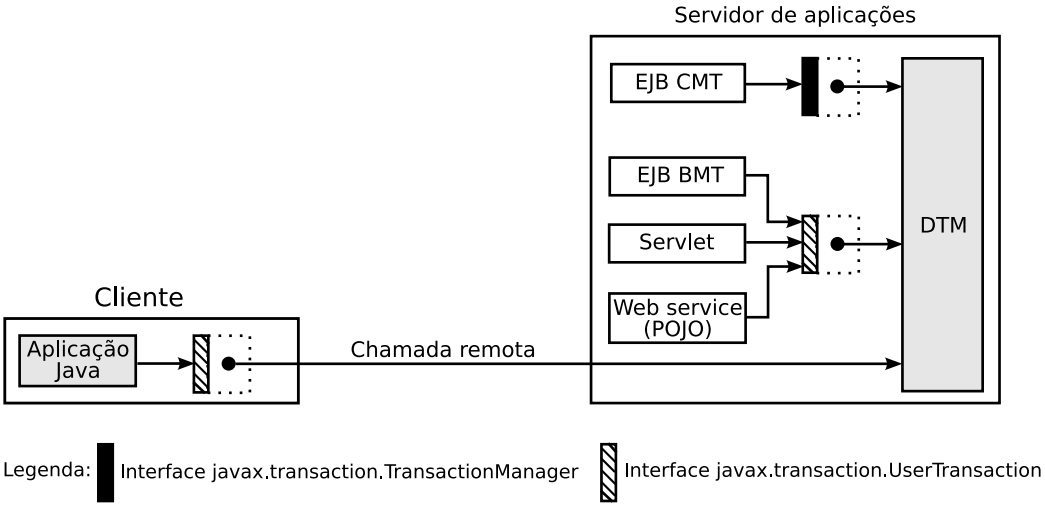



Figura 4.31: As aplicações interagem com o *DTM* através das interfaces de alto nível da *JTA*.

```
public interface javax.transaction.UserTransaction
{
   public void begin() throws ...;
   public void commit() throws ...;
   public void rollback() throws ...;
   public void setRollbackOnly() throws ...;
   public int getStatus() throws ...;
   public void setTransactionTimeout(int seconds) throws ...;
}
```

Figura 4.32: A interface `javax.transaction.UserTransaction`.

Os componentes que são executados na máquina virtual Java [44] do servidor de aplicações utilizam uma implementação da interface `UserTransaction` que efetua chamadas locais ao *DTM*. Já para as aplicações Java *stand-alone* que executam fora da máquina virtual do servidor de aplicações, a situação é um pouco mais complexa. Elas utilizam uma implementação de `UserTransaction` que se comunica com o *DTM* através de algum mecanismo de chamada remota.

No nosso trabalho, desenvolvemos uma implementação da interface `UserTransaction` que se comunica com o *DTM* através do protocolo *SOAP*. Na realidade, não há comunicação direta com o *DTM*, pois as chamadas *SOAP* são direcionadas aos *port types* definidos em *WS-Coordination* e *WS-AtomicTransaction*, que por sua vez as delegam ao *DTM* (Figura 4.33).

O exemplo apresentado na Figura 4.34 ilustra o uso da interface `UserTransaction` para efetuar a demarcação explícita de uma transação. Esse código poderia ser utilizado num *servlet*

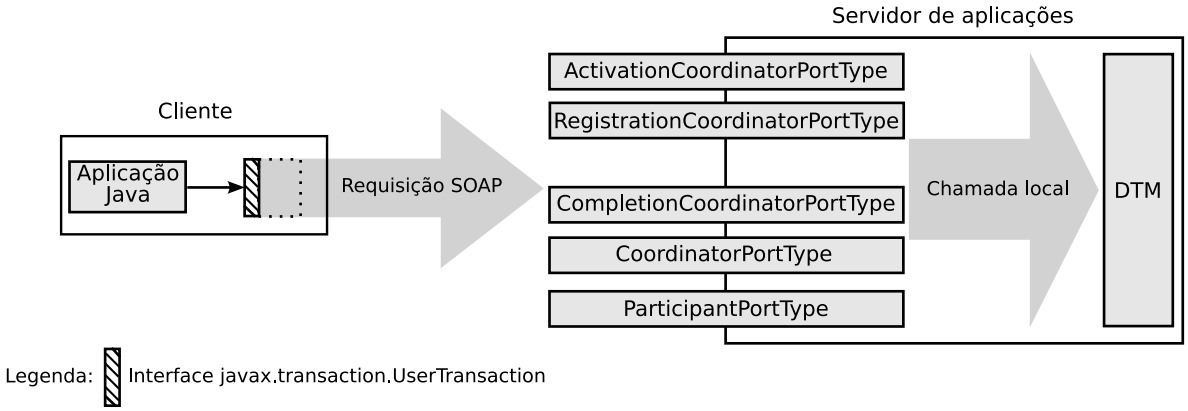



Figura 4.33: Desenvolvemos uma implementação de `UserTransaction` se comunica via *SOAP* com os *port types* definidos em *WS-Coordination* e *WS-AtomicTransaction*. Estes, por sua vez, delegam ao *DTM* as requisições do cliente.

ou numa aplicação Java *stand-alone*. No exemplo, nota-se que a primeira ação da aplicação é a obtenção de uma referência para uma implementação de `UserTransaction`. No caso dos *servlets* e das aplicações Java *stand-alone*, isso é feito através de uma busca num serviços de nomes através do *JNDI* (linhas 1 e 2). Esse procedimento pode, entretanto, variar. No caso dos componentes *EJB* configurados para usar *BMT*, por exemplo, uma implementação de `UserTransaction` é obtida utilizando o método `getUserTransaction` da interface `javax.ejb.EJBContext`.

```
(1)   InitialContext context = new InitialContext();
(2)   UserTransaction userTx = (UserTransaction) context.lookup("UserTransaction");
(3)   userTx.begin();
(4)      webservice1.umaOperacaoDeNegocio();
(5)      webservice2.outraOperacaoDeNegocio();
(6)      ...
(7)   userTx.commit();
```

Figura 4.34: Exemplo de demarcação explícita de uma transação utilizando a interface `UserTransaction`.

A seguir, o exemplo utiliza o método `begin` da interface `UserTransaction` para iniciar uma transação (linha 3). Uma simples chamada a esse método é tudo o que a aplicação precisa para iniciar uma transação, pois todas as ações necessárias para efetivamente criar a transação são executadas implicitamente pela implementação da interface `UserTransaction`. A aplicação chama então as operações de alguns *Web services* de negócio (linhas 4 a 6), sendo que tais operações são executadas como parte da transação corrente (em outras palavras, elas

propagam o contexto transacional). Vale ressaltar que a aplicação chama as operações de negócio normalmente, isto é, da mesma forma que ela as chamaria no caso não transacional. Concluído o trabalho que deve ser efetuado no contexto da transação, a aplicação chama o método `commit` para efetivá-la (linha 7). Novamente, isso é tudo o que a aplicação precisa fazer, pois todas as ações necessárias para efetivar a transação são executadas transparentemente pela implementação de `UserTransaction`.

O exemplo mostra o quão simples é a demarcação transacional do ponto de vista da aplicação. Tudo o que uma aplicação precisa fazer é indicar o início e o fim de uma transação, explicitando se a transação deve ser efetivada ou abortada. Toda a complexidade envolvida numa transação distribuída (os mecanismos de chamada remota utilizados na comunicação entre os membros da transação, o registro de participantes com o coordenador, a execução do protocolo *2PC*, etc.) é gerenciada transparentemente pelo middleware. Cabe lembrar, ainda, que a demarcação de transações pode ser feita automaticamente pelo servidor de aplicações, retirando da aplicação toda e qualquer responsabilidade por interações explícitas com o serviço de transações. Isso é o que ocorre com os componentes *EJB* configurados para utilizar *CMT*.

Embora o exemplo em discussão não efetue chamadas *JBoss Remoting* ou *CORBA* dentro do escopo da transação, ele poderia fazê-lo. Tais chamadas também propagariam o contexto transacional e, portanto, seriam executadas como parte da transação distribuída. Novamente, toda a complexidade envolvida na execução da transação seria gerenciada transparentemente pelo middleware.

Para finalizar a discussão sobre a interação com o serviço de transações, cabe ressaltar duas características da *JTA*. Primeiramente, a *JTA* não contempla transações aninhadas. Isso quer dizer que, caso uma aplicação chame o método `begin` da interface `UserTransaction` quando uma transação já estiver associada à linha de execução chamadora, uma exceção será lançada. Em segundo lugar, a *JTA* especifica que transações devem ser associadas a linhas de execução. É possível, portanto, escrever uma aplicação com múltiplas linhas de execução que executam transações concorrentemente. Para contemplar esse cenário, as implementações da interface `UserTransaction` do *JBoss* empregam o padrão *thread-specific storage* [59].

## 4.7 Os testes de unidade

Elaboramos mais de uma centena de testes de unidade para o nosso serviço de transações. Esses testes foram escritos utilizando o arcabouço de testes *JUnit* [46] e estão divididos em quatro conjuntos:

- O primeiro conjunto verifica o correto funcionamento das operações dos *port types*;

- O segundo conjunto verifica o correto funcionamento da implementação de `UserTransaction` que se comunica via *SOAP* com o *DTM*;
- O terceiro conjunto verifica o correto funcionamento do *DTM* na coordenação de transações distribuídas envolvendo *Web services*;
- O quarto e último conjunto verifica o correto funcionamento do *DTM* na coordenação de transações distribuídas envolvendo recuperação de falhas.

Os testes do primeiro conjunto são os mais simples. Eles são os responsáveis por exercitar as implementações dos *port types* definidos em *WS-Coordination* e *WS-AtomicTransaction*. Por exemplo, um dos testes desse conjunto verifica se um `ParticipantPortType` responde a uma mensagem `Rollback` com uma mensagem `Aborted`. Há ainda testes que enviam mensagens inválidas para os *port types* e verificam as ações estes tomam. Por fim, há também testes que verificam o formato e a correta transmissão das mensagens *SOAP* enviadas pelos *port types*. Alguns testes, por exemplo, verificam se as informações de endereçamento definidas em *WS-Addressing* estão presentes nas mensagens. Outros testes verificam se as conexões *HTTP* estão sendo utilizadas corretamente (por exemplo, se elas estão retornando o código de resposta *HTTP* correto). Os testes presentes no primeiro conjunto exercitam todas as mensagens definidas em *WS-Coordination* e *WS-AtomicTransaction*.

Os testes do segundo conjunto exercitam a implementação de `UserTransaction` que se comunica com o servidor de aplicações via *SOAP*. Todas as operações da interface `UserTransaction` são exercitadas. Há, inclusive, testes que verificam o correto funcionamento da nossa implementação de `UserTransaction` quando múltiplas linhas de execução a utilizam concorrentemente.

O terceiro conjunto de testes verifica o correto funcionamento do nosso serviço de transações para *Web services* na coordenação de transações distribuídas. Os testes desse conjunto sempre envolvem três instâncias do servidor de aplicações *JBoss*: um coordenador e dois participantes (Figura 4.35). Em alguns dos testes, os participantes utilizam recursos transacionais *XA* (bancos de dados *Apache Derby* [83]). Em outros testes, os participantes utilizam recursos *XA* arremedados (*mocks*). Isso foi feito para que fosse possível manipular os votos dos participantes na primeira fase do *2PC*. Por exemplo, se um dos testes precisa que um participante vote "sim" e que outro vote "não" para a efetivação da transação, basta registrar com os participantes recursos arremedados que dêem essas respostas. Forçar um recurso *XA* real a votar "não" exigiria induzi-lo a uma falha, e por esse motivo o uso de recursos arremedados é interessante. Eles nos permitem controlar os votos dos participantes e conseqüentemente exercitar o coordenador com várias combinações de votos.

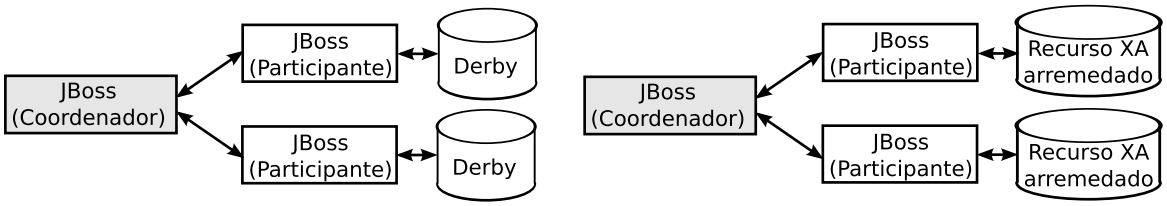



Figura 4.35: Os testes do terceiro conjunto envolvem sempre dois participantes, que são instâncias do *JBoss*. Em alguns testes os participantes utilizam recursos transacionais *XA* (bancos de dados *Derby*), enquanto em outros os participantes utilizam recursos *XA* arremedados.

Por fim, temos o quarto conjunto de testes. Nele existem testes que verificam o correto funcionamento do serviço de transações na presença de falhas. Os testes desse conjunto forçam a queda de um dos servidores de aplicações, que pode ser o coordenador ou um dos participantes. Em cada teste há um coordenador e dois ou três participantes. As quedas dos servidores de aplicações ocorrem nos seguintes pontos do protocolo *2PC*:

- Queda do coordenador após a decisão de efetivar a transação ter sido salva em meio persistente: o coordenador é encerrado imediatamente após a escrita de um registro `MULTI_TM_TX_COMMITTED` no seu *log* local. Nesse caso, o procedimento de recuperação executado no coordenador irá se encarregar de efetivar a transação distribuída quando o coordenador voltar à execução.

- Queda do coordenador após a apuração dos votos, mas antes da decisão de efetivar a transação ter sido salva em meio persistente: o coordenador é encerrado imediatamente antes da escrita de um registro `MULTI_TM_TX_COMMITTED` no seu *log* local. Nesse caso, o procedimento de recuperação executado no coordenador, juntamente com a mensagem `Replay` enviada pelo participante, irão se encarregar de abortar a transação distribuída.

- Queda de um participante após o seu voto "sim" ter sido tanto salvo em meio persistente quanto informado ao coordenador: o participante é encerrado após adicionar um registro `TX_PREPARED` ao seu *log* local e enviar de uma mensagem `Prepared` ao coordenador. Nesse caso, após o restabelecimento do participante, o procedimento de recuperação executado neste irá enviar uma mensagem `Replay` ao coordenador, que por sua vez fará com que o ramo da transação existente no participante seja efetivado.

- Queda de um participante após o seu voto "sim" ter sido salvo em meio persistente, porém antes do voto ser informado ao coordenador: o participante é encerrado imediatamente após a adição de um registro `TX_PREPARED` ao seu *log* local. Desse modo, a queda

ocorre antes do participante enviar a mensagem `Prepared` ao coordenador. Nesse caso, o procedimento de recuperação executado no participante irá enviar uma mensagem `Replay` ao coordenador, que por sua vez fará com que o ramo da transação existente no participante seja abortado.

- Queda de um participante antes do seu voto "sim" ter sido salvo em meio persistente: o participante é encerrado antes de adicionar ao seu *log* local um registro `TX_PREPARED` (e portanto antes de enviar uma mensagem `Prepared` ao coordenador). Nesse caso não há o envio de uma mensagem `Replay` ao coordenador. O procedimento de recuperação executado no participante apenas aborta o ramo da transação.

Os testes com recuperação de falhas envolvem até três participantes e as quedas são simuladas em cada um desses participantes. Por exemplo, numa fração dos testes o participante que sofre quedas é o primeiro, já numa outra fração o participante que sofre quedas é o segundo, e por fim numa última fração o participante que sofre quedas é o terceiro. Há também testes em que o coordenador sofre quedas.

Os mecanismos de chamada remota utilizados para se comunicar com os participantes também variam de teste para teste. Até três mecanismos são utilizados durante a execução de um teste. Há testes, por exemplo, que envolvem simultaneamente recursos remotos acessíveis via *JBoss Remoting*, *IIOP* e *SOAP*. Em alguns testes o participante a sofrer quedas é o acessível via *JBoss Remoting*, em outros é o acessível via *IIOP*, e em outros é o acessível via *SOAP*. Desse modo, são testadas quedas de participantes acessíveis através de cada um dos mecanismos de chamada remota contemplados pelo *DTM*.

# 5 Resultados experimentais

Neste capítulo apresentamos os resultados experimentais obtidos com a nossa versão estendida do gerenciador de transações do *JBoss*. O capítulo encontra-se dividido em duas seções. A Seção 5.1 apresenta uma avaliação de desempenho do gerenciador na coordenação de transações que empregam cada um dos seguintes mecanismos de chamada remota: *JBoss Remoting*, *CORBA/IIOP* e *Web services/SOAP*. A Seção 5.2 discute os experimentos de interoperabilidade.

## 5.1 Avaliação de desempenho

Desempenho é indubitavelmente uma das características mais importantes de um sistema gerenciador de transações. Por esse motivo, realizamos uma série de experimentos de avaliação de desempenho envolvendo o gerenciador de transações do *JBoss*. Em particular, avaliamos: *(i)* a sobrecarga imposta pelo uso de um protocolo baseado em *XML* (*SOAP*) em relação a protocolos binários (*JBoss Remoting* e *IIOP*), *(ii)* a sobrecarga imposta pela propagação do contexto transacional e *(iii)* a sobrecarga total imposta pelo uso de um gerenciador de transações. Ademais, efetuamos também experimentos de escalabilidade.

### 5.1.1 O ambiente de testes

O ambiente de execução utilizado nos experimentos foi composto por computadores equipados com um processador *Intel Pentium* 4 de 3.0 GHz (com suporte a *HyperThreading*) e 1 GB de memória *RAM*. As máquinas estavam interconectadas através de uma rede *Ethernet* [47] de 100 Mbit/s e executavam o sistema operacional *Linux* versão 2.6.17 (*SMP*).

Utilizamos o *Sun Java SE Development Kit* (*JDK*) [58], versão 5.0, e a versão de desenvolvimento (*SVN tag 5.0.0.Beta2_JBossWS_2_0_0_GA*) do servidor de aplicações *JBoss*, juntamente com as modificações que efetuamos para contemplar transações distribuídas envolvendo *Web services*. Mais precisamente, a versão utilizada do *JBoss* incluía os seguintes componentes: *JBoss Remoting* 2.2.0.GA, *JBoss Serialization* [38] 1.0.3.GA, *JacORB* [16] 2.2.4.jboss.patch1,

*JBossWS* [40] 2.0.0.GA e *Woodstox* [81] 3.1.1. Exceto quando expresso o contrário, os experimentos foram sempre executados com o cliente e o servidor em máquinas distintas.

