Tolerância a falhas em Serviços Web

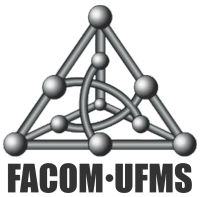
José Ricardo da Silva

Dissertação de Mestrado

Orientação: Prof. Dr. Irineu Sotoma

Área de Concentração: Computação Distribuída

Monografia apresentada como requisito para a obtenção do título de mestre em Ciência da Computação.



Faculdade de Computação
Universidade Federal de Mato Grosso do Sul
25 de janeiro de 2011

Pesquisa desenvolvida com suporte financeiro da Fundação de Apoio ao Desenvolvimento do Ensino, Ciência e Tecnologia do Estado de Mato Grosso do Sul (Fundect), proc. 41/100.270/2006 e suporte parcial do CNPq proc. 620171/2006-5.

Agradecimentos

À minha família, principalmente minha mãe e minha avó, que sempre apoiaram as minhas decisões e me ajudaram nos momentos de dificuldade.

Aos membros da banca examinadora, Prof. Dr. Irineu Sotoma, Prof. Dr. Edmundo Roberto Mauro Madeira e Prof. Dr. Ronaldo Alves Ferreira, pela leitura do texto da dissertação, pelas sugestões e pelas críticas construtivas, que permitiram melhorar o texto final da dissertação. Os mesmos também foram os membros da banca de qualificação de mestrado, contribuindo positivamente para a pesquisa.

À FUNDECT (processo número 41/100.270/2006) e ao CNPq (processo número 620171/2006-5), pelo apoio financeiro.

Aos colegas de mestrado, que sempre me ajudaram fornecendo esclarecimentos sobre as disciplinas e com dicas sobre a condução da pesquisa.

Aos amigos e namoradas, pelos momentos inesquecíveis de companheirismo.

Por fim, quero deixar um agradecimento especial ao meu orientador de mestrado, o Prof. Dr. Irineu Sotoma, que também foi meu orientador de trabalho de conclusão de curso. Sem o seu apoio técnico e motivacional e sua enorme paciência durante todos esses anos, a conclusão deste trabalho não teria sido possível. Irineu, muito obrigado.

Resumo

Apesar do grande número de trabalhos sobre tolerância a falhas baseados em técnicas de replicação de serviços web, até onde vai o nosso conhecimento, nenhum deles realiza a recuperação de falhas de maneira transparente ao cliente sem utilizar *proxies*. Além disso, a maioria das soluções é restrita a apenas uma plataforma de desenvolvimento.

Primeiramente, este trabalho trata o problema da recuperação de falhas de maneira transparente ao cliente através de uma extensão ao padrão WS-Addressing de serviços web, que permite a especificação de réplicas de um serviço e define a forma como as interações entre os *endpoints* devem ocorrer.

Em seguida, é feita a proposta de um novo *middleware* de tolerância a falhas em serviços web independente de plataforma de desenvolvimento, chamado SimpleRep. Adicionalmente, detalhes de implementação e avaliações dos protótipos construídos durante o trabalho também são fornecidas.

Palavras-chave: *sistemas distribuídos, tolerância a falhas, replicação, serviços web, simplerep*

Abstract

In spite of the great number of works on fault tolerance in web services, none of them, to the best of our knowledge, establishes a client-transparent failover mechanism without proxies. Besides, most of the solutions work only in a specific development platform.

Firstly, this work solves the client-transparent failover problem by extending the WS-Addressing web services standard. This extension allows the specification of a list of replicas of a service and defines the way interactions between endpoints must be done.

After that, a new platform-independent fault tolerance middleware for web services, called SimpleRep, is proposed. Additionally, the prototype implementation details and evaluations are presented.

Keywords: *distributed systems, fault tolerance, replication, web services, simplerep*

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Arquiteturas Orientadas a Serviços (SOA) [26, 43] têm sido adotadas nos mais variados cenários. A possibilidade de criar sistemas independentes de plataforma e de linguagem de programação e a possibilidade de integração de sistemas de maneira transparente fornecem novas oportunidades aos desenvolvedores. A utilização de serviços web[1] em sistemas críticos demanda a criação e a padronização de mecanismos que assegurem a dependabilidade[2] das soluções.

Em caso de falhas críticas, apenas técnicas de replicação podem garantir o acesso aos dados críticos, além de permitir uma reação coordenada [8, 6]. Assim sendo, a primeira melhoria necessária ao conjunto atual de especificações relativas a serviços web é a definição de mecanismos que permitam a utilização de serviços replicados.

Trabalhos anteriores sobre tolerância a falhas (TF) em serviços web podem ser divididos, basicamente, em duas categorias: os que utilizam replicação (ativa [52], passiva [9] ou uma variação, como a replicação semi-passiva [15], por exemplo) e os que utilizam implementações diferentes da mesma interface de serviço (*n-Versões* [3]).

Ortogonalmente às abordagens de TF se encontra a transparência da solução de TF ao cliente. Apesar do inegável valor das propostas que não são transparentes ao cliente, na prática é mais fácil adotar uma solução transparente. Isto pode ser dito porque, especificamente no caso de serviços web, é grande a heterogeneidade entre os clientes.

O principal objetivo da tecnologia de serviços web é permitir um alto nível de interoperabilidade entre as soluções. Para tornar isto possível, a arquitetura possui padrões abertos em seus alicerces. Entretanto, apesar da existência de vários padrões já estabelecidos pelo W3C e pela OASIS, o tópico de TF em serviços web ainda não foi padronizado. Há várias propostas (ver Seção 4.4.1) para adicionar TF às soluções, entretanto, até onde vai o nosso conhecimento, todas utilizam soluções *ad-hoc* para o tratamento de falhas. A única proposta que estende um padrão é a de Fang et al., que estende a WSDL [65].

[1] O termo "Serviço Web" neste texto é utilizado para se referir a serviços baseados em padrões do *World Wide Web Consortium* (W3C) e da *Organization for the Advancement of Structured Information Standards* (OASIS), como o SOAP, a *Web Services Definition Language* (WSDL) e o restante de padrões WS-*.

[2] Habilidade de evitar falhas que sejam mais frequentes ou severas do que o aceitável [31].

Contudo, essa proposta apresenta algumas deficiências que esperamos ter sanado com este trabalho (ver Seção 4.4).

A maioria das arquiteturas de TF propostas não pode ser considerada transparente ao cliente, pois nestes casos ou a implementação do serviço (lógica de negócio) precisa ser modificada para tratar as falhas e interações com as réplicas ou as falhas são tratadas por código do *middleware* de TF. A primeira abordagem é indesejável, pois aumenta o nível de acoplamento entre a solução de TF e a implementação do serviço. No segundo caso, o desenvolvedor que consome o serviço precisa utilizar a API do *middleware* de TF, dificultando a adição de TF a serviços legados e, em alguns casos, reduzindo o nível de interoperabilidade das soluções, contrariando uma das premissas fundamentais de serviços web.

Por outro lado, as soluções transparentes de que temos conhecimento (ver Seção 4.4) não podem ser consideradas totalmente tolerantes a falhas. Nestes casos a transparência é alcançada com a inserção de um *proxy*, responsável pelas interações com as réplicas. Contudo, erros de hardware e software ainda podem ocorrer nos *proxies*, além de partições na rede. Assim, uma alternativa simples para atingir um maior nível de TF nas interações com um serviço é tornar o cliente ciente da replicação do serviço. Para isto, propomos a extensão de um padrão amplamente difundido [68, 24, 20]: o WS-Addressing, principalmente pelo fato de ser este o padrão responsável pelo endereçamento de *endpoints*[3]. A extensão permite a especificação de réplicas de um serviço e fica encapsulada no WS-Addressing. Por este motivo, pode ser considerada transparente ao cliente: o único componente a ser modificado na pilha de serviços web é o módulo responsável pelo processamento do WS-Addressing[4]. Uma vez que o código responsável pelo tratamento do WS-Addressing, tanto no cliente quanto no servidor, atenda à especificação do WS-Addressing em conjunto com a extensão proposta, será possível realizar o tratamento de falhas de maneira transparente, independentemente da plataforma de serviços web utilizada.

O restante deste texto é organizado da seguinte forma: uma breve introdução à TF em serviços web é fornecida no Capítulo 2. O Capítulo 3 apresenta os resumos dos trabalhos relacionados. No capítulo 4 é apresentada a primeira contribuição deste trabalho: uma extensão ao padrão WS-Addressing que permite a recuperação de falhas transparente ao cliente. A proposta de uma nova arquitetura de TF em serviços web, que chamamos de SimpleRep, é apresentada no Capítulo 5. Finalmente, o Capítulo 6 realiza as considerações finais em relação ao trabalho como um todo, as contribuições do projeto, as dificuldades encontradas e as sugestões de alguns trabalhos futuros.

[3]O WS-Addressing define um *endpoint* como sendo uma entidade, processador ou recurso referenciável ao qual mensagens podem ser endereçadas [62].

[4]Os padrões geralmente são implementados como módulos dos *frameworks* de serviços web.

Capítulo 2

Tolerância a Falhas em Serviços Web

Vários trabalhos que fornecem tolerância a falhas (TF) em Serviços Web já foram propostos. Neste capítulo serão apresentados os conceitos básicos de TF e de Serviços Web.

2.1 Tolerância a Falhas em Sistemas Distribuídos

Um sistema distribuído é um sistema em que componentes localizados em computadores em rede comunicam-se entre si e coordenam suas ações por meio da troca de mensagens. O compartilhamento de recursos é a principal motivação para a construção de sistemas distribuídos. Dentre os principais desafios encontrados durante a construção de sistemas distribuídos estão a heterogeneidade dos componentes, liberdade para a adição, remoção e substituição de componentes, segurança, escalabilidade, tratamento de falhas, concorrência e transparência [13].

Dentre todos os desafios apresentados pelos sistemas distribuídos, estamos interessados no problema de como um sistema distribuído pode continuar fornecendo um determinado serviço após a ocorrência de falhas em alguns de seus componentes.

Sistemas computacionais falham ocasionalmente. Quando uma falha ocorre, os serviços podem produzir resultados incorretos ou podem até mesmo parar antes de completar o seu processamento. Um sistema tolerante a falhas deve continuar funcionando, possivelmente de forma degradada, quando estas falhas ocorrerem. A degradação pode ser percebida tanto no desempenho quanto na qualidade do serviço, mas deve ser proporcional às falhas que tenham ocorrido [53].