Realizamos três ajustes nas configurações originais do *JBoss*. Primeiramente, o uso do *log*[1] no servidor de aplicações foi desabilitado durante a realização dos experimentos. Desse modo, escritas no *log* não influenciaram os nossos resultados. Em segundo lugar, o *JBoss Remoting*, que por padrão utiliza seriação Java, foi configurado para utilizar um mecanismo de seriação próprio do *JBoss*, denominado *JBoss Serialization*. Este último é consideravelmente mais eficiente do que a seriação Java. Assim sendo, acreditamos que esse ajuste possibilitou uma avaliação de desempenho mais justa no caso do *JBoss Remoting*. Por fim, aumentamos o tamanho do *heap* da máquina virtual Java para 768 MB, com o objetivo de minimizar o número de coletas de lixo executadas durante os experimentos (um *heap* pequeno implica em execuções mais freqüentes do coletor de lixo). Mais precisamente, foram utilizados os parâmetros `-Xms768m` e `-Xmx768m` na inicialização das máquinas virtuais do cliente e do servidor.

Nos experimentos de escalabilidade, utilizamos o *Apache JMeter* [4] para simular um grande número de clientes. O *JMeter* permite que múltiplos computadores sejam coordenados para efetuar requisições simultâneas a uma só máquina (o servidor, que no nosso caso é o gerenciador de transações). Em todos os experimentos de escalabilidade, tomamos as medidas necessárias para que o limitante da vazão do sistema nunca estivesse presente nos clientes. Em outras palavras, utilizamos um número de computadores clientes suficientemente grande para que os clientes não representassem o gargalo de desempenho nos experimentos.

### 5.1.2 Metodologia

Cada um dos experimentos foi composto por duas fases. Na primeira delas, cinco mil chamadas remotas foram executadas com o objetivo de colocar o sistema num estado estável. A seguir, tomadas de tempo[2] foram efetuadas para quinze mil chamadas consecutivas e várias estatísticas foram então calculadas tendo como base essas quinze mil medições. Cada um dos experimentos envolveu, portanto, a execução de vinte mil chamadas remotas.

Antes da execução de cada um dos experimentos, os servidores de aplicações envolvidos foram reiniciados. Isso tinha como objetivo garantir sempre as mesmas condições iniciais para todos os experimentos.

[1] Estamos nos referindo ao *log* de rastreamento, que registra informações sobre o funcionamento do servidor de aplicações. Os experimentos foram realizados com o *log* transacional habilitado.
[2] Para todas as medições foi utilizada a operação `System.nanoTime()` do *JDK*.

### 5.1.3 Estimativa da sobrecarga imposta pelo uso de um protocolo baseado em XML

Com o objetivo de estimar a sobrecarga imposta pelo uso de um protocolo baseado em *XML* (*SOAP*) em relação a protocolos binários (*JBoss Remoting*[3] e *IIOP*), efetuamos uma comparação de desempenho entre o *JBoss Remoting*, o *IIOP* e o *SOAP*. No decorrer dos experimentos, mensuramos no cliente a duração de diversas chamadas remotas. A duração de uma chamada remota consiste no tempo transcorrido entre o início da chamada remota e o fim da devolução dos resultados ao cliente.

Utilizamos nos experimentos um componente *EJB* de sessão sem estado que implementa as operações da interface exibida na Figura 5.1. O componente, compatível com a versão 2.1 da especificação *EJB*, foi configurado para ser acessível simultaneamente via *JBoss Remoting*, *IIOP* e *SOAP*. Desse modo, a classe que trata as chamadas remotas é sempre a mesma, independentemente do protocolo de comunicação utilizado.

```
public interface RoundTripTime extends javax.ejb.EJBObject
{
   int[] getInt100() throws RemoteException;
   double[] getDouble100() throws RemoteException;
   String getString100() throws RemoteException;
   String getString1000() throws RemoteException;
   String getString10000() throws RemoteException;
   DataSet getDataSet() throws RemoteException;
   DataSet[] getDataSet10() throws RemoteException;
   DataSet[] getDataSet100() throws RemoteException;
}
```

Figura 5.1: A interface `RoundTripTime`, implementada pelo componente *EJB* de sessão sem estado utilizado nos experimentos.

Os métodos do componente *EJB* foram implementados de maneira trivial. Eles simplesmente retornam um valor fixo e, portanto, não executam nenhum processamento significativo. Conseqüentemente, a duração de uma chamada remota ao componente consiste num bom indicador da sobrecarga imposta pelo protocolo de comunicação empregado. Vários tipos de dados foram transmitidos com o objetivo de determinar qual a influência deles sobre a duração das

[3] Diferentemente do *IIOP* e do *SOAP*, o *JBoss Remoting* é um arcabouço para chamadas remotas, e não um protocolo de comunicação. Trata-se de um arcabouço bastante flexível, que comporta chamadas remotas sobre uma variedade de protocolos de comunicação. No caso dos nossos experimentos, o transporte das chamadas *JBoss Remoting* ocorreu através de um protocolo próprio do *JBoss*, baseado no *JBoss Serialization*. No decorrer deste capítulo, utilizaremos o termo "*JBoss Remoting*" como se ele fosse um protocolo de comunicação. Portanto, quando nos referirmos ao "protocolo" *JBoss Remoting*, estaremos na verdade dizendo que a comunicação ocorre através de um protocolo baseado no *JBoss Serialization*.

chamadas. Como é possível observar, os dois primeiros métodos da interface apresentada na Figura 5.1 devolvem um vetor de tipos primitivos Java (o vetor devolvido tem comprimento 100). Os métodos `getString100`, `getString1000` e `getString10000` devolvem, respectivamente, cadeias de caracteres de comprimento 100, 1000 e 10000. Já os três últimos métodos, denominados `getDataSet`, `getDataSet10` e `getDataSet100`, devolvem uma, dez e cem instâncias da classe `DataSet`. Essa classe consiste num agregado composto por tipos primitivos e uma cadeia de caracteres. Nos nossos experimentos, a cadeia de caracteres presente num `DataSet` apresentava sempre comprimento dez. A Figura 5.2 exibe a classe `DataSet`.

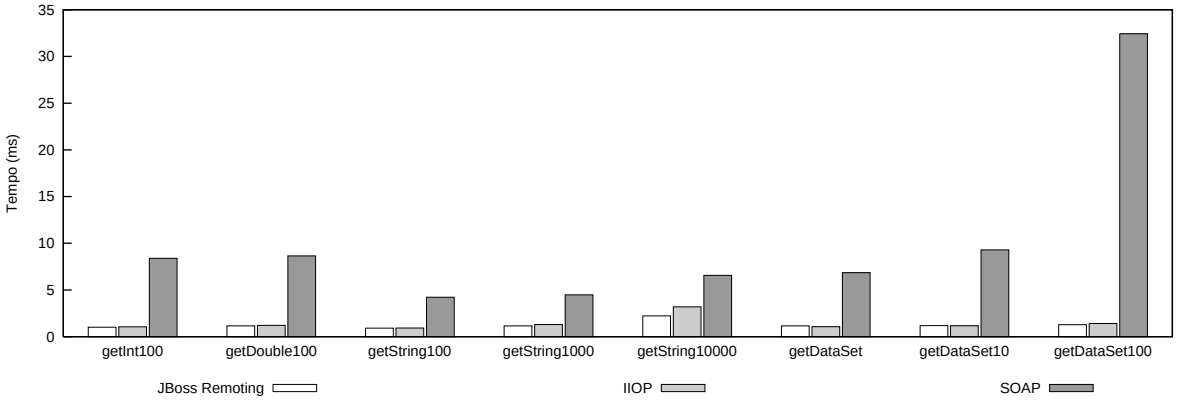
```
public class DataSet implements Serializable
{
   private byte _byte;
   private boolean _boolean;
   private short _short;
   private int _int;
   private long _long;
   private float _float;
   private double _double;
   private String _string;
   ...
}
```

Figura 5.2: A classe `DataSet`.

No nosso primeiro conjunto de experimentos, o componente *EJB* foi configurado com o atributo transacional `Never` para todos os seus métodos. Isso significa que não houve nenhuma sobrecarga transacional, o que possibilitou uma melhor estimativa da sobrecarga imposta pelo uso dos diferentes protocolos de comunicação. A Figura 5.3 mostra os resultados do nosso primeiro conjunto de experimentos. A Figura 5.3a apresenta a duração média das chamadas remotas a cada uma das operações da interface `RoundTripTime`. Na tabela da Figura 5.3b, encontramos diversas estatísticas sobre as chamadas remotas: a média, o desvio padrão, o máximo, o mínimo e o intervalo de confiança (I.C.) de 95%. A Figura 5.3c mostra os tamanhos, em bytes, das mensagens trocadas entre o cliente e o servidor.

É importante esclarecer um detalhe sobre os tempos apresentados nas Figuras 5.3a e 5.3b: eles não incluem o período de estabelecimento de conexão com o servidor. Em todos os experimentos o cliente estabelecia uma conexão com o servidor e a reutilizava para efetuar as subseqüentes chamadas. Isso reduziu consideravelmente a duração média de uma chamada remota.

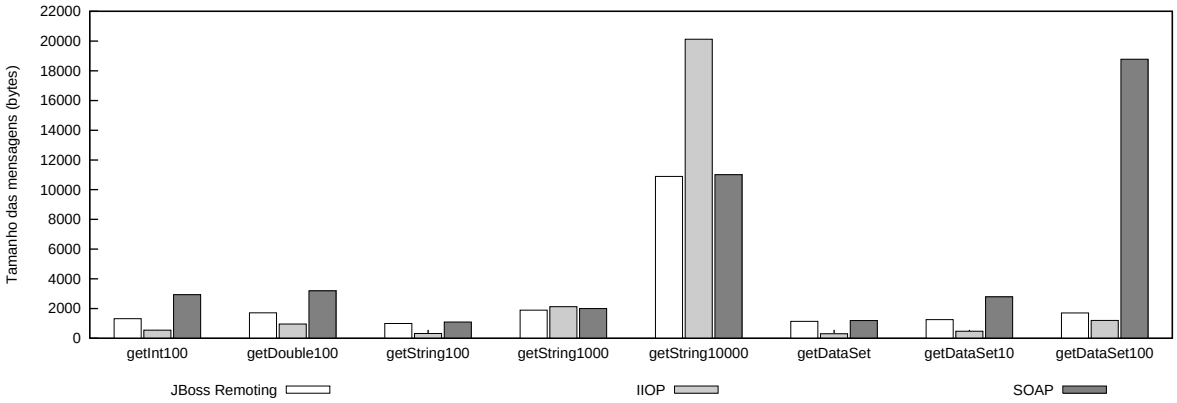
Feita essa ressalva, analisemos agora a Figura 5.3a. Ela nos mostra que o desempenho do *JBoss Remoting* e do *IIOP* são bastante semelhantes. Particularmente, aquele apresenta uma




(a) Durações médias das chamadas remotas.

| | | int100 | double100 | string100 | string1000 | string10000 | dataSet | dataSet10 | dataSet100 |
|---|---|---|---|---|---|---|---|---|---|
| JBoss Remoting | Média | 1,02 | 1,15 | 0,92 | 1,16 | 2,23 | 1,15 | 1,18 | 1,28 |
| | Desvio padrão | 6,10 | 6,04 | 6,10 | 6,05 | 8,15 | 6,05 | 6,31 | 6,34 |
| | Mínimo | 0,84 | 0,86 | 0,72 | 0,94 | 1,86 | 0,87 | 0,87 | 1,07 |
| | Máximo | 745,13 | 735,37 | 742,23 | 736,61 | 740,96 | 736,02 | 734,13 | 736,16 |
| | I.C. (95%) | [0,92; 1,11] | [1,06; 1,25] | [0,83; 1,02] | [1,06; 1,26] | [2,10; 2,36] | [1,05; 1,25] | [1,08; 1,28] | [1,18; 1,38] |
| IIOP | Média | 1,05 | 1,21 | 0,93 | 1,30 | 3,20 | 1,07 | 1,17 | 1,42 |
| | Desvio padrão | 7,57 | 7,63 | 7,39 | 8,28 | 8,47 | 8,90 | 8,95 | 9,14 |
| | Mínimo | 0,72 | 0,74 | 0,58 | 0,86 | 2,82 | 0,71 | 0,73 | 0,97 |
| | Máximo | 914,18 | 924,20 | 894,40 | 903,53 | 908,45 | 942,16 | 948,68 | 947,65 |
| | I.C. (95%) | [0,93; 1,17] | [1,09; 1,33] | [0,81; 1,05] | [1,17; 1,43] | [3,06; 3,33] | [0,92; 1,21] | [1,02; 1,31] | [1,28; 1,57] |
| SOAP | Média | 8,40 | 8,65 | 4,23 | 4,48 | 6,57 | 6,85 | 9,29 | 32,43 |
| | Desvio padrão | 19,67 | 20,61 | 15,97 | 17,50 | 24,14 | 17,50 | 19,94 | 37,65 |
| | Mínimo | 7,21 | 7,43 | 3,44 | 3,54 | 5,09 | 5,83 | 8,11 | 29,10 |
| | Máximo | 716,76 | 722,45 | 785,99 | 757,74 | 705,60 | 753,16 | 742,50 | 831,32 |
| | I.C. (95%) | [8,08; 8,71] | [8,32; 8,98] | [3,97; 4,49] | [4,20; 4,76] | [6,19; 6,96] | [6,57; 7,13] | [8,97; 9,61] | [31,83; 33,04] |

(b) Estatísticas sobre as durações das chamadas remotas. Todos os tempos são expressos em milissegundos.




(c) Tamanhos das mensagens trocadas entre o cliente e o servidor.

Figura 5.3: Os resultados das chamadas remotas às operações da interface `RoundTripTime`. O componente *EJB* estava configurado com o atributo transacional `Never`.

pequena vantagem sobre este na maioria dos casos. O protocolo *SOAP*, entretanto, exibe um desempenho bastante inferior, sendo cerca de 20 vezes mais lento do que o *JBoss Remoting* e o *IIOP* no caso da operação `getDataSet100`.

Ainda analisando a Figura 5.3a, é possível notar que os tipos de dados utilizados influenciaram de maneira significativa os tempos de execução das chamadas *SOAP*. Em particular, as operações que devolvem tipos compostos (`getDataSet`, `getDataSet10` e `getDataSet100`) ou vetores de tipos primitivos (`getInt100` e `getDouble100`) apresentaram um tempo de execução consideravelmente maior do que as operações que devolvem cadeias de caracteres. Tomemos um exemplo: uma chamada *SOAP* à operação `getString10000` teve uma duração média de $6,57$ milissegundos, enquanto uma chamada à operação `getDataSet10` apresentou uma duração média de $9,29$ milissegundos. Cabe ressaltar que a chamada à operação `getString10000` foi mais rápida apesar desta envolver mensagens maiores do que as da operação `getDataSet10`. Conforme exibido na Figura 5.3c, a chamada `getString10000` envolve uma troca de mensagens *SOAP* que totaliza 11010 bytes. Já para a chamada `getDataSet10`, esse valor é de 2788 bytes. A operação `getString10000` apresentou um tempo de execução menor porque suas mensagens continham menos elementos *XML* do que as mensagens da operação `getDataSet`. Esse mesmo argumento pode ser aplicado para justificar o menor tempo de execução da operação `getString10000` em relação às operações `getInt100` e `getDouble100`. Os resultados sugerem, portanto, que a principal fonte de lentidão do protocolo *SOAP* não é prolixidade de suas mensagens, mas sim a manipulação destas por meio de um analisador *XML*.

Há ainda mais um fato que aponta o processamento das mensagens *SOAP* como o principal gargalo de desempenho desse protocolo. A Figura 5.3c mostra que, para a operação `getString1000`, os tamanhos das mensagens *JBoss Remoting* e *SOAP* são aproximadamente os mesmos. Se o tamanho das mensagens constituísse o principal gargalo de desempenho do *SOAP*, as tomadas de tempo apresentadas por este e pelo *JBoss Remoting* deveriam ser parecidas. Contudo, não é isso o que ocorre. Como a Figura 5.3a mostra, o *JBoss Remoting* exibe um desempenho muito superior ao do *SOAP*. Conseqüentemente, podemos concluir que o mecanismo de seriação *XML* utilizado pelo *SOAP* é muito mais ineficiente do que o mecanismo de seriação (*JBoss Serialization*) empregado pelo *JBoss Remoting*. Em outras palavras, o *SOAP* foi mais lento do que o *JBoss Remoting* nos nossos experimentos principalmente devido à manipulação das mensagens *XML*. Outros trabalhos [1,23,29] também apontam o processamento das mensagens *SOAP* como o principal gargalo de desempenho do protocolo.

No caso dos servidores de aplicações *Java EE*, como é o caso do *JBoss*, não é difícil entender por que o processamento das mensagens *SOAP* é tão custoso. A especificação *JAX-RPC*, parte das plataformas *J2EE* 1.4 e *Java EE* 5.0, define um ambiente bastante flexível para a

execução de *Web services*. Esse mesmo ambiente é também, entretanto, complexo e ineficiente. Em particular, a *API* utilizada para o processamento das mensagens *SOAP* é baseada no *DOM*, que prioriza a facilidade de manipulação das mensagens *XML* e não a eficiência. Além disso, como discutimos na Seção 4.4, o processamento de uma requisição enviada a um *Web service* compatível com a *JAX-RPC* envolve muito mais do que o uso de uma *API* baseada no *DOM*. O tratamento de uma requisição *SOAP* num ambiente *JAX-RPC* é, portanto, bastante oneroso.

É importante salientar que, apesar do tamanho das mensagens *SOAP* não ter influenciado significantemente os resultados dos nossos experimentos (por exemplo, a diferença do tempo médio entre as operações `getString100` e `getString1000` foi de apenas $0,25$ milissegundos), este é um fator primordial numa ampla variedade de cenários. Por exemplo, quando um cliente se comunica com o servidor através de um enlace de baixa velocidade (um enlace *ADSL*, por exemplo), tipicamente os custos de comunicação excedem em várias ordens de magnitude os custos de processamento das mensagens. Nesse cenário, o tamanho das mensagens tenderá a ser o fator dominante no tempo de execução de uma chamada remota.

Analisemos agora a Figura 5.3c. Primeiramente, esclarecemos que os números apresentados nessa figura contabilizam apenas os bytes que compõem a carga útil dos segmentos *TCP* envolvidos nas trocas de mensagens. A figura nos mostra que as mensagens *JBoss Remoting* e *IIOP* são, geralmente, mais enxutas do que as mensagens *SOAP* equivalentes. Um fato interessante ocorre com o *IIOP*: quando esse protocolo transporta cadeias de caracteres longas, suas mensagens tornam-se as maiores entre os três protocolos (veja o caso das operações `getString1000` e `getString10000`). Isso ocorre porque, por padrão, o *IIOP* (na realidade o *JacORB*) emprega o padrão de codificação *UTF-16* [41], que tipicamente utiliza dois bytes para representar um caractere. No caso do *JBoss Remoting* e do *SOAP*, o padrão de codificação empregado é o *UTF-8* [41], que geralmente utiliza apenas um byte para representar cada caractere.