Dentre as técnicas utilizadas para o tratamento de falhas podemos citar [13]:

- Detecção de falhas: alguns tipos de falhas podem ser detectados. Outros, entretanto, são mais difíceis de se detectar como, por exemplo, um servidor na Internet que tenha "quebrado". O desafio é manter o sistema funcionando na presença de falhas que não possam ser detectadas, mas que podem ser suspeitas.
- Mascaramento de falhas: algumas falhas podem ser escondidas ou pode ser possível

diminuir os seus danos. Uma mensagem pode ser retransmitida em caso de perda, por exemplo.

- Recuperação de falhas: envolve o desenvolvimento de sistemas capazes de armazenar os estados de dados permanentes, possibilitando a recuperação dos mesmos após uma falha ou então retroceder o sistema a um estado consistente (*roll-back*).

- Tolerância a falhas e Redundância: serviços podem se tornar tolerantes a falhas utilizando componentes redundantes (réplicas). O desenvolvimento de técnicas para a manutenção dos dados atualizados nas réplicas sem perda excessiva de desempenho é um desafio.

Em relação aos tempos de processamento e de troca de mensagens, os sistemas distribuídos podem ser classificados, basicamente, como sendo síncronos ou assíncronos [13]. Sistemas síncronos são aqueles em que limites bem definidos são impostos em relação aos tempos de processamento e de entrega de mensagens. Sistemas distribuídos assíncronos, por outro lado, são aqueles sem restrições quanto aos tempos de computação e de comunicação [13]. Pela ausência de restrições, o modelo de sistema assíncrono é considerado mais genérico, sendo que os algoritmos desenvolvidos neste modelo podem ser implementados nos outros modelos.

Existe também uma classe intemediária entre os sistemas síncronos e os assíncronos, que é a dos sistemas distribuídos parcialmente síncronos [34], em que os componentes possuem alguma informação sobre o tempo (não necessariamente exata). Por exemplo, processos em uma rede parcialmente síncrona poderiam ter acesso a relógios quase sincronizados, mas que se incrementassem à mesma taxa, ou poderiam ter limites aproximados para os tempos de computação ou de entrega de mensagens. Modelos parcialmente síncronos são provavelmente mais realísticos, pois os sistemas reais, incluindo-se a Internet, geralmente utilizam algum tipo de informação sobre o tempo.

A questão chave em tolerância a falhas é determinar se um processo participante de uma computação distribuída deve continuar esperando por um determinado evento ou se deve deixar de esperar por ele. Devido à falta de limites de tempo nos sistemas assíncronos, alguns problemas tornam-se muito difíceis, e até mesmo impossíveis, de serem resolvidos. Já foi provado, por exemplo, que o problema do Consenso[1] não pode ser resolvido em sistemas assíncronos em que ao menos um processo pode falhar [19]. Esta impossiblidade se deve ao fato de que, nestes sistemas, não é possível distinguir um processo que tenha "quebrado" de um processo muito lento. Assim, a habilidade de detectar as falhas de processos torna-se uma questão fundamental [15].

2.1.1 Modelo do Sistema

Este trabalho tem como foco sistemas distribuídos assíncronos em que processos falham apenas por "quebra", ou seja, quando param sua execução prematuramente e esta parada

[1]De maneira simplificada, o Consenso permite que processos cheguem a uma mesma decisão, dependente dos valores de entrada, mesmo podendo ocorrer falhas em alguns dos processos [11].

é permanente (modelo *crash-stop*). Partições na rede não são toleradas e os processos se comunicam por meio de canais de comunicação quasi-confiáveis, ou seja, canais que fornecem as seguintes garantias [15]:

- Se um processo correto p envia uma mensagem m para um processo correto q, então q, em algum momento futuro, receberá m.
- Mensagens não são duplicadas.
- Mensagens não podem ser corrompidas.
- Mensagens não podem ser geradas pelo canal.

Além disso, assume-se que as requisições feitas por um cliente a um servidor são bloqueantes, ou seja, uma nova requisição só é realizada após o término (recebimento da resposta) da anterior.

2.2 Serviços Web

Arquiteturas Orientadas a Serviços (SOAs[2]) [26, 43] estão em franca disseminação em quase todas as áreas da computação. A possibilidade de criar sistemas realmente independentes de plataformas e de linguagens de programação ("qualquer coisa pode falar com praticamente qualquer outra coisa" [8]) juntamente com a possibilidade de integração de sistemas de forma transparente fornecem grandes oportunidades para os desenvolvedores de soluções.

Do ponto de vista dos negócios, tendências como a terceirização de operações não críticas e a importância da reengenharia de processos foram determinantes para a adoção das SOAs. Do ponto de vista tecnológico, partindo do aprendizado com objetos distribuídos e de *middlewares* orientados a mensagens foi possível criar uma nova arquitetura fundada em padrões [68] visando à interoperabilidade.

Serviços web são baseados no modelo cliente/servidor estruturado para a utilização da maior quantidade possível de padrões. Já se pensa na utilização de serviços web para a padronização de comunicação com pequenos sensores e outros dispositivos como sistemas de entretenimento domésticos, cafeteiras e torradeiras, por exemplo [7].

Serviços web permitem interações automáticas entre computadores que, de outra forma, teriam que ser realizadas pelos usuários. Esta automatização, no entanto, introduz novos problemas na operação dos sistemas [38]:

- Um problema em um dos participantes de uma atividade distribuída pode ocasionar problemas em outros integrantes do sistema e prejudicar os relacionamentos entre o sistema e seus usuários.

[2] *Service Oriented Architectures.*

- Atividades que se baseiam na integração de vários sistemas apresentam desafios para confiabilidade, disponibilidade, consistência dos dados, concorrência, escalabilidade e segurança.
- Características como a escolha de fornecedor de serviço em tempo de execução, por exemplo, apesar de úteis, inserem novos problemas como garantir a autenticidade, a qualidade e o desempenho do fornecedor.

Apesar dos inúmeros benefícios, este modelo de construção de soluções ainda se encontra em seus estágios iniciais em alguns requisitos necessários à sua ampla adoção, dentre os principais a dependabilidade. Moser et al. [38] mostra que mesmo SOA's pouco complexas, com poucas camadas e com o envolvimento de poucos sistemas, possuem uma grande probabilidade de falhar quando sujeitas a cargas médias de serviço (100.000 atividades em um dia, por exemplo). Gilbert e Lynch [23] provam que é impossível que um serviço web ofereça consistência/atomicidade[3], disponibilidade[4] e tolerância a partições na rede[5] simultaneamente até mesmo em sistemas parcialmente síncronos, como a Internet.

Caso o sucesso das SOAs continue a aumentar, como é previsto [8, 37], e alcançar sistemas críticos em termos de dependabilidade, mecanismos que forneçam garantias de bom funcionamento destes sistemas deverão ser criados. Técnicas de segurança resolvem apenas parte dos problemas. Em caso de falha, apenas técnicas de replicação podem garantir acesso aos dados críticos, além de permitir uma reação de forma coordenada [8].

Birman et al. [8, 6] aponta razões para o uso de replicação em sistemas que devem ser altamente confiáveis, principalmente os baseados em serviços web. A replicação de serviços web sem o armazenamento de estado entre as requisições (*stateless*) é considerada simples de ser empregada, um exemplo é fornecido por Wah (ver Seção 3.2.1). Por outro lado, a replicação de serviços web que mantém o estado entre as requisições (*statefull*) requer a utilização de mecanismos mais sofisticados [41]. Birman [7] também destaca uma tendência nos novos trabalhos na área de sistemas distribuídos de propor soluções que realizam muitas trocas de mensagens, assumindo que o custo de troca de mensagens seja equivalente ao de processamento. Sabe-se, entretanto, que esta hipótese não se confirma nos sistemas reais, principalmente naqueles baseados em serviços web, devido aos vários protocolos presentes nas camadas de comunicação baseados em um padrão textual, a *eXtensible Markup Language* (XML) [66]. Assim, técnicas de replicação que utilizem o menor número de rodadas de comunicação e troca de mensagens são de especial interesse nas arquiteturas baseadas em serviços web.

[3]Operações parecem ser realizadas de uma só vez, em um único instante.

[4]Cada requisição recebida por um nó correto deve resultar em uma resposta.

[5]Em um sistema tolerante a partições na rede é permitida a perda de um número arbitrário de mensagens entre as partições.

Capítulo 3

Trabalhos Relacionados

Este capítulo apresenta os resumos dos trabalhos relacionados. Primeiramente, na Seção 3.1 são apresentados alguns trabalhos que utilizam o modelo de replicação ativa, o mesmo implementado, inicialmente, pelo SimpleRep (ver Capítulo 5). Em seguida, na Seção 3.2, são apresentados os demais trabalhos estudados durante a pesquisa.

3.1 Trabalhos que utilizam replicação ativa

3.1.1 A Middleware for Replicated Web Services

Ye e Shen [69] (2005) propõem um *middleware* para a replicação ativa de serviços web. Ao utilizar a comunicação em grupo para replicação, os autores argumentam que a utilização de *multicast* probabilístico (PBCAST [25]) permite uma maior escalabilidade da arquitetura e, portanto, resolveram utilizá-lo. Os autores vão mais longe e utilizam uma variante do PBCAST chamada TOPBCAST, implementada utilizando o JGroups [4], garantindo que, levando-se em conta as taxas de perda de mensagens e falhas dos servidores, todas as mensagens serão entregues aos seus destinos de forma totalmente ordenada.

Nesta arquitetura há duas entidades principais: uma *proxy* para os serviços web PWSS[1] e os serviços web replicados (WSS[2]). Assim como no WS-Replication [49] (ver Seção 3.1.2), os clientes acessam os serviços web através de uma *proxy* que é responsável pelo *multicast* das mensagens para as réplicas.

O *middleware* oferece também um pacote Java chamado RWS que fornece classes que permitem a interação entre um cliente e as PWSSs.

Cada PWSS possui 2 módulos, um manipulador de mensagens (MH) responsável pelo recebimento de requisições, envio de respostas e tratamento de erros dos WSSs; e um gerenciador de comunicação em grupo (GCM), responsável pela distribuição das requisições (usando TOPBCAST) e pela detecção de falhas.

[1] *Proxy Web Service Site.*
[2] *Web Service Site.*

Uma falha em uma PWSS deve ser tratada pelo cliente. O pacote RWS inclui classes que tratam as falhas nas PWSSs automaticamente. Os programadores precisam apenas especificar os endereços dos PWSSs em um arquivo de configuração.

O problema de serviços web replicados implementados utilizando *threads*[3] é contornado através da utilização de outro pacote, chamado MWS, fornecido pelos autores.

Apenas testes em uma LAN foram fornecidos sem uma descrição dos serviços sendo fornecidos. É possível observar que a sobrecarga gerada a partir da utilização do *middleware* para a replicação de serviços sem *threads*, com tempos de processamento curtos, é grande, e diminui conforme o tempo de processamento dos serviços aumenta. Os experimentos realizados com serviços web com múltiplas *threads* mostram que a sobrecarga é fortemente dependente do número de réplicas, sendo que a utilização de mais que quatro réplicas passa a gerar uma sobrecarga acima do aceitável (sempre acima de 20%), na opinião dos autores.

3.1.2 WS-Replication

Salas et al. [49] (2006) criaram o WS-Replication, um framework para a replicação de serviços web em WANs constituído por dois componentes principais: um componente de replicação e um componente de multicast confiável (WS-Multicast[4]). O serviço de WS-Multicast fornece uma implementação de comunicação em grupo utilizando apenas SOAP.

O componente de replicação de serviços web consiste em:

- um instalador (*deployer*) de serviços web: fornece um serviço de instalação de serviços web nas réplicas a partir de um servidor central. Isto é feito através do envio de pacotes (assim como os arquivos .jar para Java) que contém todos os dados necessários para a instalação dos serviços web.
- um gerador de *proxies*: gera um *proxy* entre o cliente e o despachante para cada operação suportada pelo serviço web.
- despachante de serviços web: (na ida) recebe as invocações do *proxy*, envia-as para o WS-Multicast e espera a resposta das réplicas; (na volta) recebe as respostas das réplicas e gera uma resposta para o *proxy* que repassa a mesma para o cliente.

As interações entre um cliente e o *framework* são ilustradas na Figura 3.1.

Osrael et al. [41] citam o WS-Multicast (juntamente com o JGroups e o Spread [2]) como sendo uma plataforma no estado da arte para comunicação em grupo. Entretanto, Osrael et al. consideram que a única diferença relevante no WS-Multicast é o fornecimento de uma interface WSDL.

[3]Duas *threads* que compartilhem algum dado para o fornecimento de um serviço web devem ser executadas na mesma ordem nas réplicas.

[4]Sob licença fechada.

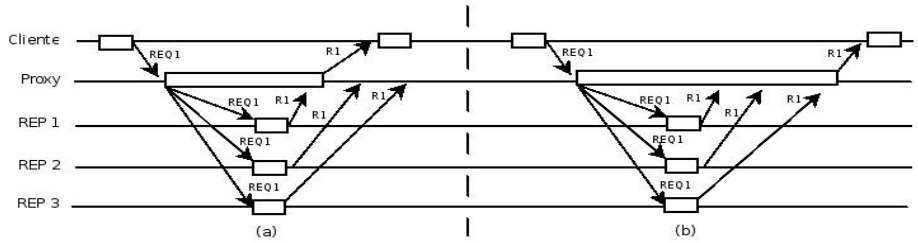


Figura 3.1: (a) Cliente acessando um serviço replicado através do WS-Replication no modo "primeira resposta", em que a resposta (R1) à requisição (REQ1) enviada ao cliente é a primeira resposta recebida por qualquer uma das réplicas. (b) Cliente acessa serviço replicado através do WS-Replication em modo "todas as respostas", em que a *proxy* espera pelas respostas de todas as réplicas ativas (com um pós-processamento opcional) para então retornar a resposta ao cliente.

Junior et al. [28] criticam o WS-Replication por requerer a instalação dos seus componentes (WS-Multicast) nas réplicas, não disponibilizando transparência de replicação quando acessa provedores web padrões. O trabalho de Junior et al. é estudado na Seção 3.2.7 e sugere a utilização de adaptadores para possibilitar a utilização de serviços web de outros fornecedores.

3.1.3 Middleware para replicação de Serviços Web baseado em Axis2

Osrael et al. [42] (2007) apresentam um *middleware* para replicação de serviços web, um módulo[5] do framework para SOAP Axis2 [46, 56], através da utilização da Comunicação em Grupo fornecida pelo *toolkit* Spread [2].

Há quatro componentes responsáveis pelo funcionamento do *middleware*:

- *Invocation Service*: responsável pelo recebimento e distribuição das requisições.
- *Replication Manager*: responsável pela manutenção do(s) grupo(s), baseando-se principalmente nas mensagens geradas pelo Spread. Em caso de partição na rede, apenas a partição com o serviço primário continua oferecendo o serviço. A sincronização dos estados de serviços novos ou que venham a se recuperar de falhas é feita através da transferência de estado[6] de um serviço que já esteja no grupo para o integrante que esteja ingressando no grupo.
- *Group Communication System*: primitivas de multicast confiável para os grupos e monitoramento dos membros do grupo são fornecidas pelo Spread.
- *Replication Protocol*: responsável pela propagação das atualizações da réplica primária. Os autores propõem a utilização de replicação passiva quente, permitindo

[5] A implementação de serviços utilizando o Axis2 é feita através de módulos.

[6] A outra alternativa seria armazenar a sequência de requisições em mídia persistente para posterior reexecução das mesmas.

que as réplicas realizem apenas processamento determinístico, eliminando uma das principais vantagens da replicação passiva.

A Figura 3.2 ilustra o funcionamento da arquitetura.

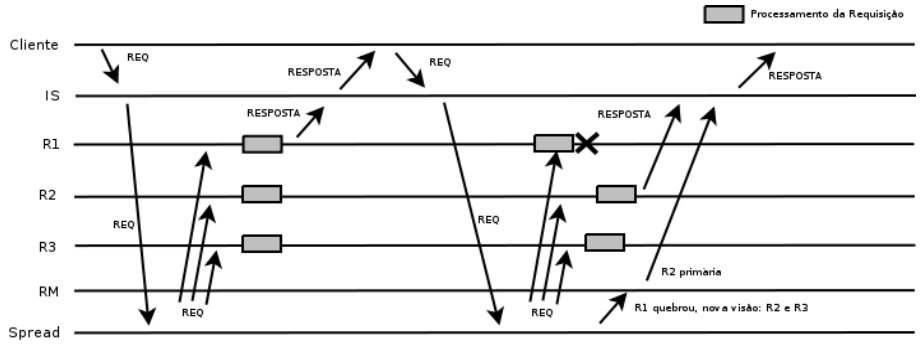


Figura 3.2: Cliente invoca o serviço através do *Invocation Service* (IS), que distribui a requisição utilizando o Spread. Quando a réplica primária R1 "quebra", o Replication Manager (RM) é notificado sobre uma nova visão do grupo de réplicas e notifica o IS, tornando R2 a nova réplica primária.

Os testes de desempenho mostraram que a sobrecarga é pequena quando há poucas réplicas e que a escalabilidade da arquitetura é diretamente dependente da escalabilidade do Spread.

3.1.4 Thema

Merideth et al [35] (2005) criaram um *middleware* para o fornecimento de tolerância a falhas bizantinas [30] (TFB) em serviços web chamado Thema. A tolerância a falhas é obtida através de um conjunto de bibliotecas que funciona utilizando a abordagem de interceptação das requisições.

O conjunto de bibliotecas do Thema inclui uma biblioteca para o cliente (Thema-C2RS) que permite o acesso de um cliente a um serviço web TFB, uma biblioteca para o servidor (Thema-RS) concebida para facilitar a criação de serviços web TFB e uma biblioteca para acesso a serviços web externos (Thema-US) de maneira segura. As bibliotecas do Thema gerenciam a comunicação entre o SOAP e as bibliotecas TFB.

O Thema foi criado através da extensão do sistema BASE [10], que fornece bibliotecas cliente-servidor para TFB, e dos *middlewares* para SOAP gSOAP [17] (C++) e Apache Axis [59] (Java), para fornecer a interface via serviços web.

Pontos contra deste *middleware* são o grande número de réplicas e a infra-estrutura necessários para o seu funcionamento, reduzindo a interoperabilidade do sistema.

3.1.5 FTWeb

Santos, Lung e Montez [51] (2005) oferecem um modelo que fornece tolerância a falhas transparente através de replicação ativa inspirado no FT-CORBA [40]. Os testes de desempenho do FTWeb, entretanto, não foram muito promissores, chegando a apresentar, em soluções com 4 réplicas, acréscimos entre 40% e 60% no tempo de resposta.

3.2 Trabalhos que não utilizam replicação ativa

3.2.1 Reliable Web Services by Fault Tolerant Techniques

Wah [67] (2006) apresenta uma arquitetura simples, baseada em um Gerenciador de Réplicas (RM) responsável por:

- Instanciar as réplicas de um serviço web.
- Escolher a melhor réplica para ser a primária.
- Registrar a WSDL que define o serviço web no servidor UDDI[7].
- Monitorar a réplica primária quanto à disponibilidade e selecionar outra réplica quando a primária falhar. A transição de um réplica para outra é feita através da simples troca da WSDL antiga pela nova no servidor UDDI.

A Figura 3.3 ilustra o funcionamento desta arquitetura.

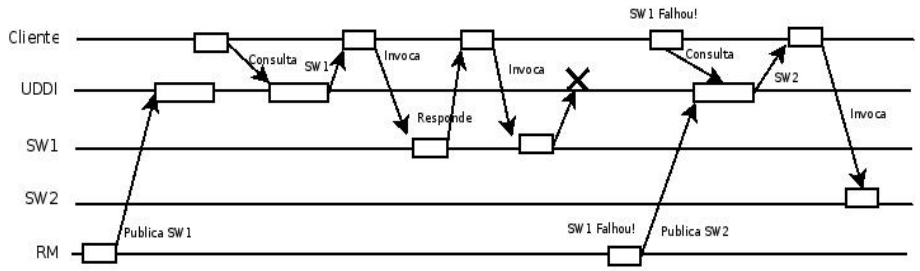


Figura 3.3: Arquitetura simples para tolerância a falhas em serviços web sem armazenamento de estado entre as interações. Ao identificar a falha no serviço web 1 (SW1) o cliente realiza uma nova consulta ao servidor UDDI que deverá retornar a nova WSDL de outra réplica (SW2), inserida no servidor pelo Gerenciador de Réplicas no momento em que o mesmo identifica a falha de SW1.

Esta arquitetura é de pouca aplicabilidade, pois só permite a replicação de serviços web em que não haja armazenamento de estado.

[7] *Universal Description, Discovery and Integration.* Padrão criado para permitir a descoberta e invocação dinâmica de serviços web.

3.2.2 FT-CORBA

Algumas soluções apresentadas neste trabalho são baseadas no FT-CORBA [40], padrão da OMG[8] para tolerância a falhas em arquiteturas CORBA. Os objetos básicos do FT-CORBA são:

- Serviço de Gerenciamento de Réplicas (RMS): responsável pela criação e remoção das réplicas no sistema. Além disso, possui uma interface para configuração das propriedades de tolerância a falhas da solução.
- Serviço de Gerenciamento de Falhas: fornece interfaces para monitoramento das réplicas e notificação de falhas ao RMS.
- Serviço de Registro e Recuperação: presente em cada uma das réplicas, é responsável pelo registro das requisições recebidas e pela recuperação das réplicas em caso de falha.