As Figuras 5.4a, 5.4b e 5.4c apresentam a duração de cada uma das 20000 chamadas remotas efetuadas sobre cada protocolo (*JBoss Remoting*, *IIOP* e *SOAP*) à operação `getString100`. Em cada uma das duas primeiras figuras, há apenas uma perturbação significativa. Essa perturbação corresponde à execução de uma coleta de lixo completa (*full garbage collection*) efetuada pela máquina virtual Java. Na terceira figura, entretanto, há uma série de perturbações, sendo algumas bastante salientes (cerca de 800 milissegundos), e outras nem tanto (por volta de 200 milissegundos). Aquelas são causadas por coletas completas de lixo, enquanto que estas são o resultado de coletas parciais (*minor garbage collection*). Nos dados exibidos nas Figuras 5.3a e 5.3b, foram incluídas as perturbações geradas pelo coletor de lixo. Optamos

por não excluir essas perturbações pois elas fazem parte da execução normal de um programa.

Ainda nas Figuras 5.4a, 5.4b e 5.4c, é possível notar que as tomadas de tempo das 30 primeiras chamadas transacionais estão destacadas. Dado que antes da execução de cada experimento os servidores de aplicações envolvidos foram reiniciados, o tempo da primeira chamada remota é consideravelmente superior aos tempos das chamadas subseqüentes.

Concluindo a discussão sobre o nosso primeiro conjunto de experimentos, podemos dizer que *JBoss Remoting* foi, na maioria dos experimentos, um pouco mais rápido que o *IIOP*, sendo o protocolo *SOAP* até 20 vezes mais lento do que esses dois.

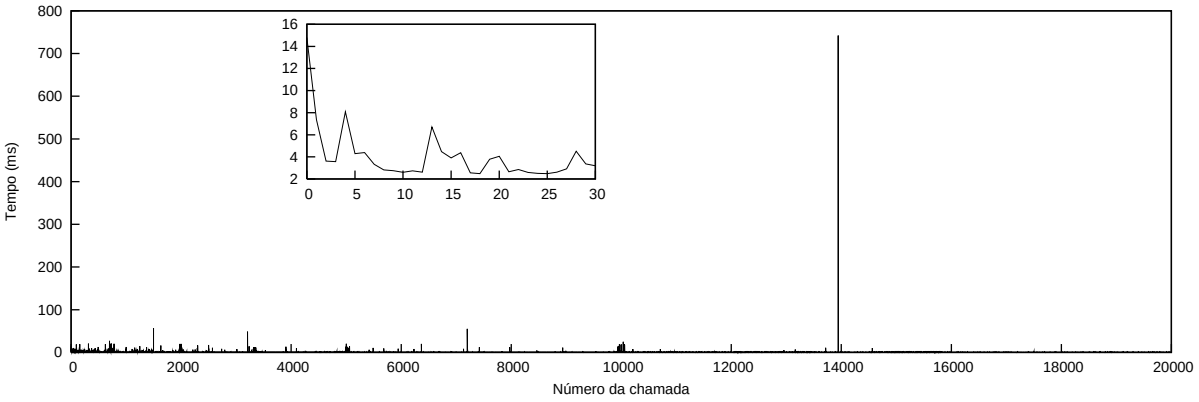
### 5.1.4 Estimativa da sobrecarga imposta pela propagação do contexto transacional

Conforme discutimos no Capítulo 4, o contexto transacional é propagado juntamente com todas as chamadas remotas efetuadas dentro do escopo de uma transação. Essa propagação impõe uma sobrecarga sobre o tempo de execução das chamadas remotas. Para estimar a intensidade dessa sobrecarga, realizamos uma série de experimentos que efetuavam chamadas remotas transacionais. Os resultados são apresentados na Figura 5.5. A Figura 5.5a apresenta a duração média das chamadas remotas a cada uma das operações da interface `RoundTripTime`, com destaque para o aumento na duração das chamadas devido à propagação do contexto transacional. Na tabela da Figura 5.5b, encontramos diversas estatísticas sobre as chamadas transacionais remotas: a média, o desvio padrão, o máximo, o mínimo e o intervalo de confiança (I.C.) de 95%. A Figura 5.5c exibe o tamanho das mensagens trocadas entre o cliente e o servidor, sendo que a parte mais escura das barras representa o tamanho do contexto transacional que é incluído nas mensagens de requisição.

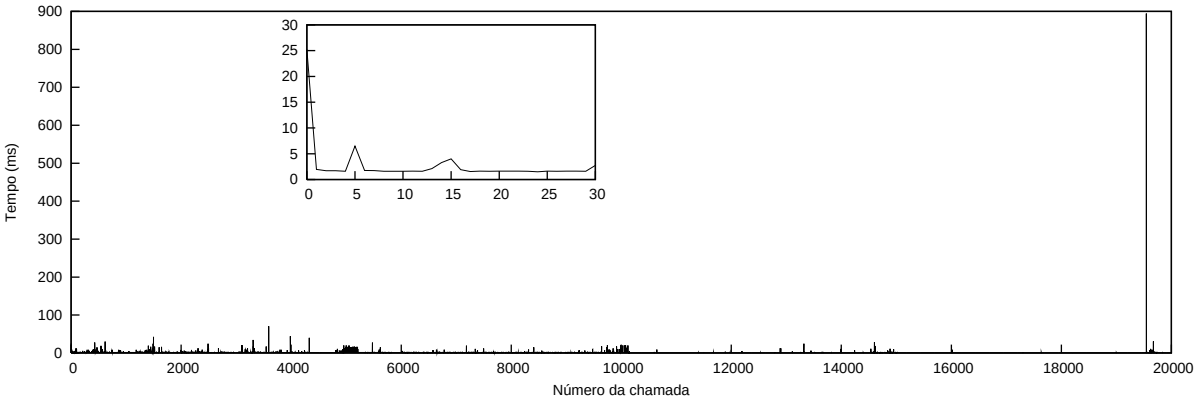
Cabe ressaltar aqui que apenas as mensagens de requisição são acrescidas do contexto transacional. Outro ponto importante a ser ressaltado é que os dados exibidos na Figura 5.5a não levam em consideração os períodos de criação e encerramento da transação. Assim sendo, os tempos exibidos nessa figura representam apenas a duração de uma chamada remota que propaga o contexto transacional.

A Figura 5.5c mostra que os contextos transacionais possuem, num determinado protocolo de comunicação, um tamanho fixo. Em particular, nos nossos experimentos um contexto transacional *JBoss Remoting* apresentou 1052 bytes. Já um contexto *IIOP* apresentou 760 bytes, valor muito próximo ao tamanho do contexto *SOAP*, que foi de 735 bytes.

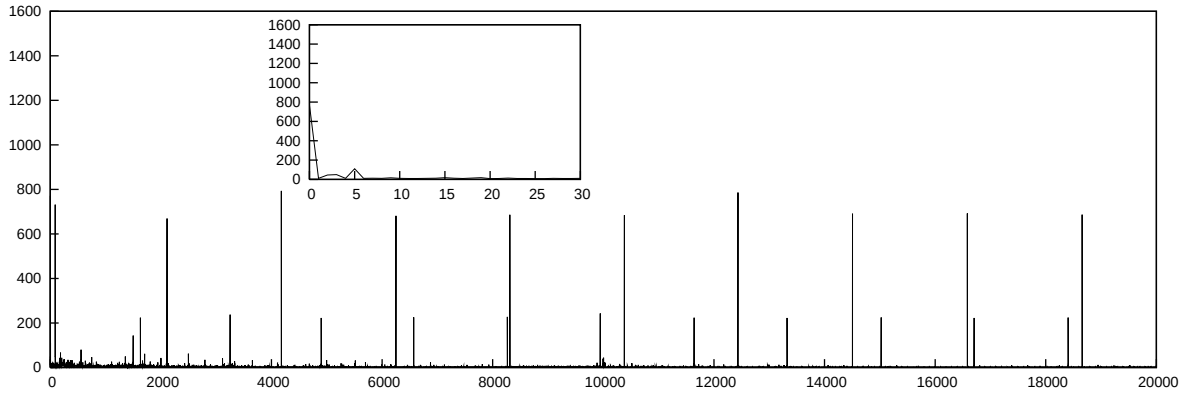
É importante frisar que, sob a perspectiva de um determinado protocolo de comunicação, o custo da propagação do contexto transacional é praticamente fixo. Ele abrange basicamente




(a) Duração de cada uma das 20000 chamadas *JBoss Remoting* à operação `getString100`.

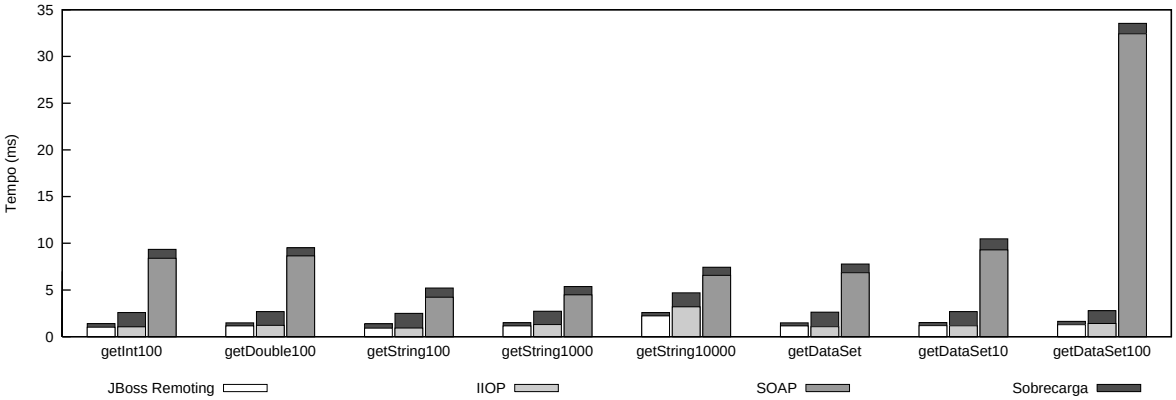



(b) Duração de cada uma das 20000 chamadas *IIOP* à operação `getString100`.




(c) Duração de cada uma das 20000 chamadas *SOAP* à operação `getString100`.

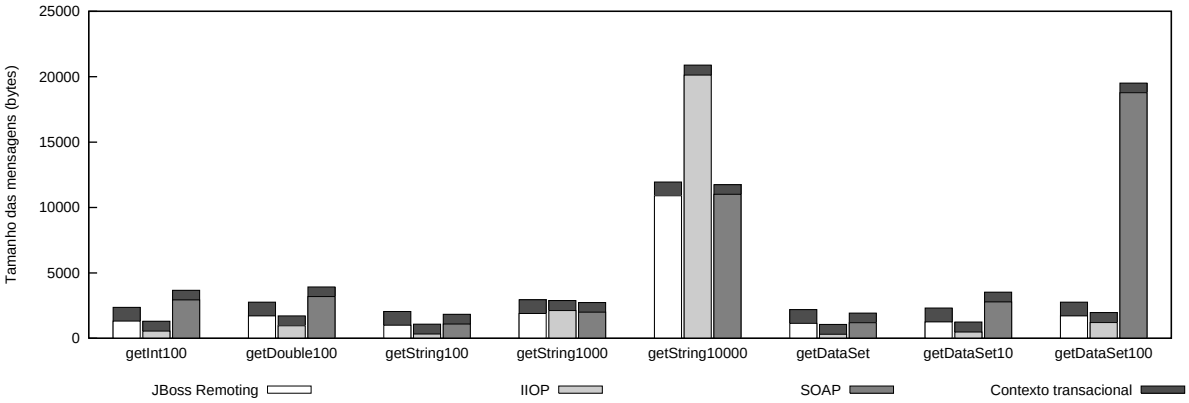
Figura 5.4: As durações das 20000 chamadas remotas à operação `getString100`. O componente *EJB* estava configurado com o atributo transacional `Never`.




(a) A sobrecarga imposta pela propagação do contexto transacional.

| | | int100 | double100 | string100 | string1000 | string10000 | dataSet | dataSet10 | dataSet100 |
|---|---|---|---|---|---|---|---|---|---|
| JBoss Remoting | Média | 1,40 | 1,49 | 1,38 | 1,50 | 2,58 | 1,49 | 1,51 | 1,64 |
| | Desvio padrão | 5,94 | 5,90 | 5,92 | 6,19 | 10,03 | 6,24 | 6,26 | 6,16 |
| | Mínimo | 1,22 | 1,23 | 1,10 | 1,22 | 2,23 | 1,12 | 1,24 | 1,36 |
| | Máximo | 720,02 | 719,99 | 722,38 | 729,49 | 772,19 | 724,13 | 727,39 | 712,29 |
| | I.C. (95%) | [1,30; 1,49] | [1,39; 1,58] | [1,28; 1,47] | [1,40; 1,60] | [2,42; 2,74] | [1,39; 1,59] | [1,41; 1,61] | [1,54; 1,74] |
| IIOP | Média | 2,58 | 2,69 | 2,50 | 2,73 | 4,70 | 2,63 | 2,68 | 2,78 |
| | Desvio padrão | 18,77 | 18,13 | 18,04 | 17,50 | 17,83 | 18,32 | 18,45 | 17,79 |
| | Mínimo | 1,97 | 2,11 | 1,94 | 2,22 | 4,10 | 2,06 | 2,08 | 2,20 |
| | Máximo | 1476,40 | 1450,62 | 1456,36 | 1418,37 | 1445,47 | 1476,61 | 1475,62 | 1468,90 |
| | I.C. (95%) | [2,28; 2,88] | [2,40; 2,98] | [2,21; 2,79] | [2,45; 3,01] | [4,42; 4,99] | [2,34; 2,92] | [2,39; 2,98] | [2,50; 3,07] |
| SOAP | Média | 9,35 | 9,52 | 5,21 | 5,36 | 7,44 | 7,78 | 10,48 | 33,55 |
| | Desvio padrão | 21,94 | 22,67 | 18,42 | 19,68 | 25,46 | 19,98 | 22,03 | 38,57 |
| | Mínimo | 7,97 | 8,12 | 4,14 | 4,31 | 5,82 | 6,59 | 8,84 | 30,05 |
| | Máximo | 729,54 | 730,87 | 749,79 | 750,60 | 676,12 | 781,25 | 758,60 | 815,33 |
| | I.C. (95%) | [9,00; 9,70] | [9,15; 9,88] | [4,91; 5,50] | [5,05; 5,68] | [7,03; 7,85] | [7,46; 8,10] | [10,13; 10,83] | [32,94; 34,17] |

(b) Estatísticas sobre as durações das chamadas transacionais. Todos os tempos são expressos em milissegundos.




(c) Tamanhos das mensagens trocadas entre o cliente e o servidor.

Figura 5.5: Os resultados da execução das chamadas remotas a cada uma das operações da interface `RoundTripTime`. Tais chamadas ocorreram dentro do escopo de uma transação.

três componentes: *(i)* a injeção do contexto transacional na mensagem de requisição, *(ii)* o transporte do contexto transacional como parte dessa mensagem e *(iii)* a extração e a importação do contexto pelo gerenciador de transações. O procedimento descrito no item *(i)* é efetuado no lado cliente, enquanto que o item *(iii)* é executado no lado servidor. Dado que os nossos experimentos foram realizados numa rede local, o custo adicional gerado pelo transporte do contexto transacional foi relativamente baixo. A maior parte do incremento nos tempos de execução foi ocasionada pela injeção, extração e importação do contexto transacional.

A Figura 5.5a mostra que a propagação do contexto transacional gerou um aumento significativo no tempo de execução das chamadas remotas. A mesma figura também mostra que, num dado protocolo de comunicação, a sobrecarga gerada pela propagação do contexto transacional é praticamente constante, isto é, ela independe da operação chamada. Isso ocorre porque o processo de injeção, transporte e extração do contexto transacional é idêntico para todas as chamadas que empregam o mesmo protocolo de comunicação. No caso do *JBoss Remoting*, a propagação do contexto transacional gerou uma sobrecarga de aproximadamente $0,4$ milissegundos por chamada remota. No caso das chamadas *IIOP*, o aumento foi de cerca de $1,5$ milissegundos por chamada, sendo que para o protocolo *SOAP* esse valor foi de aproximadamente $1,0$ milissegundos.

Cabe destacar que, devido à elevada sobrecarga transacional no caso do *IIOP*, o *JBoss Remoting* mostrou-se mais eficiente do que este em todas as chamadas transacionais efetuadas nos nossos experimentos. O *IIOP* apresentou uma sobrecarga transacional elevada devido às referências *CORBA* (para o `Terminator` e para o `Coordinator`) presentes no contexto transacional. Nossos experimentos mostraram que, no *ORB* que empregamos (o *JacORB*), o empacotamento e o desempacotamento de referências *CORBA* são operações lentas.

As Figuras 5.6a, 5.6b e 5.6c apresentam a duração de cada uma das 20000 chamadas transacionais efetuadas sobre cada protocolo (*JBoss Remoting*, *IIOP* e *SOAP*) à operação `getString100`. Na primeira figura, há apenas duas perturbações significativas. Ambas são causadas por coletas completas de lixo. Na segunda figura, temos perturbações mais salientes, causadas por coletas completas de lixo, e perturbações menos salientes, causadas por coletas parciais de lixo. Um ponto a ser ressaltado é a cadência de tais perturbações. Em outras palavras, as perturbações ocorrem após um número praticamente constante de chamadas. Isso ocorre porque as chamadas remotas são efetuadas repetidamente. Como as chamadas são sempre as mesmas num dado experimento, é esperado que o "lixo" seja produzido a uma taxa praticamente constante. Na terceira figura, temos um cenário bastante parecido com o da segunda figura: perturbações maiores e menores, ambas periódicas. Nos dados exibidos nas Figuras 5.5a e 5.5b, foram incluídas as perturbações geradas pelo coletor de lixo. A exemplo

do que foi feito nas seções anteriores, não excluímos essas perturbações pois elas fazem parte da execução normal de um programa.

Ainda nas Figuras 5.6a, 5.6b e 5.6c, é possível notar que as tomadas de tempo das 30 primeiras chamadas remotas estão destacadas. Dado que antes da execução de cada experimento os servidores de aplicações envolvidos foram reiniciados, o tempo da primeira chamada remota é consideravelmente superior aos tempos das chamadas subseqüentes.