Dentre os principais desafios encontrados ao criar o FT-CORBA, os autores citam o problema de fazer referência a um grupo de réplicas de forma transparente, o qual só é possível através de alterações no código do *middleware* (ORB) ou através de interceptação de chamadas. Este problema também ocorre com os serviços web, pois não há padrão que estabeleça um meio para a representação de um grupo de réplicas.

3.2.3 FT-SOAP

Fang et al. [18] sugere uma arquitetura baseada no modelo *crash-stop* e com replicação passiva utilizando a abordagem de serviço. A tolerância a falhas é fornecida através de quatro componentes: um Gerenciador de Réplicas (RM), um detector de falhas, um notificador de falhas e um mecanismo de registro de mensagens e recuperação.

A proposta é baseada no FT-CORBA [40] com dois objetivos principais: estabelecer um padrão para serviços baseados em SOAP tolerantes a falhas e implementar um protótipo de FT-SOAP. Além disso foi necessária a inserção de uma nova *tag Web Service Group* (`<WSG/>`) na WSDL. Este elemento é o responsável pela descrição das réplicas para os clientes FT-SOAP, permitindo que os clientes possam selecionar serviços secundários em caso de falha da réplica primária. Segundo os autores esta abordagem é transparente, pois clientes que não utilizem FT-SOAP poderão continuar utilizando a réplica primária enquanto não houver falha.

O gerenciador de réplicas basicamente oferece interfaces para a definição de propriedades da replicação (tipo, número inicial de réplicas, etc), criação e manutenção de grupos de serviços e, por fim, publicação da WSDL gerada no servidor UDDI.

Os testes de desempenho realizados pelos autores mostraram que mesmo para uma detecção de falhas agressiva (intervalos de 20s) a sobrecarga de processamento gerada foi

[8] *Object Management Group* - Consórcio para a padronização de soluções baseadas em CORBA.

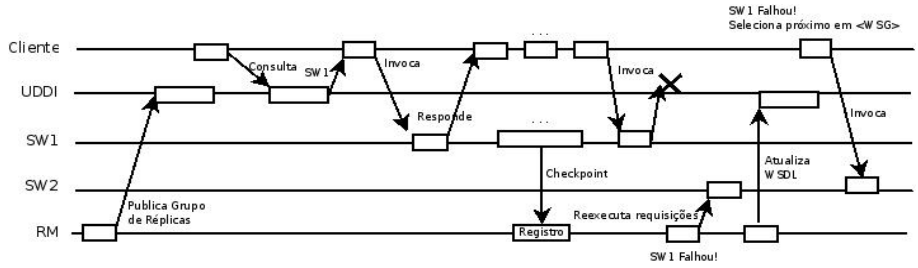


Figura 3.4: Principais interações entre o cliente, as réplicas e o gerenciador de réplicas do FT-SOAP. Note que o registro ilustrado na figura não é feito pelo RM propriamente dito, mas sim pelo componente de registro de requisições.

de apenas 3,6% para um grupo de 100 serviços sendo oferecidos. Para mensagens de até 512 KB a sobrecarga para a realização dos *checkpoints* e de *logging* também se mostrou pequena.

Osrael et al. [42] apontaram que este *middleware* tem a deficiência de ser baseado em versões antigas dos padrões para serviços web. Além disso, uma versão para clientes do FT-SOAP deve ser instalada nos mesmos.

3.2.4 FAWS

Jayasinghe [27] descreve a arquitetura FAWS[9] que fornece tolerância a falhas transparente ao cliente. O autor também menciona a possibilidade de oferecer replicação não transparente (o cliente receberia uma lista de servidores e, sempre que o mesmo recebesse um pacote de erro SOAP ou um erro dos meios de transporte (TCP e HTTP), passaria a enviar requisições para o próximo servidor da lista) mas não fornece detalhes de como essa abordagem pode ser utilizada através do FAWS.

O FAWS é baseado em uma arquitetura de objetos distribuídos constituída por quatro componentes:

- *FT-Front*: funciona como um *proxy* que recebe as requisições dos clientes, registrando as mesmas para uma eventual retransmissão em caso de falha na réplica primária. O *FT-Front* também é capaz de detectar uma falha na réplica primária, permitindo, assim, o funcionamento correto do sistema mesmo em caso de falha do *FT-Detector*.
- *FT-Admin*: comunica-se constantemente com o *FT-Detector* e com o *FT-Front* para manter contínuo o fornecimento dos serviços. É responsável também pela configuração do sistema.
- *FT-Detector*: é o detector de falhas da arquitetura. Basicamente envia pacotes para a porta do serviço na réplica primária para verificar se a mesma está ativa e envia

[9] *FAult tolerance for Web Services.*

pacotes *ICMP Echo Request* (*ping*) para todas as máquinas para verificar se houve alguma falha de hardware.

- *FT-Monitor*: é um subcomponente do *FT-Admin*, responsável pelo armazenamento do estado corrente das réplicas.

Osrael et al. [42] não consideram o FAWS uma arquitetura tolerante a falhas, pois não é feita replicação dos componentes do *middleware*. Apenas a falha do componente FT-Detector pode ser tolerada.

3.2.5 ADAPT

Bartoli et al. [5, 61] apresentam um *framework* para a composição de serviços web chamada ADAPT. Os autores começam por separar os serviços web em dois grupos: serviços básicos (BS[10]), que não dependem de outros serviços para funcionar, e serviços compostos (CS[11]), disponíveis através da composição de dois ou mais serviços básicos e/ou compostos. Não são feitas quaisquer outras distinções entre os serviços. Se um serviço pode ser descrito através de WSDL, ele pode ser usado no ADAPT.

Os autores também desenvolveram uma *linguagem para especificação de serviços*, responsável pela descrição (opcional) de atributos não-funcionais dos serviços, como as capacidades transacionais, sequência de invocações permitidas pelos serviços e atributos de desempenho; um esquema XML para composição de serviços também é oferecido para a definição de *workflows* dentro da aplicação, juntamente com um editor visual para a composição de serviços que, entre outras funcionalidades, verifica se a composição possui *deadlocks* e *loops* infinitos, além de um depurador de execução.

3.2.6 DeW

Alwagait e Ghandeharizadeh apresentam o *framework* DeW[12] em 2004 [1], construído com a finalidade de fornecer serviços web independentes de localização física, ou seja, que forneçam resultados de maneira correta enquanto houver ao menos uma réplica disponível.

O que é proposto pelos autores é a criação do que eles chamaram de *registro DeW*, que desempenha o papel de um servidor para gerenciamento de exceções, ao qual os serviços web publicariam suas exceções e seus *handlers*[13] e de onde as *proxies* destes serviços web poderiam obter os *handlers* para tratar as exceções capturadas. Deste modo, o problema da migração poderia ser tratado publicando-se uma exceção do tipo "ServicoWebNaoEncontrado", por exemplo, e o seu *handler* em um registro DeW. Neste novo cenário, quando a *proxy* recebesse a exceção "ServicoWebNaoEncontrado" de N1, bastaria contactar o registro DeW e obter o *handler* para esta exceção, no caso, o endereço de N2.

[10] *Basic Service*.
[11] *Composite Service*.
[12] *Dependable Web Services*.
[13] Código, dados ou ambos.

Para tornar o registro DeW possível, são necessárias algumas alterações na estrutura de uma solução:

- O acesso aos serviços Web deve ser feito através das *DewProxies*:
  - Micro *proxies*: específicas para cada aplicação. Implementam a funcionalidade necessária para o acesso a um serviço remoto.
  - Macro *proxies*: monitoram o funcionamento das micro *proxies* através da interceptação das invocações a uma micro *proxy*, instanciam temporizadores para detecção de falhas e, quando necessário, disparam exceções, recebendo os *handlers* de um registro DeW e os utilizam para se recuperar de uma falha.
- Utilização de *DeWSDL*, uma extensão da WSDL, utilizada para gerar as *proxies* dos serviços web. A DeWSDL adiciona duas *tags* à WSDL: `<exceptionlist>` e `<dewexception>` responsáveis pela descrição de uma lista de exceções por estouro de tempo de espera[14] para cada operação fornecida pelo serviço web. Esta alteração na WSDL não afeta o funcionamento dos aplicativos legados, geralmente geradores de *proxies* a partir de documentos WSDL.
- Alta disponibilidade e desempenho da estrutura de registros DeW, para permitir uma boa escalabilidade das soluções. O problema da centralização dos *handlers* em um registro DeW é solucionado pelos autores através da utilização de tabelas de dispersão distribuídas [7] ou através de anycast [44].
- Presença de uma infra-estrutura de chaves públicas, necessária para verificação da integridade dos *handlers*.

A Figura 3.5 ilustra o funcionamento do DeW.

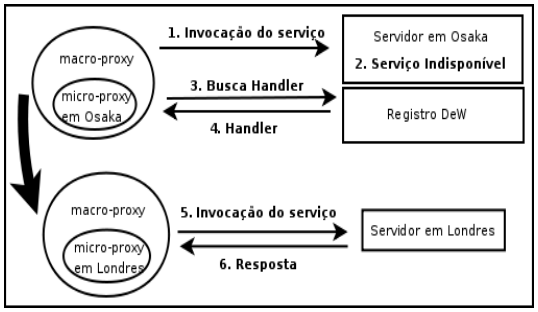


Figura 3.5: Interações entre a *macro-proxy* e o registro DeW para o tratamento de uma falha de um servidor em Osaka (baseada nas figuras de Alwagait e Ghandeharizadeh [1]).

A estrutura do *middleware* causa fortes impactos na construção de um solução. Além da infra-estrutura necessária para o funcionamento do mesmo, é necessário que os desenvolvedores da solução prevejam as exceções possíveis de serem disparadas, reduzindo a transparência da mesma. Os autores não forneceram qualquer tipo de resultado em relação ao desempenho da arquitetura.

[14] As exceções por resultados inesperados já são suportadas pelo elemento `<fault>` da WSDL.

3.2.7 Smart Proxies para o uso de Web Services Replicados

A ferramenta SmartWS [28] (2006) fornece recursos para replicação heterogênea de serviços web através de *Smart Proxies*, que nada mais são que *proxies* que possuem a habilidade de selecionar os servidores a serem acessados pelos clientes. Isso é feito através de uma ferramenta que gera as *proxies* a partir de uma especificação em uma linguagem criada pelos autores.