### 5.1.5 Estimativa da sobrecarga imposta pelo uso de um serviço transacional

Esta seção apresenta os resultados de mais dois conjuntos de experimentos. No primeiro desses conjuntos, medimos qual o tempo necessário para iniciar e efetivar uma transação. No segundo, avaliamos o impacto causado pelo uso de um serviço transacional no desempenho de uma aplicação distribuída que envolve dois recursos transacionais *XA*.

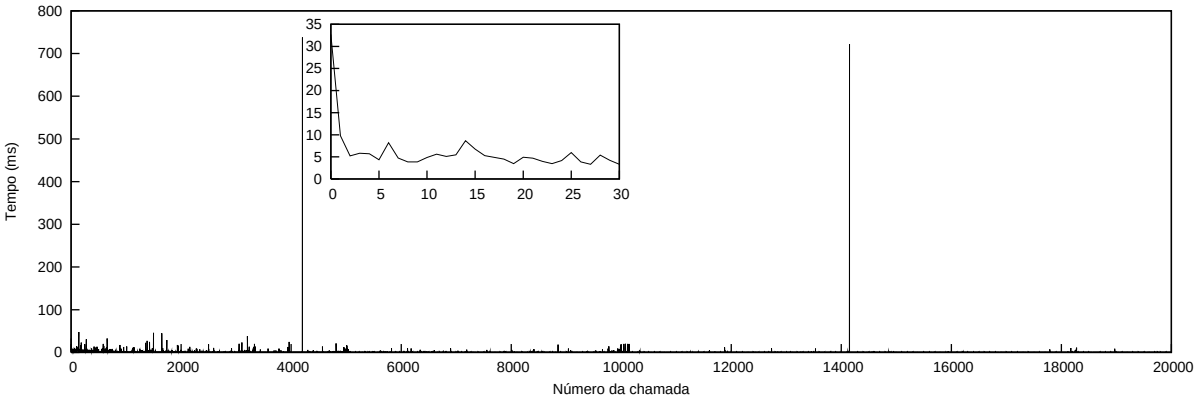
#### 5.1.5.1 Criação e efetivação de transações

Nosso primeiro conjunto de experimentos tinha como objetivo estimar o intervalo de tempo necessário para criar e efetivar uma transação. Nos experimentos desse conjunto, as mensagens para a criação e a efetivação da transação foram enviadas ao servidor de aplicações utilizando cada um dos três protocolos de comunicação mencionados nesse capítulo: *JBoss Remoting*, *IIOP* e *SOAP*. Isso nos permitiu estimar a sobrecarga imposta pelo uso de um determinado protocolo de comunicação na criação e efetivação de transações.

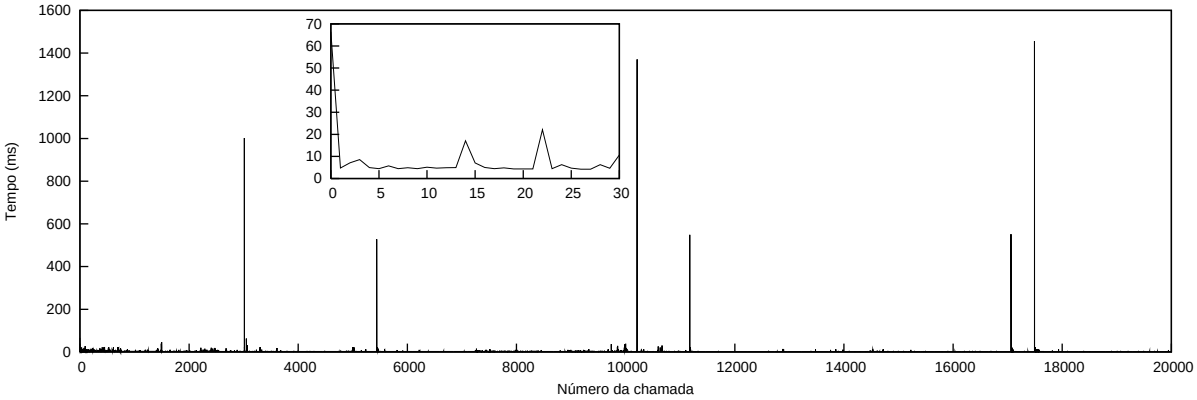
O código utilizado no cliente para criar e encerrar uma transação foi semelhante ao exibido na Figura 5.7. Cabe ressaltar que, nessa figura, a referência `userTransaction` está vinculada a uma instância de `javax.transaction.UserTransacion`. A implementação de `UserTransaction` utilizada pelo cliente variou de acordo com o protocolo empregado na comunicação com o servidor.

Medir o tempo necessário para iniciar e efetivar uma transação é importante porque essas são etapas essenciais de uma transação, já que toda transação não abortada inclui em seu tempo de execução o período gasto na sua criação e na sua efetivação. Cabe ressaltar que, nos experimentos executados, as transações foram criadas e imediatamente efetivadas. Por esse motivo, elas não envolveram nenhum recurso transacional, o que tornou o processo de efetivação bastante rápido.

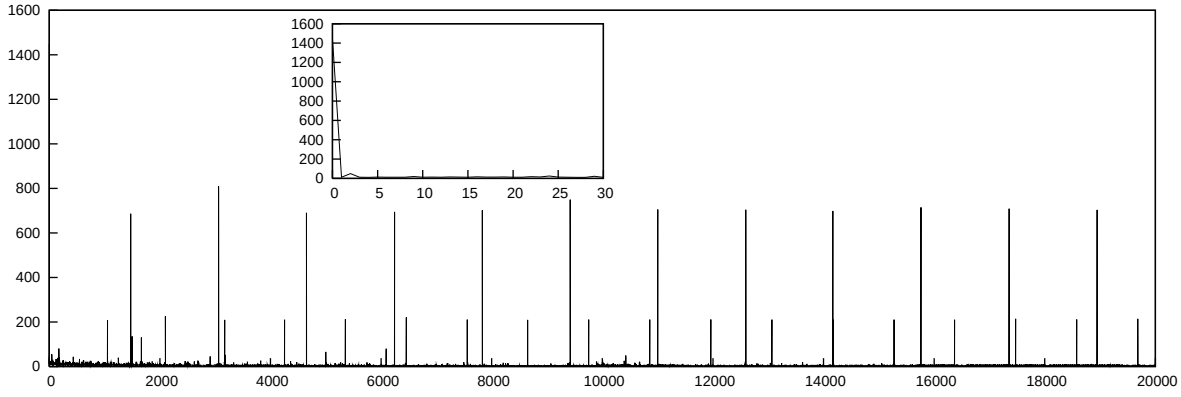
Tivemos ainda uma outra motivação para mensurar os tempos de criação e efetivação de transações. Tais medições nos possibilitariam avaliar o desempenho de nossas implementações dos *port types* definidos em *WS-Coordination* e *WS-AtomicTransaction*. Como discutido na




(a) Duração de cada uma das 20000 chamadas *JBoss Remoting* à operação `getString100`.




(b) Duração de cada uma das 20000 chamadas *IIOP* à operação `getString100`.



(c) Duração de cada uma das 20000 chamadas *SOAP* à operação `getString100`.

Figura 5.6: As durações das 20000 chamadas remotas à operação `getString100`. Tais chamadas ocorreram dentro do escopo de uma transação.

```
userTransaction.begin();
userTransaction.commit();
```

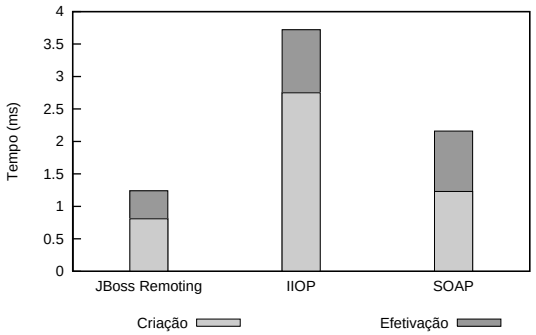
Figura 5.7: O código utilizado no cliente para criar e efetivar uma transação.

Seção 4.4, nossa implementação inicial desses *port types*, baseada na *JAX-RPC*, apresentou problemas de desempenho e foi substituída por uma implementação mais eficiente, baseada em *servlets* e *StAX*. Ao executar os experimentos, portanto, esperávamos que dessa vez o desempenho do protocolo *SOAP* ficasse mais próximo dos desempenhos exibidos pelo *JBoss Remoting* e pelo *IIOP*.

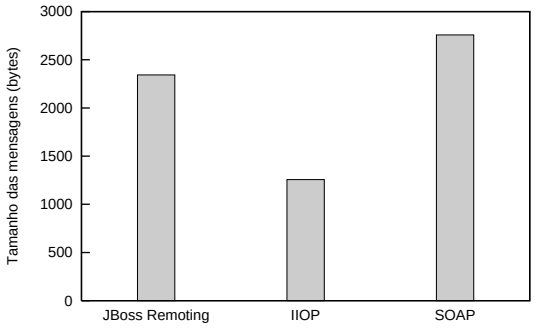
Os resultados dos experimentos são exibidos na Figura 5.8. Cada barra da Figura 5.8a está dividida em duas partes. A porção inferior de uma barra representa a duração média da chamada remota responsável por criar uma nova transação. Já a parte superior exibe a duração média da chamada remota responsável por efetivar a transação. A Figura 5.8b mostra os tamanhos, em bytes, das mensagens trocadas entre o cliente e o servidor. Por fim, a Figura 5.8c mostra as estatísticas sobre as tomadas de tempo.

Conforme mostra a Figura 5.8a, mais uma vez o *JBoss Remoting* apresentou o melhor desempenho entre os três. A vantagem do *JBoss Remoting* pode ser explicada novamente pelo emprego de um mecanismo de seriação (*JBoss Serialization*) mais eficiente que os utilizados pelos outros protocolos. Surpreendentemente, a Figura 5.8a também mostra que, nesse conjunto de experimentos, a comunicação via *SOAP* foi mais rápida do que a via *IIOP*. Isso se deveu a duas razões: *(i)* a criação de um contexto transacional *CORBA* foi, no *ORB* que empregamos, um procedimento bastante lento (devido a presença de referências *CORBA* no contexto) e *(ii)* os *port types* que atendem as requisições *SOAP* foram implementados de maneira bastante eficiente.

Conforme a Figura 5.8b mostra, a efetivação de uma transação consome aproximadamente o mesmo intervalo de tempo no caso do *IIOP* ($0,97$ milissegundos) e do *SOAP* ($0,93$ milissegundos). Contudo, nossos experimentos também mostraram que a criação de uma transação via *IIOP* é bem mais lenta ($2,75$ milissegundos) do que a criação via *SOAP* ($1,23$ milissegundos). Por esse motivo, o intervalo de tempo necessário para a criação e a efetivação de uma transação é maior no caso do *IIOP* do que no caso do *SOAP*. Um exame detalhado do processo de criação de transações via *IIOP* mostrou que o gargalo de desempenho está na criação das referências *CORBA* presentes no contexto transacional. Mais precisamente, se considerarmos o tempo gasto no servidor para atender uma requisição *IIOP* que solicita a criação de uma transação, cerca de 80% dele é consumido pelo método `org.jacorb.poa.POA.create_reference_with_id`, utilizado para criar referências *CORBA*.




(a) Duração média da criação e efetivação de uma transação.




(b) Tamanhos das mensagens trocadas entre o cliente e o servidor.

| | | Criação | Efetivação | Total |
|---|---|---|---|---|
| JBoss Remoting | Média | 0,81 | 0,43 | 1,23 |
| | Desvio padrão | 1,43 | 0,21 | 1,45 |
| | Mínimo | 0,73 | 0,34 | 1,10 |
| | Máximo | 174,07 | 19,35 | 174,66 |
| | I.C. (95%) | [0,78; 0,83] | [0,42; 0,43] | [1,21; 1,25] |
| IIOP | Média | 2,75 | 0,97 | 3,72 |
| | Desvio padrão | 18,60 | 0,75 | 18,63 |
| | Mínimo | 2,35 | 0,82 | 3,22 |
| | Máximo | 1520,85 | 53,31 | 1522,01 |
| | I.C. (95%) | [2,46; 3,05] | [0,96; 0,98] | [3,42; 4,02] |
| SOAP | Média | 1,23 | 0,93 | 2,16 |
| | Desvio padrão | 7,52 | 5,79 | 9,49 |
| | Mínimo | 1,01 | 0,81 | 1,86 |
| | Máximo | 765,63 | 659,65 | 766,57 |
| | I.C. (95%) | [1,11; 1,35] | [0,83; 1,02] | [2,01; 2,31] |

(c) Estatísticas sobre a criação e efetivação de transações. Todos os tempos são expressos em milissegundos.

Figura 5.8: Os resultados dos experimentos que envolveram a criação e a efetivação de transações.

A utilização de *servlets* e da *StAX* na implementação dos *port types* que atendem as requisições *SOAP* é a razão do bom desempenho apresentado por esse protocolo. Cabe aqui uma discussão sucinta sobre os fatores que levaram a esse bom desempenho: um *servlet* recebe uma requisição *SOAP*, que é então processada utilizando a *StAX*. Esse processamento ocorre por meio de um código bastante específico e otimizado. Em outras palavras, o código que trata as requisições destinadas aos *port types* definidos em *WS-Coordination* e *WS-AtomicTransaction* é capaz de processar, utilizando a *StAX*, apenas as mensagens definidas nessas especificações. Esse é um dos motivos pelos quais a implementação dos *port types* é tão eficiente. Ademais, as mensagens *SOAP* destinadas a esses *port types* não são completamente processadas. O analisador *XML* deixa de processar uma mensagem tão logo extraia dela todas as informações relevantes. No caso de uma mensagem `Prepared`, por exemplo, o analisador pára de processar a mensagem *SOAP* assim que ele encontra o elemento `Prepared`. Se esse elemento está presente numa mensagem *SOAP*, já é possível saber que se trata de uma mensagem `Prepared` e,

portanto, o restante da mensagem não é analisado.

A geração de mensagens de resposta pelos *port types* também é efetuada de modo bastante eficiente. Para toda mensagem *SOAP* definida em *WS-Coordination* e *WS-AtomicTransaction*, desenvolvemos uma classe Java que a representa. Essa classe armazena uma espécie de molde da mensagem *SOAP* representada. Durante a construção de uma mensagem *SOAP*, esse molde é preenchido com os valores presentes nas variáveis de instância da classe. Como o molde mencionado é constituído por uma série de cadeias de caracteres, a construção de uma mensagem *SOAP* fica reduzida à concatenação de cadeias de caracteres pré-definidas (que compõem o molde) com cadeias geradas dinamicamente (que representam os valores das variáveis de instância da classe).

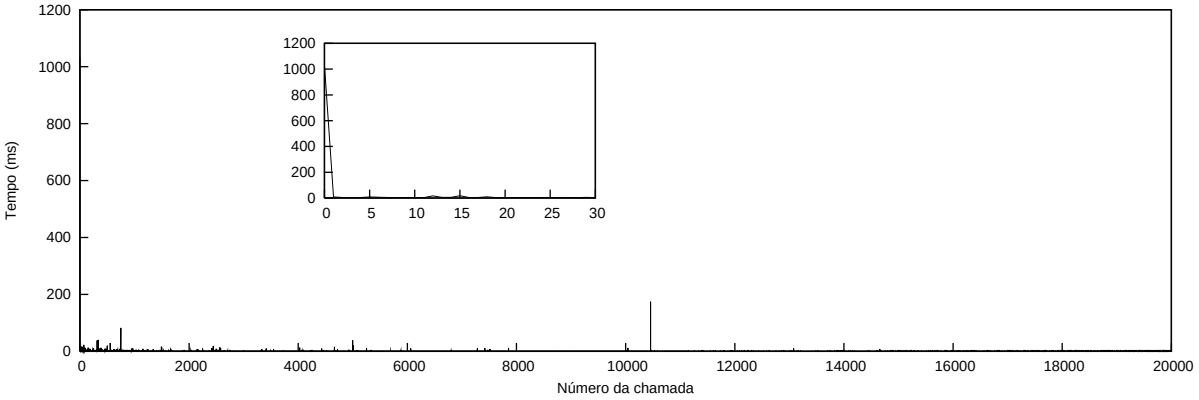
As Figuras 5.9a, 5.9b e 5.9c apresentam a duração das 20000 chamadas destinadas a criação e efetivação de transações. Na primeira figura, há apenas uma perturbação significativa, causada por uma coleta parcial de lixo. Na segunda figura, temos perturbações mais salientes, causadas por coletas completas de lixo, e perturbações menos salientes, causadas por coletas parciais de lixo. Na terceira figura, temos um cenário análogo ao da segunda: perturbações maiores e menores, ambas periódicas. Nos dados exibidos nas Figuras 5.8a e 5.8c, foram incluídas as perturbações geradas pelo coletor de lixo. Como nas seções anteriores, não excluímos essas perturbações pois elas fazem parte da execução normal de um programa.

#### 5.1.5.2 Transações envolvendo recursos transacionais

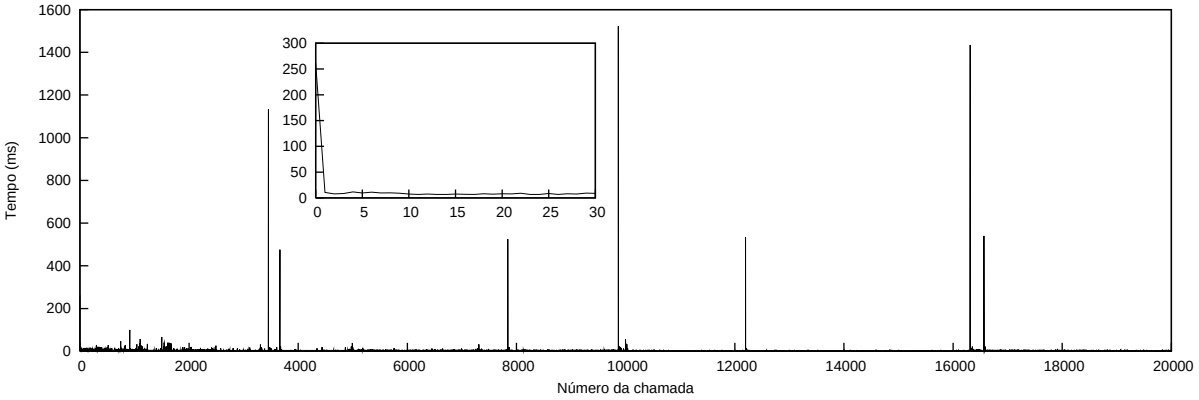
Os experimentos descritos até agora não envolveram nenhum recurso transacional. Devido a isso, em tais experimentos o processo de efetivação de uma transação foi bastante rápido, pois ele não exigiu a execução do protocolo *2PC*. Para estimar a sobrecarga total imposta pelo uso de um serviço transacional, é necessário que ocorra a execução do *2PC* como parte do processo de efetivação de uma transação. Tendo isso em vista, desenvolvemos uma aplicação distribuída bastante simples. Ela é composta por um cliente *stand-alone* e dois componentes *EJB* implantados em instâncias distintas de servidores de aplicações. Cada um desses componentes armazena seus dados em instâncias distintas de um sistema gerenciador de banco de dados (SGBD)[4]. Há também um terceiro servidor de aplicações, que atua como o coordenador da transação distribuída. Esse cenário é ilustrado na Figura 5.10.