A SmartWS fornece recursos para:

- definição das interfaces e da localização dos servidores que disponibilizam um determinado serviço web;
- definição das políticas que serão utilizadas pelos clientes para seleção dos servidores especificados no item anterior. As políticas suportadas são:
  - estática: definida pelo programador;
  - aleatória: o servidor que fornecerá o serviço web é escolhido de forma aleatória;
  - paralela: todos os servidores atendem a todas as requisições. A resposta entregue ao cliente é aquela que for recebida primeiro.
  - melhor mediana: calcula a mediana entre as k últimas invocações de um serviço e invoca o serviço no servidor que possuir a menor mediana.
  - PBM (*Parallel Best Median*): combina aspectos da seleção por melhor mediana com a seleção em paralelo. Neste modelo são invocadas em paralelo todas as réplicas que possuam medianas menores ou iguais a $k * m$, limitadas a um número máximo de p invocações, sendo m a menor das medianas e p e k constantes arbitrárias que determinam o grau de paralelismo das invocações.

  Silva e Mendonça [54] oferecem um estudo empírico sobre o desempenho das políticas de seleção de serviços Web descritas acima. Os autores realizaram experimentos com a arquitetura demonstrando os cenários em que as políticas de seleção de réplicas são bem sucedidas.
- definição de objetos **adaptadores**, responsáveis pela conversão das interfaces providas por um servidor para a interface requerida pelo cliente. Esta funcionalidade permite a utilização de réplicas heterogêneas em relação às interfaces.
- definição de um protocolo de invocação, isto é, um conjunto de regras que determinem as sequências válidas de chamadas de métodos definidos na interface de um serviço web. As invocações de operações que alterem o estado de um serviço web, por exemplo, devem ser feitas em sequência para o mesmo servidor até que se inicie uma nova sequência de invocações independente de estado.

A Figura 3.6 fornece um esquema da invocação de um serviço através da SmartWS.

Apesar da flexibilidade da arquitetura, a *Smart Proxy* ainda constitui um ponto único de falha. Além disso, um alto nível de colaboração entre os fornecedores de serviços

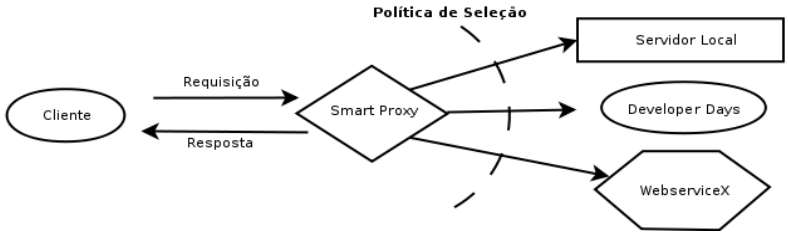


Figura 3.6: Replicação heterogênea de serviços web através de *Smart Proxies*. O cliente conhece apenas a WSDL da *Smart Proxy*, ficando a cargo desta adaptar as invocações do cliente para as interfaces WSDL dos serviços externos (DeveloperDays e WebserviceX).

terceirizados seria necessário para possibilitar a replicação de serviços *stateful*, tanto para armazenamento de estado no servidor quanto no cliente.

3.2.8 NaradaBrokering

Fox, Pallickara e Parastatidis [22] (2004) descreveram como foi o processo de extensão do *middleware NaradaBrokering* para fornecer uma interface de serviços web e como esta extensão pode beneficiar uma solução baseada em serviços web.

O *NaradaBrokering* [47] é um *middleware* orientado a mensagens que fornece um *framework* de notificações que entrega mensagens de uma origem para um conjunto de consumidores de maneira eficiente. Dentre as principais funcionalidades oferecidas pelo *NaradaBrokering* podem ser citadas: criptografia e ordenação de mensagens, compressão, entrega confiável de mensagens, abstração de protocolo de transporte e endereçamento, além de funcionalidades para tolerância a falhas e qualidade de serviço.

Dentre as principais contribuições da nova arquitetura pode ser destacada a implementação da *WS-ReliableMessaging* e a previsão de implementação da *WS-Reliability* (ver a Seção **??**) juntamente com a federação[15] entre os dois padrões.

3.2.9 FT-GRID

Townend et al. [60] (2005) criaram o FT-GRID, ferramenta que facilita a utilização de SOAs para a construção de arquiteturas baseadas em múltiplas versões (MVD[16]), também conhecidas como arquiteturas *n-Version*, aproveitando-se da grande possibilidade de fornecimento de implementações diferentes para uma mesma funcionalidade.

Arquiteturas *n-Version* utilizam vários serviços com funcionalidades equivalentes mas implementados de maneiras diferentes para, através do consenso entre as respostas, fornecer um nível satisfatório de tolerância a falhas. A efetividade do método se baseia no fato de que, apesar da grande possibilidade de presença de falhas em cada uma das versões, é também grande a possibilidade de que as falhas estejam localizadas em posições diferentes

[15] O termo federação é encontrado na literatura de serviços web significando cooperação ou integração.
[16] *Multi-version design.*

no código de cada uma delas. Deste modo, espera-se que o mecanismo de consenso entre as respostas descarte as respostas incorretas. Quando não é possível obter consenso, o usuário do sistema deve ser notificado.

Os autores realizaram testes de injeção de falhas e de atrasos, utilizando a ferramenta WS-FIT [33], que mostraram a inserção de pouco atraso devido à utilização da ferramenta. Além disso, os ganhos em confiabilidade no caso de injeção de falhas e em tolerância a atrasos (utilizando mais réplicas que o necessário) foram satisfatórios.

A maior deficiência do modelo *n-Version* é a possibilidade de ocorrer uma *falha por caminho comum*[17] (CMF). Uma CMF ocorre quando dois ou mais serviços utilizam ao menos um serviço em comum, podendo, em casos extremos, acontecer de um serviço muito popular ser utilizado por todas as réplicas. Nestas situações o modelo *n-Version* é de pouca utilidade, pois não será possível obter consenso nos casos de falha dos serviços em comum. Para contornar o caso das CMFs foi desenvolvida a técnica de *procedência dos dados*[18], que consiste basicamente na documentação do processo realizado para a geração da informação. Townend et al. escolheram a ferramenta PreServ [45], que deve ser instalada em todos os *hosts* participantes da solução, para obter a procedência dos dados. Juntamente com a PreServ foi utilizado um mecanismo de atribuição de pesos aos serviços web de acordo com a quantidade de vezes em que os mesmos retornam uma resposta que faça parte do consenso, permitindo, então, que o FT-GRID possa determinar serviços web que possuam procedência semelhante.

Para os testes finais foram utilizadas 5 réplicas de um serviço web, cada uma hospedada em um servidor Tomcat separado, implementadas com o apache Axis + PreServ. Cada uma destas réplicas precisa invocar 2 outros serviços web (o primeiro com 4 réplicas e o segundo com 3 réplicas) escolhidos aleatoriamente em tempo de execução.

Através deste artigo foi possível observar a grande oportunidade oferecida pelas SOAs para a construção de soluções altamente confiáveis através do modelo *n-Version*. Foi possível observar também o problema das CMFs em SOAs e como a utilização da procedência dos dados pode ser útil na sua resolução.

3.2.10 WS-FTM

Looker e Munro [32] (2005) criaram a WS-FTM, uma arquitetura baseada no FT-GRID (versão sem a verificação da procedência dos dados).

O trabalho basicamente descreve alguns detalhes de implementação do WS-FTM, sendo que o embasamento teórico é o mesmo fornecido pelo trabalho de Townend et al. [60]. Os autores, entretanto, dão um enfoque maior à tolerância a falhas bizantinas, possível através da utilização de $3f + 1$ versões do serviço web para tolerar f falhas.

Dentre as fraquezas desta arquitetura podem ser apontadas a restrição de sua utilização apenas em soluções feitas em Java e a necessidade de implementações diferentes para o

[17] Tradução livre do termo *common-mode failure*.
[18] Citada no texto como *provenance*.

mesmo serviço (inerente ao modelo *n-Version).*

A Figura 3.7 ilustra o funcionamento do WS-FTM.

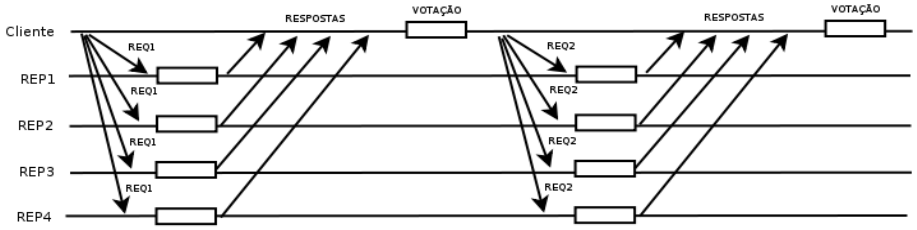


Figura 3.7: Invocação de serviços de acordo com o modelo *N-Version*. O cliente invoca várias implementações diferentes do mesmo serviço e assume como resposta aquela que tiver sido obtida o maior número de vezes (votação).

3.2.11 Using WS-BPEL to Implement Software Fault Tolerance for Web Services

Dobson [16] (2006) apresenta um mapeamento de alguns métodos de tolerância a falhas para WS-BPEL[19], visando à construção de um mecanismo extensível que permita a adição de outros métodos pelo desenvolvedor além de sua rápida configuração por usuários não avançados.

A WS-BPEL é um padrão OASIS que define uma linguagem XML utilizada para a composição de serviços web através de *workflows*. O desenvolvedor define a sequência de execução de serviços web através da sintaxe fornecida pelo padrão e a execução do mesmo fica a cargo de um mecanismo de execução[20] WS-BPEL comercial ou de código livre como o Apache ODE [58].

Os autores não fornecem testes de desempenho e nem de melhorias em relação à dependabilidade da nova solução. A principal vantagem apontada é a possibilidade de tornar um processo de execução inteiro tolerante a falhas. Outra diferença importante entre esta solução e maioria das outras apresentadas é a ausência de elementos específicos no servidor de aplicativos, ficando a lógica de tolerância a falhas no mecanismo de execução do WS-BPEL que, por outro lado, necessita de melhorias em relação à dependabilidade para não se tornar um ponto único de falha.

3.2.12 Aumento da resiliência dos Web services com uma Infraestrutura Peer-to-Peer

Correia e Cardoso [12] (2005) descrevem uma arquitetura baseada em grupos de réplicas mantidos através do modelo Peer-to-Peer (P2P), em que qualquer nó da rede pode dispo-

[19] *Web Services Business Process Execution Language*
[20] WS-BPEL *Engine*.

nibilizar serviços. Os serviços web foram implementados utilizando Tomcat/Axis 1.1 e a rede P2P foi construída através da plataforma JXTA [55] da Sun Microsystems.