O cliente modifica o estado dos dois componentes *EJB*, sendo que ele faz isso utilizando exclusivamente cada um dos três protocolos de comunicação abordados neste capítulo (*JBoss*

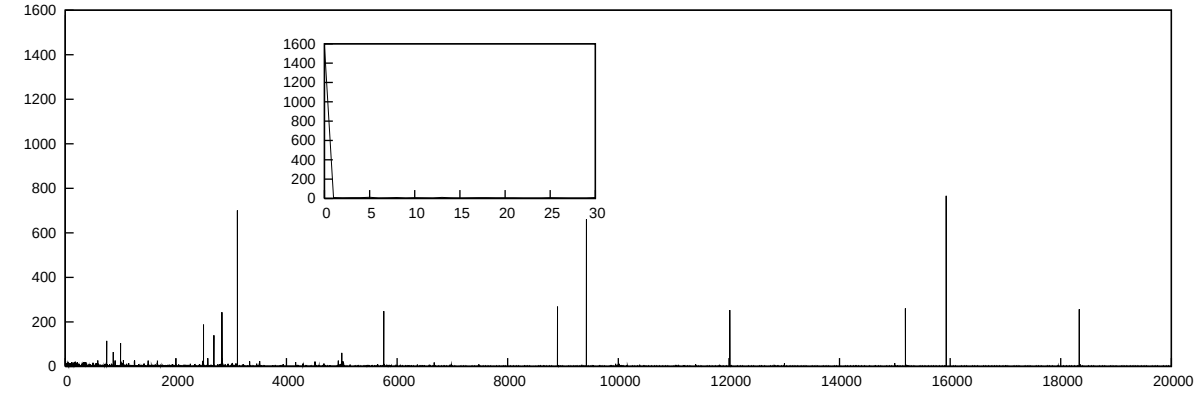
[4]Cada uma das instâncias do SGBD é executada no mesmo processo que um servidor de aplicações, possibilitando uma maior eficiência na utilização do SGBD.




(a) As durações das 20000 chamadas *JBoss Remoting* destinadas à criação e efetivação de transações.




(b) As durações das 20000 chamadas *IIOP* destinadas à criação e efetivação de transações.




(c) As durações das 20000 chamadas *SOAP* destinadas à criação e efetivação de transações.

Figura 5.9: As durações das 20000 chamadas destinadas à criação e efetivação de transações.

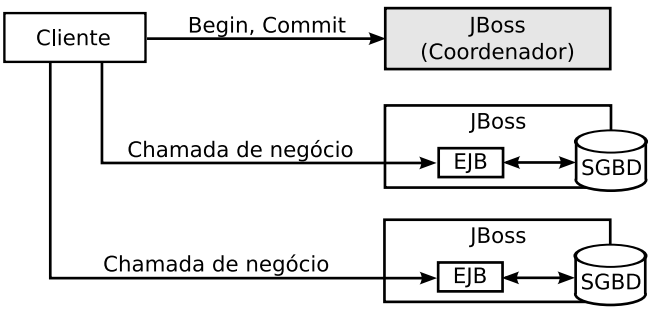



Figura 5.10: O cenário utilizado para estimar a sobrecarga total imposta pelo uso de um serviço transacional.

*Remoting*, *IIOP* e *SOAP*). As modificações dos estados dos componentes ocorrem de três modos: *(i)* através de chamadas remotas que são efetuadas fora do escopo de uma transação, *(ii)* através de chamadas remotas feitas dentro do escopo de uma transação, mas com o *log* transacional desativado, e *(iii)* através de chamadas remotas efetuadas dentro do escopo de uma transação, com o *log* transacional ativado. Com isso, foi possível comparar os tempos de execução do experimento sem e com o emprego de transações. Além disso, a desativação do *log* transacional no cenário *(ii)* nos permitiu estimar a sobrecarga gerada pelo emprego deste.

Em cada um dos três cenários apresentados, foram executadas um total de 8000 chamadas remotas. As durações das 2000 primeiras chamadas foram desconsideradas (período de aquecimento) e as tomadas de tempo restantes foram utilizadas no cálculo da média e de outras estatísticas sobre o experimento.

Nos dois cenários em que ocorre a execução do *2PC* (*ii* e *iii*), o protocolo utilizado para efetuar as chamadas remotas é o mesmo empregado pelo coordenador durante a execução do *2PC*. Desse modo, se os componentes forem chamados utilizando o protocolo *SOAP*, por exemplo, então as mensagens do *2PC* serão enviadas utilizando também o protocolo *SOAP*. No caso em que as chamadas remotas ocorrem fora do escopo de uma transação, não há execução do *2PC*.

A interface implementada pelos componentes *EJB* é exibida na Figura 5.11. Cada componente *EJB* representa uma lista de contatos bastante simplificada, à qual um cliente pode adicionar novos contatos. Como os dados são armazenados num SGBD, a adição de um novo contato faz com que um recurso transacional remoto seja registrado com o coordenador da transação distribuída (desde que a chamada remota seja efetuada dentro do escopo de uma transação e que o recurso transacional remoto não tenha sido previamente registrado com o coordenador). Cabe ressaltar que cada componente *EJB* utiliza a sua própria instância de

SGBD. Logo, como o cliente utiliza dois componentes *EJB* implantados em instâncias distintas de servidores de aplicações, dois recursos transacionais remotos serão registrados com o coordenador da transação.

```
public interface ContactList extends javax.ejb.EJBObject
{
   void addContactEntry(String name, String email) throws RemoteException;
}
```

Figura 5.11: A interface `ContactList`, cuja operação é implementada pelo componente *EJB* utilizado no experimento.

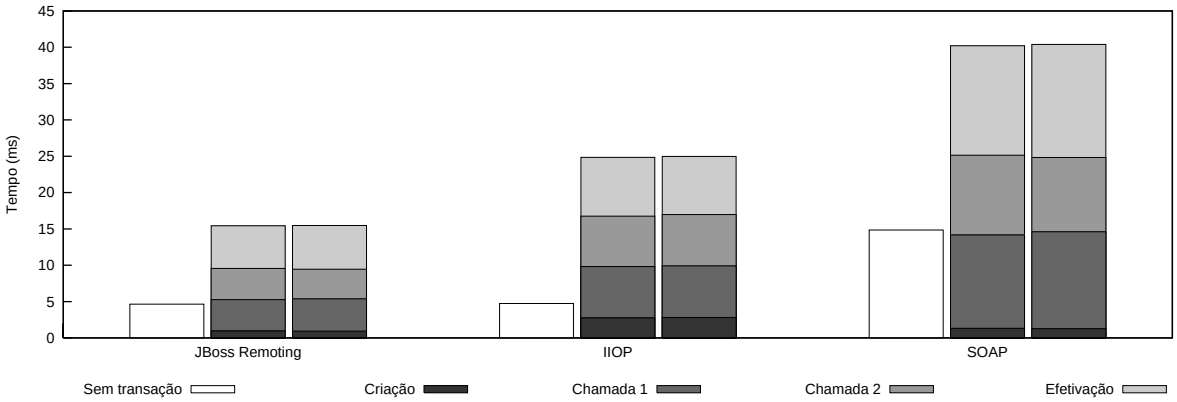
A Figura 5.12 mostra uma versão simplificada do cliente que efetua chamadas remotas transacionais. As implementações de `UserTransaction` utilizadas, bem como os representantes locais dos *EJBs*, variaram de acordo com o protocolo de comunicação empregado no experimento.

```
userTransaction.begin();
   beanNoPrimeiroServidor.addContactEntry("Nome1", "E-mail1");
   beanNoSegundoServidor.addContactEntry("Nome1", "E-mail1");
userTransaction.commit();
```

Figura 5.12: O código utilizado no cliente para alterar o estado dos componentes *EJB*.

Os resultados do experimento são exibidos na Figura 5.13. Para cada protocolo, há um conjunto de três barras na Figura 5.13a. A barra branca representa o tempo de execução de duas chamadas remotas à operação `addContactEntry`, efetuadas fora do escopo de uma transação. As outras duas barras representam o tempo de execução de uma transação distribuída que engloba duas chamadas remotas à operação `addContactEntry`. A barra central representa o tempo médio de execução da transação quando o *log* transacional do servidor de aplicações está desabilitado. A terceira barra, por sua vez, representa o tempo médio de execução da transação com o *log* transacional habilitado. Cabe ressaltar que, na segunda e na terceira barras, o tempo total foi dividido em quatro partes: criação da transação, chamada transacional ao primeiro componente, chamada transacional ao segundo componente e efetivação da transação.

Como a Figura 5.13a mostra, mais uma vez o *JBoss Remoting* apresentou um desempenho superior. O *IIOP* exibiu uma sobrecarga transacional bastante elevada, mas mesmo assim ele apresentou um desempenho consideravelmente melhor que o do *SOAP*. Este último foi novamente o mais lento de todos. Como é possível observar, os tempos exibidos nesse experimento




(a) Estimativa da sobrecarga total imposta pelo uso de um serviço transacional.

| | | Criação | Chamada 1 | Chamada 2 | Efetivação | Total |
|---|---|---|---|---|---|---|
| JBoss Remoting | Média | 0,94 | 4,43 | 4,08 | 6,03 | 15,49 |
| | Desvio padrão | 0,33 | 15,11 | 1,35 | 18,52 | 24,11 |
| | Mínimo | 0,82 | 3,49 | 3,36 | 4,36 | 12,48 |
| | Máximo | 17,49 | 939,25 | 31,98 | 936,74 | 957,20 |
| | I.C. (95%) | [0,93; 0,95] | [3,97; 4,90] | [4,04; 4,12] | [5,46; 6,60] | [14,74; 16,23] |
| IIOP | Média | 2,80 | 7,11 | 7,07 | 7,99 | 24,97 |
| | Desvio padrão | 9,16 | 24,42 | 25,89 | 27,23 | 45,81 |
| | Mínimo | 2,54 | 5,89 | 5,85 | 5,76 | 20,10 |
| | Máximo | 581,20 | 1547,33 | 1538,03 | 1529,56 | 1564,60 |
| | I.C. (95%) | [2,52; 3,09] | [6,35; 7,87] | [6,27; 7,87] | [7,15; 8,83] | [23,55; 26,39] |
| SOAP | Média | 1,27 | 13,34 | 10,22 | 15,57 | 40,40 |
| | Desvio padrão | 0,15 | 28,88 | 27,13 | 43,38 | 59,57 |
| | Mínimo | 1,11 | 8,56 | 8,59 | 10,93 | 29,46 |
| | Máximo | 8,66 | 864,16 | 1417,94 | 830,86 | 1472,66 |
| | I.C. (95%) | [1,27; 1,28] | [12,44; 14,23] | [9,38; 11,06] | [14,23; 16,92] | [38,55; 42,25] |

(b) Estatísticas sobre o tempo de execução de uma transação distribuída com o *log* transacional ativado.

Figura 5.13: Os resultados dos experimentos que visavam estimar a sobrecarga imposta por um serviço transacional.

foram maiores que os exibidos anteriormente. Isso se deveu a dois fatores. Primeiramente, esse experimento envolveu o registro de recursos transacionais remotos com o coordenador e a execução do protocolo *2PC*. Houve, portanto, uma série de trocas de mensagens[5] que não ocorreu nos experimentos anteriores. Em segundo lugar, os componentes *EJB* utilizados nesse experimento foram mais complexos, pois eles armazenavam seu estado num SGBD.

Ainda analisando a Figura 5.13a, é possível observar que o *log* transacional é responsável por uma fração não significativa do tempo total de execução da transação. Em particular, ele gera uma pequena sobrecarga apenas na etapa de efetivação da transação. Isso ocorre porque é nessa etapa que são gravados os registros `MULTI_TM_TX_COMMITTED` e `TX_PREPARED` nos *logs* locais do coordenador e dos participantes da transação.

[5]Mais precisamente, no caso transacional há duas mensagens de registro de participantes, duas mensagens `Prepare` e duas mensagens `Commit`. Nenhuma dessas mensagens é empregada no caso não transacional.

Ainda na Figura 5.13a, é possível observar que a sobrecarga total gerada pelo emprego de um serviço transacional foi elevadíssima. Por exemplo, no caso do *JBoss Remoting*, o uso de um serviço transacional elevou o tempo de execução do experimento em cerca de 233%. Para o *IIOP* e o *SOAP* esse aumento foi da ordem de 424% e 172%, respectivamente. Um serviço de transações só deve ser empregado, portanto, quando de fato houver a necessidade de atomicidade.

Cabe aqui uma ressalva em relação à sobrecarga imposta pelo uso de um serviço transacional nos nossos experimentos. O uso de SGBDs que executam nos mesmos processos dos servidores de aplicações possibilitou maior eficiência nas chamadas aos bancos de dados. A elevação do tempo de execução (expressa em percentual) provocada pelo serviço transacional foi, portanto, maior do que seria caso fossem empregados SGBDs remotos.

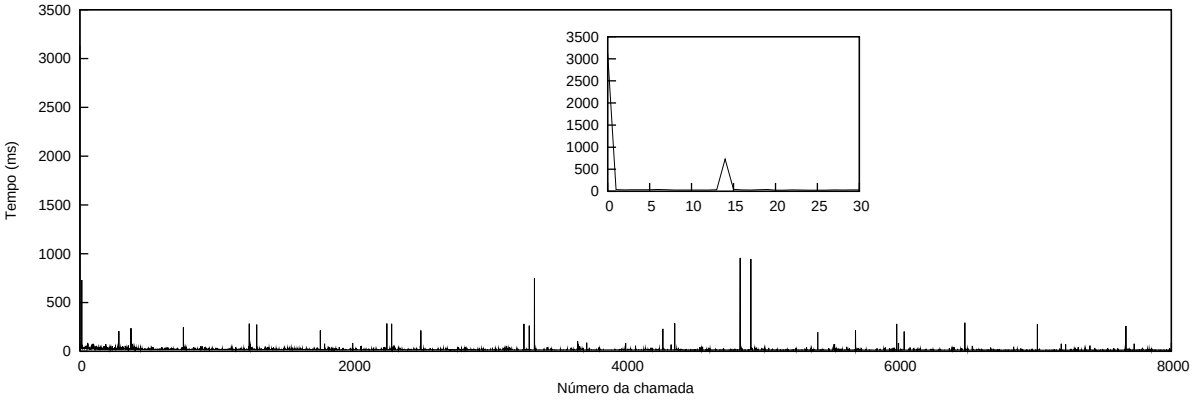
As Figuras 5.14a, 5.14b e 5.14c apresentam a duração de cada uma das 8000 chamadas remotas efetuadas pelos nossos experimentos sobre cada protocolo (*JBoss Remoting*, *IIOP* e *SOAP*). Os tempos se referem ao cenário em que o gerenciador de transações foi utilizado com o *log* transacional habilitado. Nas três figuras, as perturbações mais salientes correspondem a execuções de coletas de lixo completas. As perturbações médias (que duram cerca de 200 a 300 milissegundos) são causadas por coletas parciais de lixo. Nos dados exibidos nas Figuras 5.13a e 5.13b, foram incluídas as perturbações geradas pelo coletor de lixo. A exemplo das seções anteriores, não excluímos essas perturbações pois elas fazem parte da execução normal de um programa.

### 5.1.6 Escalabilidade

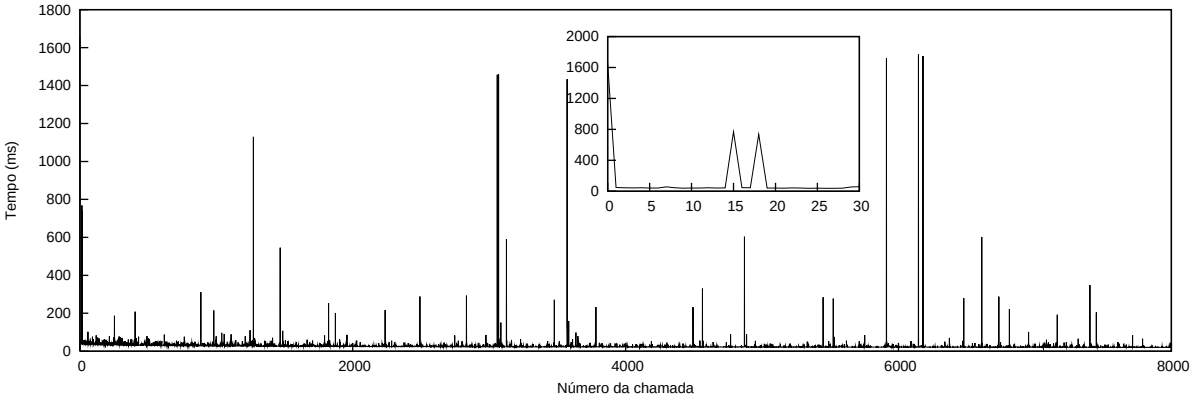
Além do desempenho, outra característica crítica dos sistemas gerenciadores de transações é a escalabilidade. Dado que esse tipo de sistema é freqüentemente submetido a um grande volume de requisições simultâneas, é importante avaliar como o seu desempenho varia de acordo com o número de clientes. Realizamos, portanto, experimentos que avaliavam o comportamento do gerenciador de transações em duas situações: *(i)* na criação e efetivação de transações via mensagens *SOAP* e *(ii)* na coordenação de transações distribuídas envolvendo recursos remotos acessíveis como *Web services*.

#### 5.1.6.1 Criação e efetivação de transações

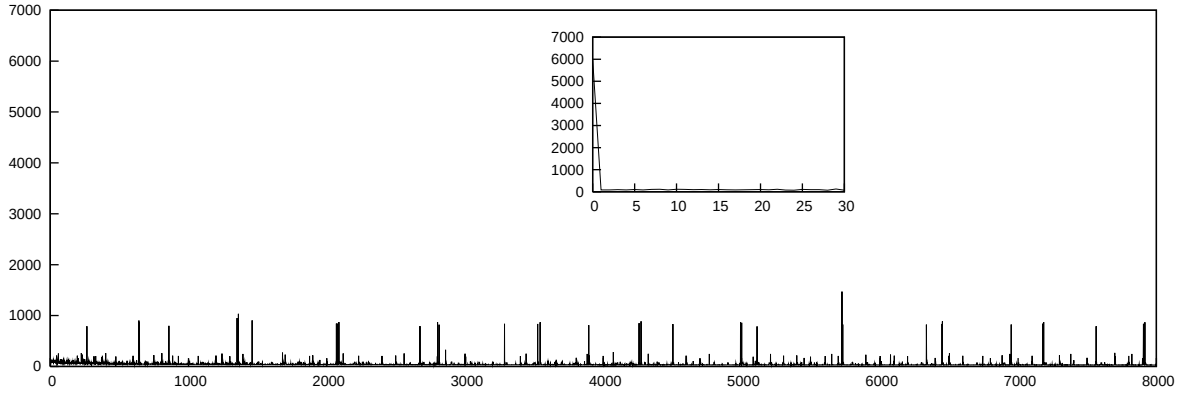
No primeiro experimento de escalabilidade, avaliamos como o gerenciador de transações se comportava quando múltiplos clientes criavam e efetivavam transações através de requisições *SOAP*. O código utilizado nos clientes para criar e encerrar as transações foi semelhante ao




(a) As durações das 8000 chamadas *JBoss Remoting* ao componente `ContactList`.



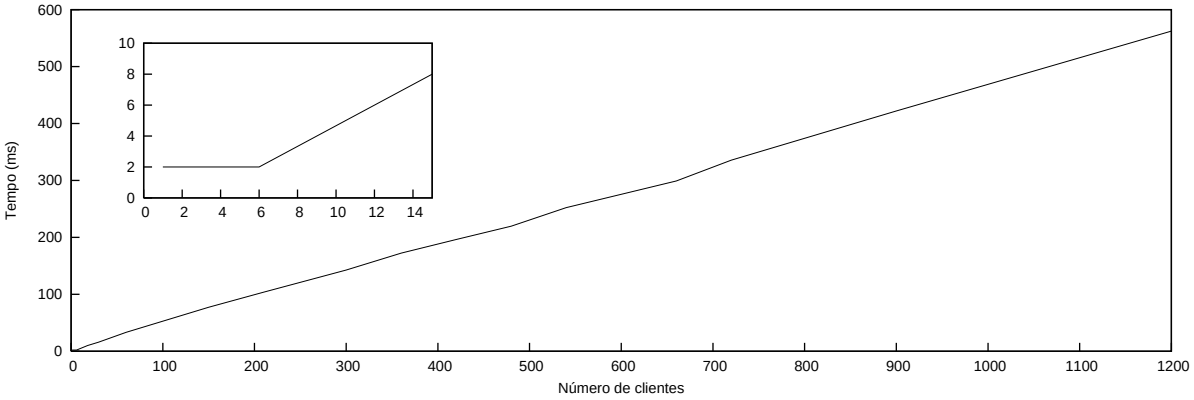

(b) As durações das 8000 chamadas *IIOP* ao componente `ContactList`.



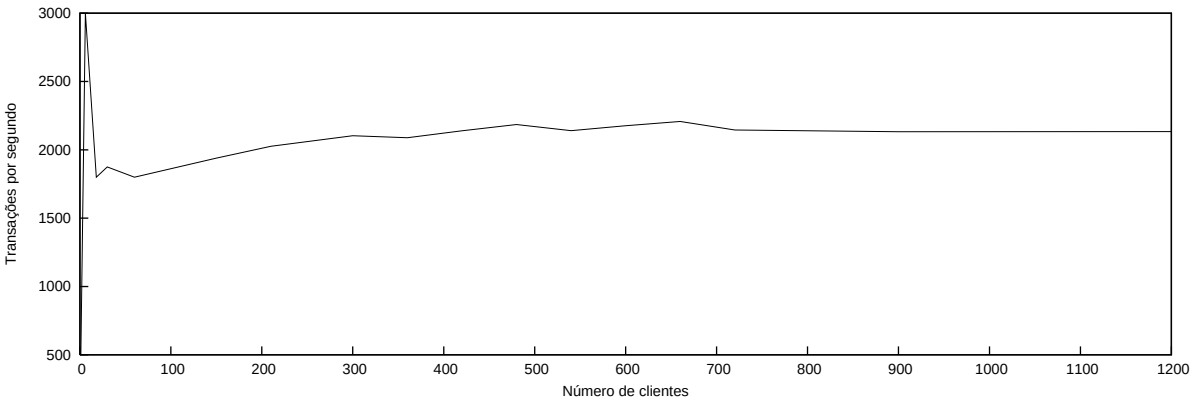

(c) As durações das 8000 chamadas *SOAP* ao componente `ContactList`.

Figura 5.14: As durações das 8000 chamadas remotas ao componente `ContactList`.

exibido na Figura 5.7. A realização desse experimento foi importante porque a criação e a efetivação de transações envolvem a execução de certos métodos sincronizados do gerenciador de transações. Em particular, a criação de uma transação implica na criação de um identificador e também de um *timeout*. Essas duas tarefas são efetuadas por métodos que executam trechos de código sincronizado (regiões críticas). Além disso, a criação e a efetivação de uma transação envolvem a adição e a remoção de entradas em tabelas associativas sincronizadas. Desse modo, um grande volume de requisições simultâneas ao gerenciador de transações poderia fazer com que a contenção nas regiões críticas degradasse o desempenho do gerenciador. Entretanto, os resultados do experimento (Figura 5.15) mostraram que isso não aconteceu.




(a) O tempo médio para a criação e a efetivação de uma transação através de mensagens *SOAP*.




(b) Número médio de transações criadas e efetivadas pelo gerenciador de transações no intervalo de um segundo.

Figura 5.15: O comportamento do gerenciador de transações de acordo com o número de clientes simultâneos. Cada cliente repetidamente criava uma transação e imediatamente a efetivava.

Como é possível observar na Figura 5.15a, o tempo necessário para se criar e efetivar uma transação permanece constante quando existem até seis[6] clientes simultâneos. A partir desse ponto, o tempo passa a crescer de forma praticamente linear com o número de clientes. Nos nossos experimentos, o tempo passou a aumentar quando o processador do servidor atingiu uma taxa de utilização de 100%. Em outras palavras, o processador do servidor foi, nos nossos experimentos, o fator limitante que provocou o aumento dos tempos medidos.

O crescimento linear pode ser justificado pelo fato que, de todo o tempo de processamento despendido no servidor para se criar e efetivar uma nova transação, apenas cerca de 1% dele é gasto em trechos de código sincronizado. A quase totalidade do tempo despendido no servidor é empregado na análise das mensagens *SOAP*, a qual não envolve regiões críticas.

Cabe ressaltar que, como esse experimento criou e imediatamente efetivou as transações, não houve o registro de nenhum recurso transacional e, portanto, não ocorreu a execução do protocolo *2PC*.

A Figura 5.15b mostra que o número médio de transações criadas e efetivadas por segundo permaneceu mais ou menos constante durante a execução do experimento. Essa uniformidade foi possível graças ao pouco tempo gasto pelo gerenciador de transações em operações sincronizadas. Como cerca de 99% do tempo de processamento despendido no servidor foi empregado em operações não sincronizadas, este pôde atender a um número praticamente constante de requisições por segundo, mesmo com o aumento no número de clientes concomitantes.

#### 5.1.6.2 Transações envolvendo recursos remotos acessíveis como Web services

Nosso segundo experimento de escalabilidade envolveu uma aplicação distribuída. Nessa aplicação, um cliente iniciava uma transação, efetuava chamadas para dois *Web services* implantados em instâncias distintas de servidores de aplicações e então efetivava a transação (Figura 5.16). Ao invés de armazenar seus estados em SGBDs, os *Web services* em questão fizeram o uso de recursos *XA* arremedados (*mocks*). O objetivo era que os recursos *XA* impusessem a menor sobrecarga possível sobre o tempo de execução da transação. Desse modo, os tempos medidos refletiriam melhor o desempenho do gerenciador de transações e não o desempenho dos recursos *XA* empregados. Se os *Web services* utilizassem um SGBD, por exemplo, uma parte considerável da duração da transação distribuída seria gasto nas operações executadas pelo SGBD. Um recurso *XA* arremedado, por outro lado, é bastante simples e eficiente, impondo uma sobrecarga mínima sobre o tempo de execução da transação distribuída.

[6] Embora pareça um número pequeno de clientes para saturar o serviço de transações, é bom lembrar que cada um deles tentava criar e efetivar o maior número possível de transações num dado intervalo de tempo. Sob condições normais de uso, onde transações são criadas mais esporadicamente, certamente o número de clientes necessário para saturar o serviço de transações seria maior.

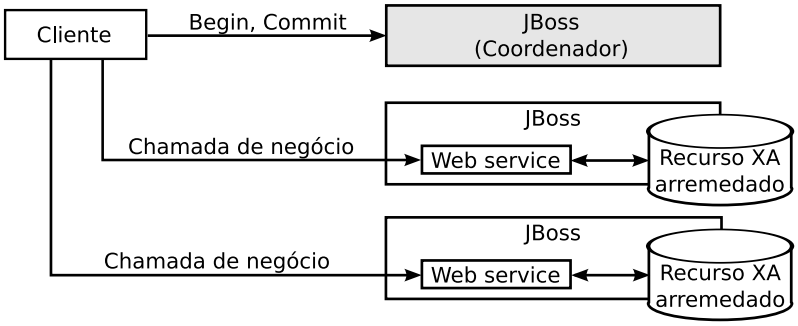


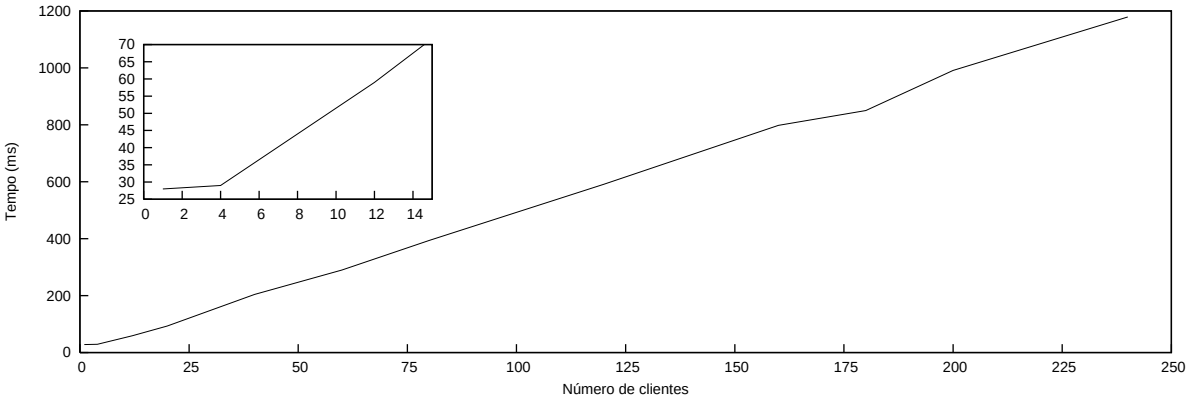

Figura 5.16: Na aplicação distribuída utilizada no experimento, o cliente chama as operações de dois *Web services* implantados em instâncias distintas de servidores de aplicações.

Os resultados da execução do experimento são exibidos na Figura 5.17. Cabe ressaltar que nesse experimento ocorreu o registro de dois recursos transacionais com o gerenciador de transações. Além disso, como parte do processo de efetivação da transação, houve a execução do protocolo *2PC*.

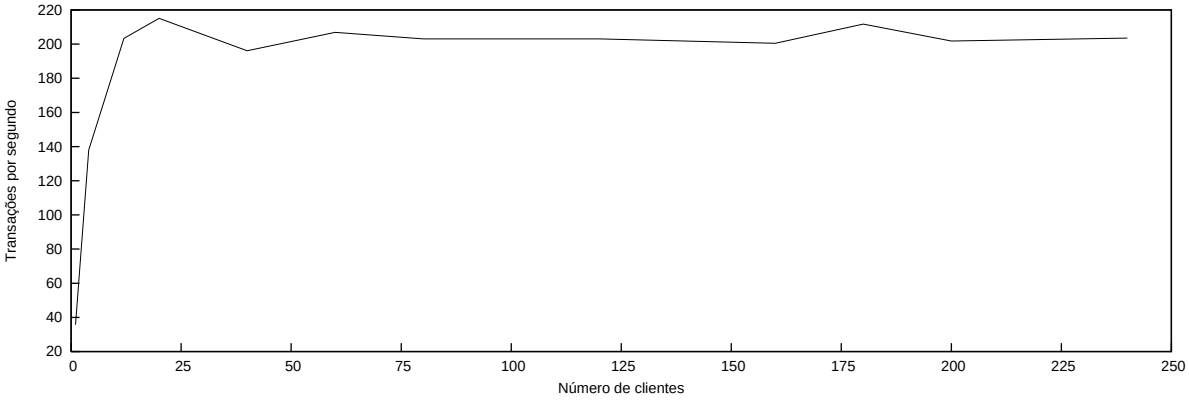
Como é possível notar na Figura 5.17a, o tempo de execução desse experimento permaneceu praticamente constante para até quatro clientes simultâneos. A partir desse ponto, o tempo de execução passou a crescer linearmente em função do número de clientes. Nos nossos experimentos, o tempo passou a aumentar quando o processador do coordenador da transação atingiu uma taxa de utilização de 100%. Em outras palavras, o processador do coordenador foi, nos nossos experimentos, o fator limitante que provocou o aumento dos tempos medidos.

O crescimento do tempo de execução do experimento foi linear porque os únicos trechos de código sincronizado executados foram aqueles utilizados na criação e na efetivação da transação. Embora o registro dos recursos transacionais e a execução do protocolo *2PC* também envolvam o uso de travas, as travas empregadas nesses casos não são globais, mas sim "por transação". Isso quer dizer, por exemplo, que dois recursos transacionais podem ser alistados em transações distintas concorrentemente. Isso é diferente do que ocorre, por exemplo, na criação do identificador de uma transação. Nesse caso, duas transações não podem obter um identificador simultaneamente.

Como há apenas um cliente por transação (existem vários clientes simultâneos, mas cada um está associado a apenas uma transação), não há acessos concorrentes aos trechos protegidos pelas travas "por transação" e, portanto, não ocorrem atrasos causados por contenção nessas travas. Cabe ressaltar que no experimento também não houve atrasos gerados por acessos simultâneos aos recursos *XA*, dado que cada *Web service* sempre instanciava um novo recurso *XA* arremedado e o alistava na transação.




(a) O tempo médio gasto pelo cliente para criar uma transação, efetuar duas chamadas transacionais e então efetivar a transação.




(b) Número médio de transações criadas e efetivadas pelo gerenciador de transações no intervalo de um segundo.

Figura 5.17: Resultados dos experimentos envolvendo recursos *XA* arremedados.

## 5.2 Interoperabilidade

Um dos principais objetivos da tecnologia *Web services* é a interoperabilidade. Há, portanto, um grande esforço da indústria para promover a interoperabilidade entre os *Web services*. Um importante passo nesse sentido foi dado em abril de 2004 com a publicação do documento *WS-I Basic Profile* [42], que complementa as especificações fundamentais de *Web services* com uma série de esclarecimentos e restrições. Destinado principalmente a desenvolvedores de middleware, o *WS-I Basic Profile* permitiu um grande avanço na interoperabilidade de diversas plataformas para a execução de *Web services*.

A publicação das denominadas "especificações WS-*", tais como *WS-Security* [7], *WS-Policy* [14], *WS-Coordination* e *WS-AtomicTransaction*, entre muitas outras, também con-

tribuiu para a interoperabilidade entre os *Web services*. Por exemplo, caso as especificações *WS-Coordination* e *WS-AtomicTransaction* não tivessem sido publicadas, não haveria nenhum modo padronizado de se efetuar chamadas transacionais entre *Web services*. Cada desenvolvedor de middleware teria de implementar o seu próprio mecanismo para realizar chamadas transacionais, resultando num cenário onde a interoperabilidade entre diferentes sistemas de middleware seria praticamente infactível. Em contrapartida, com a publicação de *WS-Coordination* e *WS-AtomicTransaction*, sistemas compatíveis com essas especificações devem, pelo menos em princípio, ser capazes de interoperar entre si.

Devido à grande relevância da interoperabilidade no escopo da tecnologia *Web services*, é importante que o serviço de transações para *Web services* desenvolvido como parte do nosso estudo seja capaz de interagir com outras implementações de *WS-Coordination* e *WS-AtomicTransaction*. Tendo isso em vista, efetuamos experimentos de interoperabilidade com dois produtos que implementam essas especificações: o *ArjunaTS* e o *IBM WebSphere*. A seguir apresentamos em maior detalhe os experimentos realizados com cada um desses produtos.

### 5.2.1 ArjunaTS

Nos experimentos que realizamos com o *ArjunaTS*, este desempenhou dois papéis. Em alguns experimentos ele atuou como o coordenador e em outros como participante da transação distribuída. Nos experimentos em que uma instância do *ArjunaTS* exerceu o papel de coordenador da transação, instâncias do servidor de aplicações *JBoss* atuaram como participantes. Já quando o *ArjunaTS* atuou como um participante da transação, uma instância do *JBoss* exerceu o papel de coordenador (Figura 5.18). Desse modo, o nosso serviço transacional para *Web services* foi exercitado tanto no papel de coordenador quanto no papel de participante.

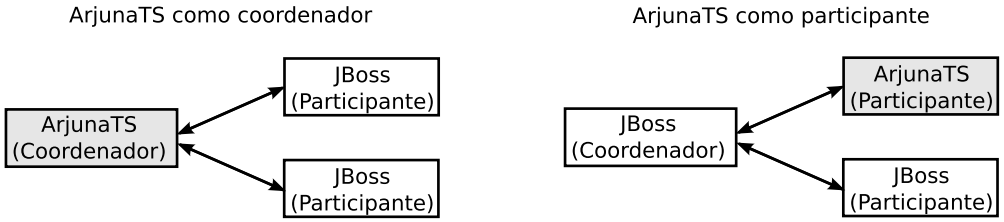



Figura 5.18: Nos experimentos de interoperabilidade, o *ArjunaTS* desempenhou os papéis de coordenador e participante de uma transação distribuída.

Os experimentos em que o *ArjunaTS* desempenha o papel de coordenador são importantes para averiguar se o serviço transacional que implementamos é capaz de interagir com um coordenador externo que não é uma instância do *JBoss*. Por outro lado, os experimentos em

que o *ArjunaTS* desempenha o papel de participante (e uma instância do *JBoss* desempenha, portanto, o papel de coordenador) são importantes para averiguar se o serviço transacional que implementamos é capaz de interagir com participantes que não são instâncias do *JBoss*.