Como resultados de testes de desempenho os autores afirmam que a taxa de crescimento do número de mensagens trocadas entre os *peers* é linear em relação ao número de *peers* em um grupo de réplicas (os testes foram realizados apenas com um grupo). O tempo de resposta médio em uma LAN com 12 *peers* foi de 0,5 segundos, mas podendo chegar a vários segundos no caso de falhas.

3.2.13 Resumo dos trabalhos

Um resumo das características dos principais trabalhos estudados é fornecido pela Tabela 3.1.

	TR	REP	TF	Des.	Extras	Prós	Contras
Ye & Shen (2005)	S	A	*crash-stop*	Aceitável até 4 réplicas	Utilizam JGroups.	TOPBCAST no GCS; Acesso síncrono e assíncrono; suporta multi-threading;	É necessário utilizar uma biblioteca Java fornecida pelos autores; desempenho.
WS-Replication	S	A	*crash-stop*	Bom	GCS JGroups.	WS-Multicast;	Requer a instalação dos componentes em cada uma das réplicas.
Osrael et al. (07)	S	SA	*crash-stop* / GMP	Bom com poucas réplicas	GCS Spread.	Replicação SA	Funciona apenas com soluções baseadas em Axis2.
FT-SOAP	N	P	*crash-stop*	Muito Bom	Baseado no FT-CORBA	-	Modificaram a WSDL; necessita de interceptadores no cliente; baseado em padrões antigos.
FAWS	s	PF	*crash-stop*	NF	RMI para comunicação interna	Simplicidade	Não replica os componentes da arquitetura
Adapt	S	NA	-	-	-	-	-
Thema	N	A	bizantinas	NF	-	Fornece um conjunto grande de bibliotecas.	Precisa de muitas réplicas; muita infra-estrutura.
FTWeb	S	A	-	Muito ruim	Utiliza domínios de serviço no lugar de um GCS; baseado no FT-CORBA	-	Desempenho
DeW	S	NA	NF	-	-	Suporta migração de serviços de forma transparente; arquitetura inovadora.	Muita infra-estrutura; necessidade de um registro DeW; altera o processo de construção da solução.
SmartWS	S	H	*crash-stop*	NF	Linguagem para construção das *proxies*.	Replicação Heterogênea.	Funciona apenas com Axis.
NaradaBrokering	N	NA	NF	-	Extensão do NaradaBrokering.	Baseado em um *framework* sólido.	Muita infra-estrutura.
FT-GRID	S	H	bizantinas	NF	Baseado no modelo *n-Version*	Bons resultados em relação ao aumento da dependabilidade; trata o problema das CMFs.	Desenvolvimento de réplicas heterogêneas.

Tabela 3.1: Comparativo entre os trabalhos de tolerância a falhas em serviços web. Legenda: TR = transparente para o cliente, REP = tipo de replicação, NA = não se aplica, NF = não fornecido(a), A = Ativa, P = Passiva, PF = Passiva Fria, SA = Semi-Ativa, H = Heterogênea, GMP = *Group Membership* Particionável.

Capítulo 4

Recuperação Transparente ao Cliente via WS-Addressing

4.1 Introdução

A maioria das arquiteturas de tolerância a falhas (TF) não pode ser considerada transparente ao cliente. Isto pode ser dito porque ou a implementação do serviço (lógica de negócio) precisa ser modificada para tratar as falhas e interações com as réplicas ou a recuperação de falhas é tratada por código (no lado do cliente) do *middleware* de TF. A primeira abordagem é indesejável, pois aumenta o nível de acoplamento entre a solução de TF e a implementação do serviço. No segundo caso, o desenvolvedor que consome o serviço precisa utilizar a API do *middleware* de TF, dificultando a adição de TF a serviços legados e, em alguns casos, reduzindo o nível de interoperabilidade das soluções, contrariando uma das premissas fundamentais de serviços web.

Por outro lado, as soluções transparentes de que temos conhecimento (ver Seção 4.4) não podem ser consideradas totalmente tolerantes a falhas. Nestes casos, a transparência é alcançada ao se inserir um *proxy*, responsável pelas interações com as réplicas. Contudo, erros de hardware e software ainda podem ocorrer nos *proxies*, além de partições na rede. Assim, uma alternativa simples para atingir um maior nível de TF nas interações com um serviço é tornar o cliente ciente da replicação do mesmo.

Para que seja possível uma recuperação transparente ao cliente, propomos a extensão de um padrão amplamente difundido [68, 24, 20]: o WS-Addressing, principalmente pelo fato de ser este o padrão responsável pelo endereçamento de *endpoints*. A extensão permite a especificação de réplicas de um serviço e fica encapsulada no WS-Addressing. Por este motivo, pode ser considerada transparente ao cliente: o único componente a ser modificado na pilha de serviços web é o módulo responsável pelo processamento do WS-Addressing[1]. Uma vez que o código responsável pelo tratamento do WS-Addressing, tanto no cliente quanto no servidor, atenda à especificação do WS-Addressing em conjunto com a extensão proposta, será possível realizar a recuperação de uma falha de maneira transparente,

[1]Os padrões geralmente são implementados como módulos dos *frameworks* de serviços web.

independentemente da plataforma de serviços web utilizada. A proposta deste capítulo resultou na publicação de um artigo no Workshop de Testes e Tolerância a Falhas (WTF 2009) [14].

O restante deste capítulo é organizado da seguinte maneira: a Seção 4.2 descreve a extensão ao WS-Addressing. A Seção 4.3 detalha a criação do protótipo e sua avaliação. A Seção 4.4 revisita algumas propostas de TF, correlacionando-as com a extensão e, por fim, a Seção 4.5 conclui o capítulo e fornece algumas sugestões de trabalhos futuros.

4.2 Estendendo o WS-Addressing

O WS-Addressing é um padrão que visa a fornecer um mecanismo de endereçamento de serviços web independente de protocolo de transporte (HTTP, SMTP, JMS, etc). O padrão define um conjunto de propriedades que são utilizadas para referenciar serviços web e facilitar o endereçamento de *endpoints* nas mensagens [62]. A criação do WS-Addressing foi necessária porque o SOAP não define uma maneira padrão de se especificar o destino de uma mensagem, como retornar uma resposta e nem para onde enviar mensagens de erro. Antes da criação do WS-Addressing, todas estas informações eram delegadas ao protocolo na camada de transporte (o HTTP, por exemplo). Esta Seção define como o WS-Addressing pode ser estendido para tornar a especificação de réplicas e o tratamento de falhas possíveis.

4.2.1 Mensagens de erro relevantes definidas pelo WS-Addressing

Além de ser o padrão responsável pela definição dos mecanismos de endereçamento de *endpoints*, é o WS-Addressing que define, também, as duas mensagens de erro pertinentes à TF [63]:

- *Destination Unreachable*: recebida pelo consumidor do serviço quando o *endpoint* de destino não pode ser alcançado por meio da rede.
- *Endpoint Unavailable*: enviada ao consumidor quando o fornecedor não é capaz de processar a requisição devido a uma falha temporária ou permanente. Opcionalmente, pode-se inserir no corpo desta mensagem de erro o parâmetro *RetryAfter*, que permite a definição de um intervalo mínimo, em milissegundos, após o qual a requisição poderá ser retransmitida.

A definição destas duas mensagens de erro relevantes à TF pelo WS-Addressing, juntamente com o que foi exposto na Seção 4.1, corroboram a escolha deste padrão como sendo o mais apropriado a ser estendido.

4.2.2 Tratamento de erros no WS-Addressing 1.0

O WS-Addressing define um elemento que pode ser utilizado para tratar falhas, o *FaultTo*. Este elemento permite que o emissor de uma mensagem especifique um terceiro *endpoint* ao qual o recebedor da mensagem deve enviar possíveis mensagens de erro. Entretanto, o uso deste elemento como base para a adição de técnicas de TF é limitado.

A Figura 4.1 ilustra uma interação de requisição e resposta típica em que o *endpoint* C acessa um serviço fornecido pelo *endpoint* S. Primeiramente, C obtém o documento WSDL que define a interface de S. Este pode ser obtido de um terceiro *endpoint* W (mensagens 1 e 2) ou diretamente de S. Em posse da interface de S, C gera um trecho de código chamado *stub*[2]. A principal funcionalidade dos stubs é facilitar a vida do programador, permitindo que estruturas da própria linguagem de programação sejam utilizadas para interagir com os serviços ao invés de manipular diretamente as mensagens em XML. A invocação de um método, por exemplo, torna-se uma invocação de serviço, feita pelo *framework* de serviços web.

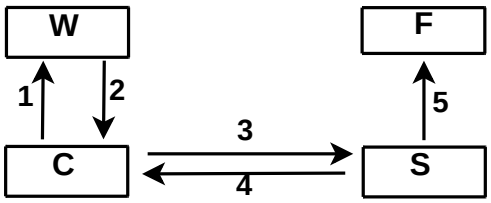


Figura 4.1: Possíveis interações entre nós utilizando serviços web.

A partir de então, C realiza requisições (mensagem 3) e S envia as respectivas respostas (mensagem 4) enquanto não há erros. Em caso de erro, S pode enviar uma mensagem de erro para C ou para um quarto *endpoint* F (mensagem 5) especificado pelo elemento *FaultTo* definido nos cabeçalhos da mensagem 3.

Ao se utilizar o elemento *FaultTo* como único meio de tratamento de falhas em uma solução baseada em serviços web, duas possibilidades são descartadas: a de aumentar a dependabilidade do sistema por meio da replicação de S utilizando apenas padrões estabelecidos e a de realizar a recuperação de maneira transparente ao cliente. Como apenas uma mensagem de erro é enviada a F, algum tipo de protocolo de recuperação de falhas *ad-hoc* deve ser estabelecido entre F, C, S e, possivelmente, as réplicas de S.

Outro problema presente no esquema de tratamento de falhas atual definido pelo WS-Addressing é o fato de a geração de mensagens do tipo *Destination Unreachable* ser opcional. Se deseja-se que o mecanismo de tolerância a falhas funcione de maneira independente do protocolo de transporte, a geração desta mensagem deve ser *obrigatória*.

[2]A maioria dos frameworks modernos para serviços web fornecem ferramentas para geração de *stubs* que recebem como entrada a interface WSDL de um serviço. Uma vez que o *stub* é gerado (ou implementado), todas as invocações realizadas por meio dele invocarão o *endpoint* definido na interface do serviço.

4.2.3 Novos elementos inseridos pela extensão

Para permitir a definição de réplicas de um serviço, complementando o elemento *FaultTo*, e permitindo a construção de soluções mais confiáveis, propomos a adição dos elementos *Replicas*, *Replica* e *Metadata* ao WS-Addressing. Estes elementos são definidos neste capítulo como se já fossem parte do XML Schema da WS-Addressing 1.0 [64], conforme definido na Figura 4.2.