Cenários em que há presença de falhas não foram contemplados pelos experimentos de interoperabilidade com o *ArjunaTS*. Isso ocorreu porque a versão do *ArjunaTS* disponível por ocasião da realização dos experimentos apresentava problemas na recuperação de falhas em transações envolvendo *Web services*. Na realidade, foram desenvolvidos experimentos de interoperabilidade que envolvem falhas, mas eles não executam corretamente devido aos problemas apresentados pelo *ArjunaTS*.

A nossa versão estendida do gerenciador de transações do *JBoss* é capaz de interoperar satisfatoriamente com o *ArjunaTS*, exceto nos casos em que este último apresenta problemas.

### 5.2.2 WebSphere

Em novembro de 2004, a *IBM* e a *Microsoft* publicaram conjuntamente um documento denominado *WS-AtomicTransaction Interop Scenarios* (*WSATIS*) [26]. Ele define dezessete experimentos para verificar a interoperabilidade entre duas implementações de *WS-Coordination* e *WS-AtomicTransaction*, além de uma série de restrições sobre os formatos das mensagens definidas nessas especificações. Esse documento foi utilizado como base para *workshops* de interoperabilidade, nos quais empresas como *IBM*, *Microsoft*, *Arjuna* e *IONA* avaliaram a interoperabilidade entre as suas implementações das especificações.

O grau de complexidade dos experimentos descritos no *WSATIS* varia bastante. Alguns deles são extremamente simples e não envolvem nenhum tipo de falha. Outros, entretanto, são bastante complexos, envolvendo perdas de mensagens (simulando falhas de comunicação) e quedas de participantes. Podemos dizer, portanto, que o conjunto de experimentos definidos no *WSATIS* é bastante completo e permite avaliar satisfatoriamente a interoperabilidade entre duas implementações de *WS-Coordination* e *WS-AtomicTransaction*.

Embora os experimentos descritos no *WSATIS* não contemplem todos os possíveis cenários de interoperabilidade, tais experimentos exercitam todas as mensagens definidas nas especificações *WS-Coordination* e *WS-AtomicTransaction*. Isso significa que, durante a execução dos experimentos, todas as mensagens definidas nessas especificações são trocadas pelo menos uma vez entre as duas implementações testadas.

Além de descrever os experimentos, o *WSATIS* complementa o formato de cada uma das mensagens definidas nas especificações *WS-Coordination* e *WS-AtomicTransaction*. Ele indica, por exemplo, quais informações de endereçamento (definidas em *WS-Addressing*) são obrigatórias em cada mensagem.

A *IBM*, além de ser uma das autoras do *WSATIS*, mantém uma instância do seu servidor de aplicações, o *WebSphere*, disponível publicamente[7] para experimentos de interoperabilidade. Essa instância do *WebSphere* implementa toda a infra-estrutura exigida pelo *WSATIS* para a execução dos experimentos de interoperabilidade. Isso é interessante pois permite que implementadores das especificações *WS-Coordination* e *WS-AtomicTransaction* possam executar, de maneira bastante simples, experimentos de interoperabilidade com o *WebSphere*.

Executamos todos os experimentos definidos no *WSATIS* utilizando a nossa versão estendida do gerenciador de transações do *JBoss* e o *WebSphere*. No caso deste último, foi utilizada a instância disponibilizada publicamente pela *IBM*. A execução dos experimentos mostrou que a nossa implementação é capaz de interoperar com o *WebSphere* nos dezessete cenários descritos pelo *WSATIS*. Portanto, podemos dizer que a nossa implementação possui interoperabilidade satisfatória com o *WebSphere*.

### 5.2.3 Nota sobre os experimentos de interoperabilidade

Embora os experimentos realizados assegurem a interoperabilidade de nossa implementação com o *ArjunaTS* e o *WebSphere* numa grande variedade de cenários, tais experimentos estão longe de contemplar todos os cenários possíveis. Todas as mensagens definidas em *WS-Coordination* e *WS-AtomicTransaction* foram exercitadas nos experimentos de interoperabilidade com o *WebSphere*. Nos experimentos com o *ArjunaTS*, apenas a mensagem `Replay` não foi testada. Isso não garante, contudo, que o nosso serviço transacional seja interoperável com essas duas outras implementações em todos os possíveis casos.

[7] `http://wsi.alphaworks.ibm.com:8080/interop/index.html`

# 6 Considerações finais

Até onde sabemos, o *JBoss* é hoje o único servidor de aplicações de código aberto que possui um gerenciador de transações com suporte a recuperação de falhas e aos padrões *CORBA OTS* e *WS-AtomicTransaction*. Se esse gerenciador de transações não dependesse de código específico do servidor *JBoss*, ele poderia ser empregado em outros ambientes e em outros servidores de aplicações. Para permitir isso, iniciamos um novo projeto, denominado *XActor* [57, 82], baseado no código do gerenciador de transações do *JBoss*.

Este capítulo está dividido em três seções. A Seção 6.1 discute o projeto *XActor*. A Seção 6.2 sumariza as principais contribuições do nosso trabalho e o compara com projetos similares. Por fim, a Seção 6.3 discute direções para trabalho futuro.

## 6.1 XActor

Após implementarmos nosso serviço de transações para *Web services*, iniciamos o projeto *XActor* com dois principais objetivos: *(i)* construir um gerenciador de transações independente de servidores de aplicações e *(ii)* permitir a implantação dinâmica de suporte a novos mecanismos de chamada remota. O projeto consiste numa bifurcação (*fork*) aprimorada do gerenciador de transações do *JBoss*. Vamos discutir a seguir os dois principais objetivos do *XActor*.

### 6.1.1 Independência do servidor de aplicações JBoss

O código inicial do *XActor* foi o código do gerenciador de transações do *JBoss*. Partindo desse código, iniciamos um processo de refatoração de modo a tornar o *XActor* independente de qualquer servidor de aplicações. Por ter sido baseado no gerenciador de transações do *JBoss*, que implementa os padrões *CORBA OTS* e *WS-Coordination*/*WS-AtomicTransaction*, o *XActor* também implementa esses padrões. Além disso, ele oferece suporte a recuperação de falhas, protegendo a integridade das aplicações de negócio mesmo na presença de quedas do coordenador ou de qualquer participante de uma transação distribuída.

Atualmente o *XActor* pode ser executado nas versões 4.2.1.GA e 5.0.0.Beta2 do servidor de aplicações *JBoss*. Embora o *XActor* já seja independente do *JBoss*, ainda é necessário desenvolver camadas de integração para que ele comporte outros servidores de aplicações. Além de servidores de aplicações, temos planos de permitir a integração do *XActor* com arcabouços de injeção de dependências, como por exemplo o *Spring* [78].

### 6.1.2 Implantação dinâmica de novos mecanismos de chamada remota

Além da independência de servidores de aplicações, outro importante objetivo do *XActor* era possibilitar a extensão do gerenciador de transações de uma maneira bastante simples. Portanto, além de refatorarmos o código do gerenciador de transações do *JBoss* para torná-lo independente de qualquer servidor de aplicações, também o refatoramos para torná-lo mais flexível. Nosso objetivo era permitir a adição de suporte a novos mecanismos de chamada remota sem exigir a recompilação do gerenciador de transações.

Esse objetivo foi atingido no *XActor* através da criação do conceito de *transactional remote method invocation plug-in*, ou simplesmente *TRMI plug-in*. Um *TRMI plug-in* é uma fina camada de software que: *(i)* implementa um conjunto de interfaces definidas pelo *XActor* e *(ii)* encapsula um mecanismo de chamada remota. O *XActor* não interage diretamente com mecanismos de chamada remota. Ele o faz apenas através dos *TRMI plug-ins*, que implementam um conjunto de interfaces conhecidas pelo *XActor*.

Atualmente o *XActor* possui *TRMI plug-ins* para três mecanismos de chamada remota: *Web services/SOAP*, *CORBA/IIOP* e *JBoss Remoting*. Quando esses três *plug-ins* estão implantados, o gerenciador de transações é capaz de coordenar transações distribuídas envolvendo qualquer um desses mecanismos de chamada remota. Uma mesma transação pode, inclusive, envolver múltiplos mecanismos de chamada remota (e.g., *CORBA/IIOP* e *Web services/SOAP*). Contudo, o *XActor* é capaz de operar mesmo sem nenhum *TRMI plug-in*. Nesse caso, ele pode coordenar apenas transações *XA*. O *XActor* possui, portanto, uma arquitetura bastante modular. Somente os *TRMI plug-ins* necessários precisam ser implantados. Por exemplo, para que o *XActor* comporte transações distribuídas via *CORBA/IIOP*, basta implantar o *TRMI plug-in* que encapsula esse mecanismo de chamada remota.

Uma característica interessante do *XActor* é que, em servidores de aplicações que comportam implantação dinâmica de componentes de serviço[1], como é o caso do *JBoss*, os *TRMI plug-ins* podem ser implantados também dinamicamente. Em outras palavras, a funcionalidade do gerenciador de transações pode ser aumentada em tempo de execução. Por exemplo, uma instância do *XActor* pode estar em execução com apenas o plug-in para *CORBA/IIOP*

[1] Um componente de serviço é um componente que estende a funcionalidade do servidor de aplicações.

implantado. Se, por alguma razão, além de transações *CORBA* for também necessário suporte a transações sobre *Web services*, basta implantar o *TRMI plug-in* para *Web services*. Para isso, não é necessário nem sequer parar a execução do gerenciador de transações. Em outras palavras, o *XActor* é capaz de coordenar transações distribuídas que empregam um conjunto extensível de mecanismos de chamada remota. Ademais, esse conjunto pode crescer em tempo de execução. Até onde temos conhecimento, nenhum outro gerenciador de transações oferece tal grau de flexibilidade.

### 6.1.3 Mais informações

Todo o código fonte do *XActor*, além de instruções de instalação e documentação, podem ser encontrados no seguinte endereço eletrônico: `http://xactor.sourceforge.net`.

## 6.2 Principais contribuições

Consideramos ser estas as principais contribuições deste trabalho:

- Uma implementação em código aberto das especificações *WS-Coordination* e *WS-Atomic-Transaction*.
- Uma avaliação comparativa do desempenho de transações distribuídas empregando três diferentes mecanismos de chamada remota.

Além dessas contribuições, a adição de um terceiro mecanismo de chamada remota (*Web services/SOAP*) ao gerenciador de transações do *JBoss* nos permitiu validar a extensibilidade desse gerenciador. A experiência que adquirimos adicionando esse terceiro mecanismo nos capacitou a propor melhorias na arquitetura do gerenciador, tendo como objetivo que este passasse a comportar novos mecanismos de chamada remota de uma maneira plugável. Embora o gerenciador de transações do *JBoss* fosse extensível, a adição de um novo mecanismo de chamada remota exigia a recompilação do gerenciador. Nossas propostas de melhoria da arquitetura, que resultaram no projeto *XActor*, eliminaram essa necessidade e possibilitaram que o suporte a um novo mecanismo de chamada remota fosse dinamicamente adicionado ao gerenciador de transações.

As contribuições correspondentes aos dois itens acima são discutidas nas próximas seções.

### 6.2.1 Implementação em código aberto das especificações WS-Coordination e WS-AtomicTransaction

Desenvolvemos, como parte deste trabalho, um serviço de transações atômicas voltado para *Web services* e compatível com as especificações *WS-Coordination* e *WS-AtomicTransaction*. Todo o código fonte de nosso projeto [82] está disponível sob a licença *LGPL*[2]. Até onde temos conhecimento, nosso serviço é a única implementação em código aberto das especificações *WS-Coordination* e *WS-AtomicTransaction* que oferece suporte a recuperação de falhas e que é capaz de interoperar com outra implementação dessas especificações em todos os cenários descritos no documento *WS-AtomicTransaction Interop Scenarios*.

Comparamos a seguir nosso serviço de transações com duas outras implementações em código aberto: o *Apache Kandula* e o *ArjunaTS*.

#### 6.2.1.1 Comparação com o Apache Kandula

Vimos na Seção 2.3 que o *Apache Kandula* é uma implementação razoavelmente completa das especificações *WS-Coordination* e *WS-AtomicTransaction*. Ele apresenta, contudo, as seguintes deficiências, relacionadas em ordem crescente de relevância:

- O registro de um recurso transacional com o coordenador da transação ocorre prematuramente, isto é, no momento em que um *Web service* recebe uma requisição transacional. O correto seria que o registro do recurso transacional fosse postergado até que o *Web service* executasse algum trabalho como parte da transação — fazendo acesso a um banco de dados, por exemplo.
- A implementação do *Kandula* é vinculada ao *Apache Axis*. O desejável seria que um serviço de transações não fosse vinculado a nenhuma plataforma de execução de *Web services*.
- O *Kandula* comporta exclusivamente recursos transacionais acessíveis como *Web services*.
- Não há suporte a recuperação de falhas, pois o gerenciador de transações trabalha apenas com dados em memória volátil.

Nossa implementação não apresenta nenhuma dessas deficiências.

[2] `http://www.gnu.org/licenses/lgpl.html`

#### 6.2.1.2 Comparação com o ArjunaTS

Há quatro características da nossa implementação que merecem ser destacadas em relação ao *ArjunaTS*. Essas características estão listadas a seguir em ordem crescente de relevância:

- As mensagens *SOAP* geradas pelo *ArjunaTS* são excessivamente prolixas, pois as declarações de espaços de nomes *XML* são repetidas várias vezes na mesma mensagem. A Figura 6.1 exibe uma mensagem `Register` gerada pelo *ArjunaTS*. Nessa mensagem, o espaço de nomes com prefixo `wsa` é declarado sete vezes, quando poderia ser declarado uma única vez. Nas mensagens geradas pelo nosso serviço de transações, tivemos o cuidado de evitar repetições de declarações de espaços de nomes *XML*. A Figura 6.2 exibe uma mensagem `Register` gerada pelo nosso serviço. Note que nessa mensagem não há repetição de nenhuma declaração de espaço de nomes *XML*.

- Desenvolvemos uma implementação da interface `javax.transaction.UserTransaction` que se comunica com o servidor de aplicações através do protocolo *SOAP*. Essa implementação pode, portanto, ser utilizada por clientes *stand-alone* que são executados fora da máquina virtual do servidor. No *ArjunaTS*, a demarcação de transações utilizando a interface `UserTransaction` pode ser realizada exclusivamente por clientes que são executados na máquina virtual do servidor de aplicações.

- No *ArjunaTS*, os recursos remotos acessíveis como *Web services* não são automaticamente registrados com o coordenador da transação. Para entendermos melhor isso, tomemos como exemplo um *Web service* que utiliza um banco de dados e que está implantado numa instância de servidor de aplicações que não é o coordenador da transação. Suponhamos que um cliente faça uma chamada transacional a esse *Web service*. No momento em que o *Web service* utilizar pela primeira vez o banco de dados, um *XAResource*[3] correspondente a esse banco de dados será alistado junto ao gerenciador de transações local (que não é o coordenador da transação global) como participante da transação. Em conseqüência disso, um recurso remoto acessível como *Web service* deve ser registrado com o coordenador global. No caso do nosso trabalho, o gerenciador de transações local se encarrega de criar um *Web service* representando esse recurso remoto e de registrá-lo com o coordenador global. Tudo isso ocorre de modo transparente para o desenvolvedor, ficando o middleware responsável por todas as tarefas relacionadas com o gerenciamento da transação distribuída. No caso do *ArjunaTS*, entretanto, o desenvolvedor tem a responsabilidade de criar um *Web service* representando o recurso remoto

[3] A interface *XAResource* é definida no padrão *XA*.

e de registrá-lo com o coordenador global. Em outras palavras, em nosso trabalho o registro do recurso remoto é efetuado implicitamente pelo middleware, enquanto que no *ArjunaTS* ele tem de ser feito explicitamente pelo desenvolvedor[4].

- O suporte a recuperação de falhas do *Arjuna XML Transaction Service* — módulo do *ArjunaTS* que lida com transações envolvendo *Web services* — ainda não é funcional[5]. Nosso trabalho não apresenta esse problema.

### 6.2.2 Avaliação comparativa do desempenho de transações distribuídas

No Capítulo 5 discutimos uma série de experimentos que foram realizados com a nossa versão estendida do gerenciador de transações do *JBoss*. Os resultados obtidos com a realização desses experimentos foram:

- Uma comparação de desempenho entre transações distribuídas empregando cada um dos seguintes mecanismos de chamada remota: *JBoss Remoting*, *CORBA/IIOP* e *Web services/SOAP*.
- Uma medida da sobrecarga imposta pela propagação do contexto transacional em cada um desses três mecanismos de chamada remota.
- Uma estimativa da sobrecarga total imposta pelo uso de um serviço transacional. Comparamos, inclusive, a sobrecarga imposta com o *log* transacional habilitado e desabilitado.
- Uma avaliação do desempenho do serviço de transações para *Web services* em função do número de clientes simultâneos.

Até onde sabemos, resultados dessa natureza não foram publicados na literatura.

### 6.2.3 Publicações

Até o momento, o trabalho apresentado nesta dissertação gerou as seguintes publicações:

[4]Está no plano de trabalho do *ArjunaTS* permitir o registro implícito de recursos remotos acessíveis como *Web services*. Na nomenclatura empregada pelo *ArjunaTS*, essa funcionalidade é denominada *transaction bridging* ou *end-to-end transaction*. Maiores detalhes podem ser vistos no seguinte endereço: `http://www.jboss.com/index.html?module=bb&op=viewtopic&t=84198`.

[5]Maiores detalhes podem ser encontrados em `http://jira.jboss.com/jira/browse/JBTM-121`.

```
<soap:Envelope xmlns:soap='http://schemas.xmlsoap.org/soap/envelope/'>
 <soap:Header>
  <wsa:To xmlns:wsa='http://schemas.xmlsoap.org/ws/2004/08/addressing'>
   http://192.168.1.1:8080/jbossws/wscoor/RegistrationCoordinator
  </wsa:To>
  <wsa:Action xmlns:wsa='http://schemas.xmlsoap.org/ws/2004/08/addressing'>
   http://schemas.xmlsoap.org/ws/2004/10/wscoor/Register
  </wsa:Action>
  <wsa:MessageID xmlns:wsa='http://schemas.xmlsoap.org/ws/2004/08/addressing'>
   -3f57feff:8f79:4644cac1:82
  </wsa:MessageID>
  <wsa:From xmlns:wsa='http://schemas.xmlsoap.org/ws/2004/08/addressing'>
   <wsa:Address>http://192.168.1.6:8080/xts/soap/RegistrationRequester</wsa:Address>
  </wsa:From>
  <wsa:ReplyTo xmlns:wsa='http://schemas.xmlsoap.org/ws/2004/08/addressing'>
   <wsa:Address>http://192.168.1.6:8080/xts/soap/RegistrationRequester</wsa:Address>
  </wsa:ReplyTo>
  <jbosswsc:CoordinatedActivityID xmlns:jbosswsc='http://www.jboss.org/wscoor/extension'>
   7
  </jbosswsc:CoordinatedActivityID>
 </soap:Header>
 <soap:Body>
  <wscoor:Register xmlns:wscoor='http://schemas.xmlsoap.org/ws/2004/10/wscoor'>
   <wscoor:ProtocolIdentifier>
    http://schemas.xmlsoap.org/ws/2004/10/wsat/Durable2PC
   </wscoor:ProtocolIdentifier>
   <wscoor:ParticipantProtocolService>
    <wsa:Address xmlns:wsa='http://schemas.xmlsoap.org/ws/2004/08/addressing'>
     http://192.168.1.6:8080/xts/soap/ATParticipant
    </wsa:Address>
    <wsa:ReferenceParameters xmlns:wsa='http://schemas.xmlsoap.org/ws/2004/08/addressing'>
     <wsarj:InstanceIdentifier xmlns:wsarj='http://schemas.arjuna.com/ws/2005/10/wsarj'>
      -3f57feff:8f79:4644cac1:81
     </wsarj:InstanceIdentifier>
    </wsa:ReferenceParameters>
   </wscoor:ParticipantProtocolService>
  </wscoor:Register>
 </soap:Body>
</soap:Envelope>
```

Figura 6.1: Exemplo de mensagem `Register` gerada pelo *ArjunaTS*.

- *Dynamic Support to Transactional Remote Invocations over Multiple Transports* [57]: artigo a ser publicado no *23rd Annual ACM Symposium on Applied Computing (SAC 2008)*. Descreve a arquitetura empregada no *XActor*, com ênfase em seu suporte a implantação dinâmica a novos mecanismos de chamada remota.

- *Lessons Learned from Implementing WS-Coordination and WS-AtomicTransaction* [61]: artigo em fase de submissão. Descreve a nossa implementação de *WS-Coordination* e *WS-AtomicTransaction*, com ênfase nas lições aprendidas durante a implementação dessas especificações.

Está em fase de preparação um terceiro artigo, contendo nossos resultados experimentais.

```
<env:Envelope xmlns:env='http://schemas.xmlsoap.org/soap/envelope/'
  xmlns:jbosswsc='http://www.jboss.org/wscoor/extension'
  xmlns:wsa='http://schemas.xmlsoap.org/ws/2004/08/addressing'
  xmlns:wscoor='http://schemas.xmlsoap.org/ws/2004/10/wscoor'>
 <env:Header>
  <jbosswsc:CoordinatedActivityID>6</jbosswsc:CoordinatedActivityID>
  <wsa:MessageID>uuid:d39448a4-f0fe-4e35-8fdf-c8b73c564cea</wsa:MessageID>
  <wsa:To>http://192.168.1.1:8080/jbossws/wscoor/RegistrationCoordinator</wsa:To>
  <wsa:Action>http://schemas.xmlsoap.org/ws/2004/10/wscoor/Register</wsa:Action>
  <wsa:ReplyTo>
   <wsa:Address>http://192.168.1.2:8080/jbossws/wscoor/RegistrationRequester</wsa:Address>
  </wsa:ReplyTo>
 </env:Header>
 <env:Body>
  <wscoor:Register>
   <wscoor:ProtocolIdentifier>
    http://schemas.xmlsoap.org/ws/2004/10/wsat/Durable2PC
   </wscoor:ProtocolIdentifier>
   <wscoor:ParticipantProtocolService>
    <wsa:Address>http://192.168.1.2:8080/jbossws/wsat/Participant</wsa:Address>
    <wsa:ReferenceProperties>
     <jbosswsc:CoordinatedActivityID>35184372088833</jbosswsc:CoordinatedActivityID>
    </wsa:ReferenceProperties>
   </wscoor:ParticipantProtocolService>
  </wscoor:Register>
 </env:Body>
</env:Envelope>
```

Figura 6.2: Exemplo de mensagem `Register` gerada pelo nosso serviço de transações para *Web services*.

## 6.3 Trabalhos futuros

**Desenvolvimento de mais experimentos de interoperabilidade.** Os experimentos de interoperabilidade podem ser ampliados. Além de aumentar o número de experimentos de interoperabilidade com o *ArjunaTS* e com o *WebSphere*, seria interessante também desenvolver experimentos para verificar a interoperabilidade da nossa implementação com implementações adicionais, como por exemplo a da *Microsoft* [60].

**Elaboração de mais experimentos de avaliação de desempenho.** Além de comparar o desempenho de transações distribuídas que empregam diferentes mecanismos de chamada remota, seria interessante comparar o desempenho do nosso serviço de transações com outros serviços. Bons candidatos para essa comparação seriam o *ArjunaTS* e os serviços de transações presentes no *WebSphere* e na plataforma *.NET*.

Nos nossos experimentos de avaliação de desempenho, utilizamos computadores pertencentes a uma mesma rede local. Seria interessante executar os mesmos experimentos utilizando máquinas interligadas por enlaces de baixa velocidade, como por exemplo enlaces *ADSL*. Proto-

colos prolixos, como o *JBoss Remoting* e o *SOAP*, certamente apresentariam uma considerável degradação de desempenho nesse cenário.

**Implementação de WS-BusinessActivity.** Há situações onde o modelo tradicional de transações atômicas não é adequado. É o caso, por exemplo, das transações de longa duração e das interações negócio-a-negócio. Para que o nosso serviço de transações possa lidar com esses cenários, pretendemos implementar também a especificação *WS-BusinessActivity*.

**Implementação das novas versões das especificações.** Em abril de 2007 foi publicada, no âmbito do *OASIS* [55], a versão 1.1 das especificações *WS-Coordination* [50] e *WS-AtomicTransaction* [51]. As novas versões das especificações apresentam poucas alterações. As mais significativas são: *(i)* a alteração dos espaços de nomes utilizados pelos elementos *XML* definidos nas especificações e *(ii)* o uso de uma versão mais nova da especificação *WS-Addressing* [32]. Seria interessante estender o nosso serviço transacional para que ele comportasse também a versão 1.1 das especificações (atualmente ele é compatível apenas com a versão 1.0 das especificações).

**Execução do XActor em outros ambientes além do JBoss.** Tencionamos desenvolver camadas de integração que permitam executar o *XActor* em outros servidores de aplicações e em arcabouços de injeção de dependências, como por exemplo o *Spring*.

**Suporte a mecanismos adicionais de chamada remota.** Temos planos de desenvolver novos *TRMI plug-ins*, de modo a ampliar o conjunto de mecanismos de chamada remota comportado pelo *XActor*. Seria interessante desenvolver, por exemplo, um *TRMI plug-in* que encapsule o mecanismo de chamada remota empregado pelo *Internet Communications Engine* (ICE) [33].

# Referências Bibliográficas


[1] Nayef Abu-Ghazaleh, Michael J. Lewis, and Madhusudhan Govindaraju. Differential serialization for optimized SOAP performance. In *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing (HPDC'04)*, pages 55–64, Washington, DC, USA, 2004. IEEE Computer Society.

[2] Yousef J. Al-Houmaily and Panos K. Chrysanthis. 1-2PC: The one-two phase atomic commit protocol. In *Proceedings of the 2004 ACM Symposium on Applied Computing*, pages 684–691, New York, NY, USA, 2004. ACM Press.

[3] Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. *Web Services: Concepts, Architecture and Applications*. Springer Verlag, 2004.

[4] Apache JMeter. `http://jakarta.apache.org/jmeter/`. Último acesso em 20 de maio de 2007.

[5] Apache Kandula. `http://ws.apache.org/kandula/`. Último acesso em 27 de maio de 2006.

[6] Vidur Apparao et al. Document Object Model (DOM) Level 1 Specification. World Wide Web Consortium, Recommendation REC-DOM-Level-1-19981001, October 1998.

[7] Bob Atkinson et al. *Web Services Security (WS-Security)*. IBM, Microsoft and VeriSign, April 2002.

[8] BEA Systems. *Streaming API for XML Specification, v1.0*, October 2003.

[9] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform Resource Identifier (URI): Generic Syntax. RFC 3986 (Standard), January 2005.

[10] Tim Berners-Lee, Robert Cailliau, Ari Luotonen, Henrik Frystyk Nielsen, and Arthur Secret. The World-Wide Web. *Communications of the ACM*, 37(8):76–82, 1994.

[11] Paul V. Biron and Ashok Malhotra. XML schema part 2: Datatypes, second edition. World Wide Web Consortium, Recommendation REC-xmlschema-2-20041028, October 2004.

[12] David Booth et al. Web services architecture. World Wide Web Consortium, Note NOTE-ws-arch-20040211, February 2004.

[13] Don Box et al. *Web Services Addressing (WS-Addressing).* BEA Systems, IBM, and Microsoft, August 2005.

[14] Don Box et al. *Web Services Policy Framework (WS-Policy).* IBM, BEA Systems and Microsoft, March 2006.

[15] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, François Yergeau, and John Cowan. Extensible markup language (XML) 1.1, second edition. World Wide Web Consortium, Recommendation REC-xml11-20060816, August 2006.

[16] Gerald Brose. JacORB: Implementation and design of a Java ORB. In *Proceedings of Distributed Applications and Interoperable Systems '97*, pages 143–154, Cottbus, Germany, 1997. Chapman & Hall.

[17] Luis Felipe Cabrera et al. *Web Services Atomic Transaction (WS-AtomicTransaction) 1.0.* Arjuna, BEA Systems, Hitachi, IBM, IONA and Microsoft, August 2005.

[18] Luis Felipe Cabrera et al. *Web Services Business Activity Framework (WS-BusinessActivity) 1.0.* Arjuna, BEA Systems, Hitachi, IBM, IONA and Microsoft, August 2005.

[19] Luis Felipe Cabrera et al. *Web Services Coordination (WS-Coordination) 1.0.* Arjuna, BEA Systems, Hitachi, IBM, IONA and Microsoft, August 2005.

[20] David Chappell. *Understanding .NET, Second Edition.* Addison-Wesley, 2006.

[21] Roberto Chinnici, Hugo Haas, Amelia A. Lewis, Jean-Jacques Moreau, David Orchard, and Sanjiva Weerawarana. Web services description language (WSDL) version 2.0 part 2: Adjuncts. World Wide Web Consortium, Candidate Recommendation CR-wsdl20-adjuncts-20060327, March 2006.

[22] Roberto Chinnici, Jean-Jacques Moreau, Arthur Ryman, and Sanjiva Weerawarana. Web services description language (WSDL) version 2.0 part 1: Core language. World Wide Web Consortium, Candidate Recommendation CR-wsdl20-20060327, March 2006.

[23] Kenneth Chiu, Madhusudhan Govindaraju, and Randall Bramley. Investigating the limits of SOAP performance for scientific computing. In *Proceedings of the 11 th IEEE International Symposium on High Performance Distributed Computing (HPDC'02)*, page 246, Washington, DC, USA, 2002. IEEE Computer Society.

[24] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Updated by RFC 2817.

[25] Marc Fleury and Francisco Reverbel. The JBoss extensible server. In *Middleware 2003 — ACM/IFIP/USENIX International Middleware Conference*, volume 2672 of *LNCS*, pages 344–373. Springer-Verlag, 2003.

[26] Thomas Freund et al. *WS Atomic Transaction Interop Scenarios*. IBM and Microsoft, November 2004.

[27] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995.

[28] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer D. Widom. *Database System Implementation*. Prentice Hall, 2004.

[29] Madhusudhan Govindaraju, Aleksander Slominski, Venkatesh Choppella, Randall Bramley, and Dennis Gannon. Requirements for an evaluation of RMI protocols for scientific computing. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, page 61, Washington, DC, USA, 2000. IEEE Computer Society.

[30] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, and Henrik Frystyk Nielsen. SOAP version 1.2 part 1: Messaging framework. World Wide Web Consortium, Recommendation REC-soap12-part1-20030624, June 2003.

[31] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, and Henrik Frystyk Nielsen. SOAP version 1.2 part 2: Adjuncts. World Wide Web Consortium, Recommendation REC-soap12-part2-20030624, June 2003.

[32] Martin Gudgin, Marc Hadley, and Tony Rogers. Web services addressing (ws-addressing) 1.0 - core. World Wide Web Consortium, Recommendation REC-ws-addr-core-20060509, May 2006.

[33] Michi Henning. A new approach to object-oriented middleware. *IEEE Internet Computing*, 8(1):66–75, 2004.

[34] Romin Irani and S. Jeelani Basha. *AXIS: Next Generation Java SOAP*. Peer Information, 2002.

[35] Java Open Application Server. `http://jonas.objectweb.org/`. Último acesso em 26 de maio de 2006.

[36] Java Open Transaction Manager. `http://jotm.objectweb.org/`. Último acesso em 10 de agosto de 2006.

[37] JBoss Remoting. `http://labs.jboss.com/portal/jbossremoting/`. Último acesso em 10 de agosto de 2006.

[38] JBoss Serialization. `http://labs.jboss.com/portal/serialization/`. Último acesso em 20 de abril de 2007.

[39] JBoss Transactions. `http://www.jboss.com/products/transactions/`. Último acesso em 10 de agosto de 2006.

[40] JBoss Web Services. `http://labs.jboss.com/portal/jbossws/`. Último acesso em 20 de abril de 2007.

[41] Jukka Korpela. *Unicode Explained*. O'Reilly, 2006.

[42] K. M. Senthil Kumar, Akash Saurav Das, and Srinivas Padmanabhuni. WS-I Basic Profile: A practitioner's view. In *Proceedings of the IEEE International Conference on Web Services (ICWS'04)*, page 17, Washington, DC, USA, 2004. IEEE Computer Society.

[43] P. Leach, M. Mealling, and R. Salz. Universally Unique IDentifier (UUID) URN Namespace. RFC 4122 (Standard), July 2005.

[44] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley, 1999.

[45] Mark C. Little and Santosh K. Shrivastava. An examination of the transition of the Arjuna distributed transaction processing software from research to products. In *Proceedings of the 2nd International Workshop on Industrial Experience with Systems Software*, pages 41–54, Boston, MA, 2002. Usenix Association.

[46] Vincent Massol and Ted Husted. *JUnit in Action*. Manning, 2003.

[47] Robert M. Metcalfe and David R. Boggs. Ethernet: distributed packet switching for local computer networks. *Commun. ACM*, 19(7):395–404, 1976.

[48] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, 17(1):94–162, 1992.

[49] Aaron Mulder. *Apache Geronimo: Enterprise Java Development and Deployment*. Addison-Wesley, 2007.

[50] Eric Newcomer, Ian Robinson, Max Feingold, and Ram Jeyaraman. *Web Services Coordination (WS-Coordination) 1.1*. IBM, IONA and Microsoft, April 2007.

[51] Eric Newcomer, Ian Robinson, Mark Little, and Andrew Wilkinson. *Web Services Atomic Transaction (WS-AtomicTransaction) 1.1*. IBM, IONA, JBoss Inc. and Microsoft, April 2007.

[52] Object Management Group. *CORBA Transaction Service Specification, version 1.4*, March 2003.

[53] Object Management Group. *Common Object Request Broker Architecture: Core Specification, version 3.0.3*, March 2004.

[54] Orbix OTS. `http://www.iona.com/products/orbix/`. Último acesso em 10 de agosto de 2006.

[55] Organization for the Advancement of Structured Information Standards. `http://www.oasis-open.org/`. Último acesso em 17 de maio de 2007.

[56] Graham D. Parrington, Santosh K. Shrivastava, Stuart M. Wheater, and Mark C. Little. The design and implementation of Arjuna. *Computing Systems*, 8(3):255–308, 1995.

[57] Francisco Reverbel and Ivan Silva Neto. Dynamic support to transactional remote invocations over multiple transports. In *Proceedings of the 23rd Annual ACM Symposium on Applied Computing (SAC 2008)*, Fortaleza, Ceará, Brazil, 2008. To be published by ACM Press.

[58] W. Clay Richardson, Donald Avondolio, Scot Schrager, and Mark W. Mitchell. *Professional Java, JDK 5 Edition*. Wrox, 2005.

[59] Douglas C. Schmidt, Hans Rohnert, Michael Stal, and Dieter Schultz. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. John Wiley & Sons, 2000.

[60] John Sharp. *Microsoft Windows Communication Foundation Step by Step*. Microsoft Press, 2007.

[61] Ivan Silva Neto and Francisco Reverbel. Lessons learned from implementing WS-Coordination and WS-AtomicTransaction. Submetido para publicação.

[62] Sun Microsystems. *Java Naming and Directory Interface Specification, v1.2*, July 1999.

[63] Sun Microsystems. *Java Transaction Service Specification, v1.0*, December 1999.

[64] Sun Microsystems. *Java Message Service Specification, v1.1*, April 2002.

[65] Sun Microsystems. *Java Transaction API Specification, v1.0.1B*, November 2002.

[66] Sun Microsystems. *Enterprise JavaBeans Specification, v2.1*, November 2003.

[67] Sun Microsystems. *J2EE Connector Architecture Specification, v1.5*, November 2003.

[68] Sun Microsystems. *Java API for XML-Based RPC Specification, v1.1*, October 2003.

[69] Sun Microsystems. *Java Servlet Specification, v2.4*, November 2003.

[70] Sun Microsystems. *SOAP with Attachments API for Java Specification, v1.2*, October 2003.

[71] Sun Microsystems. *Java API for XML Web Services Specification, v2.0*, October 2005.

[72] Sun Microsystems. *Web Services Metadata for the Java Platform Specification, v2.0*, February 2005.

[73] Sun Microsystems. *Java Database Connectivity Specification, v4.0*, December 2006.

[74] Sun Microsystems. *Java Platform Enterprise Edition Specification, v1.5*, May 2006.

[75] The Open Group. *Distributed Transaction Processing: The XA Specification*. X/Open Company, December 1991.

[76] Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn. XML schema part 1: Structures, second edition. World Wide Web Consortium, Recommendation REC-xmlschema-1-20041028, October 2004.

[77] VisiBroker ITS. `http://info.borland.com/techpubs/its/`. Último acesso em 10 de agosto de 2006.

[78] Craig Walls and Ryan Breidenbach. *Spring in Action, Second Edition*. Manning, 2007.

[79] WebLogic. `http://www.bea.com/weblogic/`. Último acesso em 26 de maio de 2006.

[80] WebSphere. `http://www.ibm.com/websphere/`. Último acesso em 26 de maio de 2006.

[81] Woodstox. `http://woodstox.codehaus.org/`. Último acesso em 31 de julho de 2007.

[82] XActor. `http://xactor.sourceforge.net/`. Último acesso em 31 de julho de 2007.

[83] Paul C. Zikopoulos, George Baklarz, and Dan Scott. *Apache Derby – Off to the Races*. IBM Press, 2005.