Um elemento do tipo *Replica* é responsável pela definição do endereço de uma réplica. Este elemento é do tipo *EndpointReferenceType* já definido pelo WS-Addressing [64], utilizado para especificar o endereço de um *endpoint*.

```
<xs:complexType name="ReplicasReferenceType">
   <xs:sequence>
     <xs:element name="Replica" type="tns:EndpointReferenceType" minOccurs="1"
                 maxOccurs="unbounded"/>
     <xs:element ref="tns:Metadata" minOccurs="0"/>
   </xs:sequence>

   <xs:attribute name="style" default="passive">
     <xs:simpleType>
       <xs:restriction base="xs:string">
         <xs:enumeration value="passive"/>
         <xs:enumeration value="active"/>
       </xs:restriction>
     </xs:simpleType>
   </xs:attribute>
</xs:complexType>

<xs:element name="Replicas" type="tns:ReplicasReferenceType"/>
```

Figura 4.2: Definição do elemento *Replicas* baseando-se no XML Schema do WS-Addressing 1.0.

Todos os elementos do tipo *Replica* em um cabeçalho devem ter um, e apenas um, elemento pai do tipo *Replicas*, que tem como funções encapsular a lista de réplicas e definir o estilo de replicação (ver Seção 4.2.4). Opcionalmente, seguindo a premissa de extensibilidade de serviços web, o elemento *Replicas* pode também possuir um elemento filho *Metadata*, já definido pelo WS-Addresing 1.0. Este elemento pode conter qualquer tipo de metadados peculiares a uma aplicação.

Serviços replicados devem inserir um, e apenas um, elemento *Replicas* nos cabeçalhos de todas as mensagens enviadas. Essa convenção permite uma implementação mais direta e simples de se gerenciar. Outras opções seriam enviar a lista de réplicas apenas quando esta fosse atualizada ou fazer com que o cliente enviasse, a cada requisição, a sua lista local das réplicas do serviço. Neste último caso, o servidor enviaria uma nova lista de réplicas sempre que houvesse disparidade entre a versão recebida e a sua versão local. Testar possíveis ganhos de desempenho através destas outras abordagens é uma sugestão para trabalhos futuros.

4.2.4 Estilos de replicação

A extensão permite a utilização de dois estilos de replicação pelo fornecedor do serviço: passiva ou ativa. O estilo é definido através do atributo *style* do elemento *Replicas*.

Na replicação passiva (*style*="passive") apenas uma réplica (primária) está ativa em um determinado instante. Assim, requisições devem ser feitas ao *endpoint* definido pelo primeiro elemento *Replica* filho de *Replicas*. Em caso de falha, a requisição é feita à próxima réplica da lista de réplicas. Neste caso, a réplica contactada pode retornar a resposta da requisição, caso ela seja a nova primária, ou pode retornar uma mensagem de erro do tipo *Endpoint Unavailable* (ver Seção 4.2.1) com a lista de réplicas atualizada em seus cabeçalhos e com o parâmetro *RetryAfter* definido[3]. No segundo caso, a lista de réplicas no cliente é atualizada e garante-se que, após o tempo definido por *RetryAfter*, as interações poderão ser reestabelecidas.

Por outro lado, caso a replicação ativa (*style*="active") esteja sendo utilizada pelo fornecedor do serviço, todas as réplicas estão ativas e devem responder às requisições. Logo, estas devem ser enviadas para todas réplicas. O módulo do WS-Addressing no cliente fica responsável por definir o critério de entrega da resposta à aplicação. Um critério geralmente utilizado é o de esperar apenas pela primeira resposta recebida de qualquer uma das réplicas. Outras opções podem vir a ser definidas, por exemplo, através do elemento *Metadata*.

4.2.5 Um Exemplo

Considere que um serviço possua duas réplicas localizadas em *http://rep1.example.com* (*Rep1*) e *http://rep2.example.com* (*Rep2*). A Figura 4.3 ilustra um exemplo simples de utilização do elemento *Replicas*. Este deve ser inserido nos cabeçalhos das mensagens de resposta, possibilitando a identificação das réplicas do serviço pelo seu consumidor.

```
<S:Envelope xmlns:S="http://www.w3.org/2003/05/soap-envelope"
            xmlns:wsa="http://www.w3.org/2005/08/addressing">
 <S:Header>
  <wsa:MessageID>http://example.com/1</wsa:MessageID>
  <wsa:Replicas>
   <wsa:Replica>
     <wsa:Address>http://rep1.example.com</wsa:Address>
   </wsa:Replica>
   <wsa:Replica>
     <wsa:Address>http://rep2.example.com</wsa:Address>
   </wsa:Replica>
  </wsa:Replicas>
  <wsa:To>http://client.example.com</wsa:To>
 </S:Header>
 <S:Body> ... </S:Body>
</S:Envelope>
```

Figura 4.3: Utilizando o elemento *Replicas* em um cabeçalho SOAP.

[3]Esta proposta não define um valor específico para *RetryAfter*, delegando essa responsabilidade ao gerenciador de réplicas no lado do servidor.

Agora considere o cenário de replicação passiva da Figura 4.4, em que mensagens são identificadas por uma tupla (endereço, tipo de mensagem, identificador da mensagem) e em que há dois tipos de mensagem: requisição (Req) e resposta (Res). Os endereços nas mensagens correspondem aos elementos *To* e *From* do WS-Addressing nas requisições e nas respostas, respectivamente.

Na Figura 4.4 o cliente recebe a resposta de sua primeira requisição e então *Rep1* quebra. Contudo, o elemento *Replicas* da Figura 4.3 estava contido nos cabeçalhos da mensagem de resposta 1. Assim, quando o cliente faz a segunda invocação ao serviço, o *stub* invocará *Rep1*[4], mas o módulo do WS-Addressing (que fica após o *stub* na cadeia de processamento das mensagens) no lado do cliente detecta que *Rep1* quebrou e contacta *Rep2*. Como esta responde à requisição, as requisições subsequentes são redirecionadas automaticamente para *Rep2* pelo módulo do WS-Addressing, mas este altera as respostas para que a aplicação as enxergue como tendo sido geradas por *Rep1*.

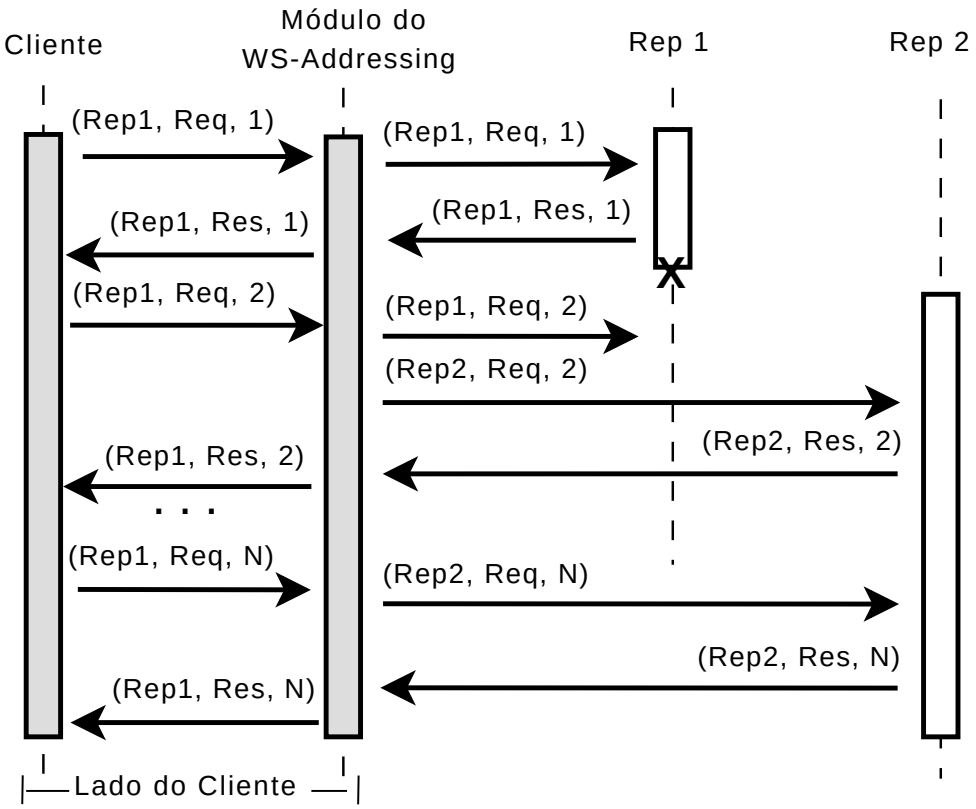


Figura 4.4: Comportamento da extensão em caso de falha da réplica primária.

4.3 O Protótipo

Para implementar o protótipo da extensão, escolhemos alterar o módulo responsável pelo processamento do WS-Addressing no *framework* Apache Axis2 1.5 em Java [56]. O Axis2

[4]O *stub* foi gerado com *Rep1* como *endpoint* de destino.

é um *framework* para desenvolvimento de serviços web maduro que oferece suporte às últimas versões de alguns padrões WS-*, dentre eles o WS-Addressing 1.0. O protótipo atual suporta apenas a replicação passiva. Esta Seção descreve seus componentes e seu funcionamento. Ao final alguns resultados preliminares de desempenho são fornecidos.

4.3.1 Estrutura do Axis2

A unidade básica para a manipulação de mensagens no Axis2 é chamada *handler*. Cada *handler* implementa um método que recebe como parâmetro, entre outros dados, o envelope SOAP, podendo manipulá-lo como desejar. Os padrões WS-* são implementados no Axis2 por meio de um conjunto de *handlers*, responsáveis pela interpretação ou inserção dos cabeçalhos específicos de cada padrão. Os *handlers* podem ser agrupados em fases que, por sua vez, constituem os fluxos do Axis2, conforme ilustrado pela Figura 4.5.

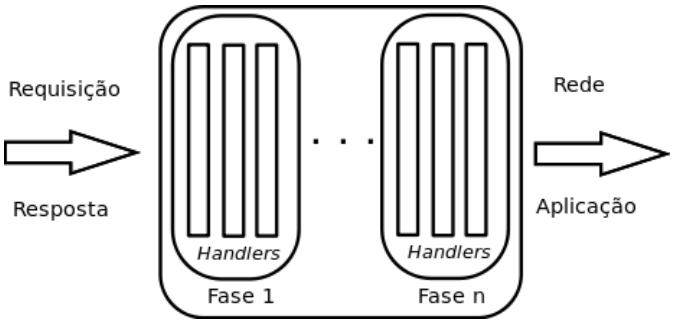


Figura 4.5: Um fluxo do Axis2.

Um fluxo nada mais é que um conjunto de fases, logo de *handlers*, responsáveis pelo processamento das mensagens antes que elas sejam entregues à aplicação, no caso dos fluxos de entrada, ou antes que elas sejam enviadas pela rede, no caso dos fluxos de saída. O Axis2 fornece quatro abstrações de fluxo de mensagens: o *InFaultFlow*, fluxo de entrada para mensagens de erro; o *InFlow*, fluxo de entrada para mensagens que não sejam de erro; o *OutFaultFlow*, fluxo de saída para mensagens de erro; e o *OutFlow*, fluxo de saída para mensagens que não sejam de erro.

Além dos fluxos, o Axis2 também possui componentes localizados após os fluxos de saída, chamados *Transport Senders*. Um *Transport Sender* é invocado após o processamento da mensagem em um fluxo de saída e é responsável pelo envio da mensagem na rede, de acordo com o protocolo de camada de transporte que implementa[5].

4.3.2 Organização do protótipo

A implementação atual do protótipo é composta por cinco componentes (ver Figura 4.6). São quatro *handlers* (ver Seção 4.3.3) e um *Transport Sender* (ver Seção 4.3.4). Além

[5]Vários protocolos de camada de transporte são implementados no Axis2. Para cada um deles, há um *Transport Sender* específico.

disso, utilizamos também o banco de dados Apache Derby [57] para armazenar as listas de réplicas de maneira persistente no cliente. Ele é acessado através da classe intermediária *AddressingStorageHelper*, que permite a troca de meio de armazenamento sem a necessidade de modificar o restante do protótipo.

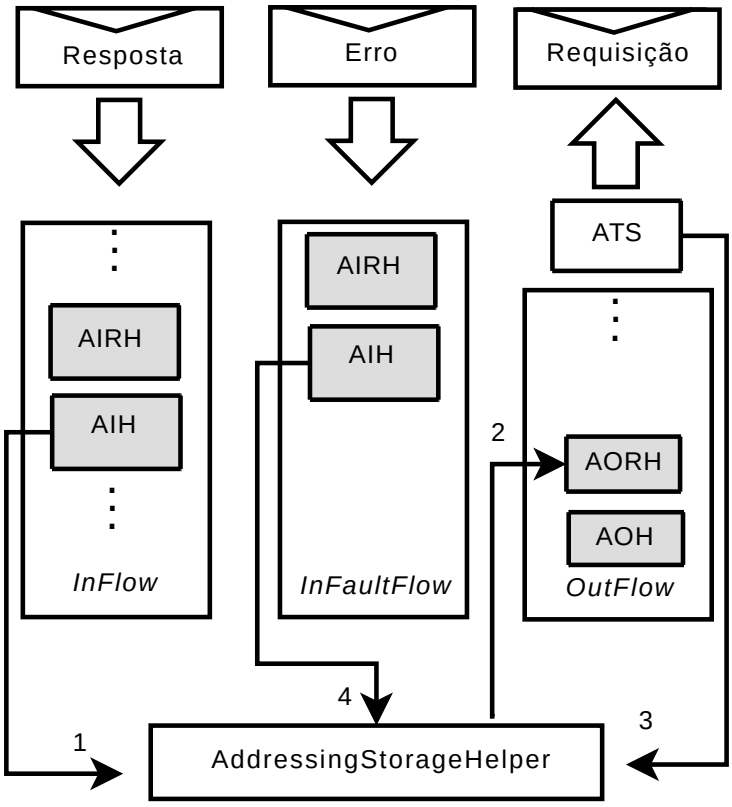


Figura 4.6: Estrutura do protótipo implementado por meio de modificações ao módulo WS-Addressing do *framework* Axis2.

4.3.3 Os *Handlers*

Os quatro *handlers* são ilustrados em caixas cinzas na Figura 4.6. O comportamento de cada um deles é descrito a seguir:

- *AddressingInHandler* (AIH): trata-se do *AddressingInHandler* distribuído juntamente com o módulo WS-Addressing do Axis2, responsável por interpretar os cabeçalhos do WS-Addressing das mensagens recebidas. Ele foi modificado para também ser capaz de interpretar o elemento *Replicas* e armazenar a lista de réplicas (mensagens 1 e 4 na Figura 4.6). Sempre que uma nova lista de réplicas é recebida, a lista antiga é descartada. O AIH, portanto, constitui a primeira porta de entrada das listas de réplicas no sistema, baseando-se apenas nos cabeçalhos do WS-Addressing. Contudo, futuramente, a lista pode vir a ser definida em outros locais (ver Trabalhos Futuros na Seção 4.5).

- *AddressingInReplicationHandler* (AIRH): localizado no *InFlow* e no *InFaultFlow*, modifica os cabeçalhos das mensagens recebidas para que estes correspondam ao *endpoint* contactado originalmente. Ele é necessário pois as respostas recebidas, advindas de uma recuperação anterior, possuirão o endereço da nova réplica primária como origem.

- *AddressingOutHandler* (AOH): trata-se do *AddressingOutHandler* distribuído juntamente com o Axis2, responsável pela inserção dos cabeçalhos do WS-Addressing nas mensagens enviadas. Este handler foi modificado para inserir também o elemento *Replicas*, caso uma lista de réplicas tenha sido especificada no arquivo de configuração do módulo do WS-Addressing, e o elemento *SequenceID* (ver Seção 5.4).

- *AddressingOutReplicationHandler* (AORH): responsável por verificar (mensagem 2 na Figura 4.6) se o serviço contactado possui réplicas, ou seja, o elemento *Replicas* foi recebido nos cabeçalhos de uma mensagem anterior. Caso o serviço possua réplicas, o *Transport Sender* atual é substituído pelo *AddressingTransportSender* (ver Seção 4.3.4).

4.3.4 O AddressingTransportSender

O *AddressingTransportSender* (ATS), assim como qualquer outro *Transport Sender*, é responsável pelo envio das mensagens na rede. Entretanto, além de enviar a mensagem, ele é também responsável por monitorar a sua chegada no *endpoint* de destino e, em caso de falha, invocar as outras réplicas do serviço. Isto é feito por meio dos seguintes passos:

- Passo 1: Modificar os cabeçalhos do WS-Addressing da mensagem sendo enviada de maneira a refletir uma possível mudança de réplica primária (ver Passo 5).

- Passo 2: Invocar o serviço.

- Passo 3: Se nenhuma falha for detectada, o cliente recebe a resposta normalmente.

- Passo 4: Se a réplica primária estiver inacessível, a próxima primária é extraída da lista de réplicas. Se não houver mais réplicas disponíveis, uma mensagem *Destination Unreachable* é enviada para a aplicação e o processo termina.

- Passo 5: O endereço da nova réplica primária é atualizado no meio de armazenamento (mensagem 3 na Figura 4.6) e o processo de recuperação volta para o Passo 1.

4.3.5 Considerações em relação à implementação

Uma decisão de implementação importante que deve ser feita se refere ao tempo de vida da lista de réplicas no cliente. Se a lista é armazenada em memória, esta será perdida

quando a sessão[6] expirar. Caso isto ocorra, se a réplica primária original quebrar e não for substituída ou não se recuperar, interações futuras não serão possíveis, a não ser que um novo *stub* do serviço seja gerado para corresponder ao endereço de uma nova réplica primária.

Uma solução para o problema de perda da lista de réplicas é utilizar um meio de armazenamento permanente, assim como é feito pelo protótipo. Neste caso, o nível de TF é estendido: se a réplica primária não se recuperar, enquanto as outras réplicas continuarem interagindo com o cliente e, por consequência, continuarem atualizando a lista de réplicas, a obtenção do serviço será possível. O custo desta solução é baixo: basta armazenar, juntamente com a lista de réplicas, o endereço da réplica primária original, que é contactada pelo *stub*.

Por fim, é preciso ressaltar que algumas medidas de segurança também precisam ser implementadas. Em relação ao elemento *Replicas*, as medidas devem ser as mesmas adotadas pelo WS-Addressing 1.0 em relação aos elementos *FaultTo* e *ReplyTo* [62]. Isto significa estabelecer um mecanismo que garanta que o emissor da lista de réplicas (fornecedor do serviço) está habilitado a redirecionar fluxos de mensagens para os *endpoints* especificados pela mesma.

4.3.6 Considerações em relação ao desempenho

Para os testes do protótipo foram usadas três réplicas em rede local de 100 Mbps, rodando a distribuição de linux Ubuntu 9.04, a JVM 1.6.0_0 e o servidor de aplicações Apache Tomcat 6.0.18. Duas réplicas possuíam processadores Intel Core 2 Duo E8300 com 4 GB de RAM e a terceira possuía dois processadores Xeon Quad Core E5504 e 16 GB de RAM. Inicialmente o banco de dados do cliente foi populado com cem *listas* de réplicas.

Primeiramente, é importante notar que apenas a inicialização do banco de dados demora 0,5 segundo, em média. Não é possível evitar essa sobrecarga, mesmo quando apenas serviços não replicados são acessados, pois o banco de dados é consultado ao menos uma vez para verificar se alguma lista relativa ao serviço já foi armazenada no cliente anteriormente.

Durante os testes foi invocado um serviço de *echo* (que apenas retorna a string fornecida como parâmetro) e cada invocação possuía um parâmetro de 1,5 KB. Testes com outros tamanhos de parâmetro foram realizados, mas não foi percebida uma grande influência nos resultados finais. Nos experimentos o SimpleRep (ver Capítulo 5) foi utilizado em todos os cenários, pois para que a extensão funcione é necessário que o servidor envie a lista de réplicas. Os experimentos avaliaram a sobrecarga imposta pela extensão em quatro cenários:

1. **Cenário 1**: O serviço foi invocado utilizando o Axis2 não modificado no cliente. Este cenário serve de base para as comparações com a extensão.

[6]Intervalo de tempo em que o processo cliente estiver ativo.

2. **Cenário 2**: O serviço foi invocado utilizando a extensão, mas o elemento *Replicas* não foi enviado nos cabeçalhos, simulando a utilização de um serviço não replicado.

3. **Cenário 3**: O serviço foi invocado utilizando a extensão, o elemento *Replicas* foi enviado, mas não há quebra da réplica primária.

4. **Cenário 4**: O serviço foi invocado utilizando a extensão e o elemento *Replicas* foi enviado, porém, com quebra da réplica primária. A quebra da réplica primária foi simulada da seguinte maneira: após trinta invocações a *thread* da aplicação cliente foi suspendida por cinco segundos e neste intervalo de tempo o servidor de aplicações (Tomcat) foi desligado. Ao retornar à execução, o cliente identificou a quebra e realizou a recuperação.

Os tempos de resposta obtidos por meio de requisições consecutivas são ilustrados pela Figura 4.7.

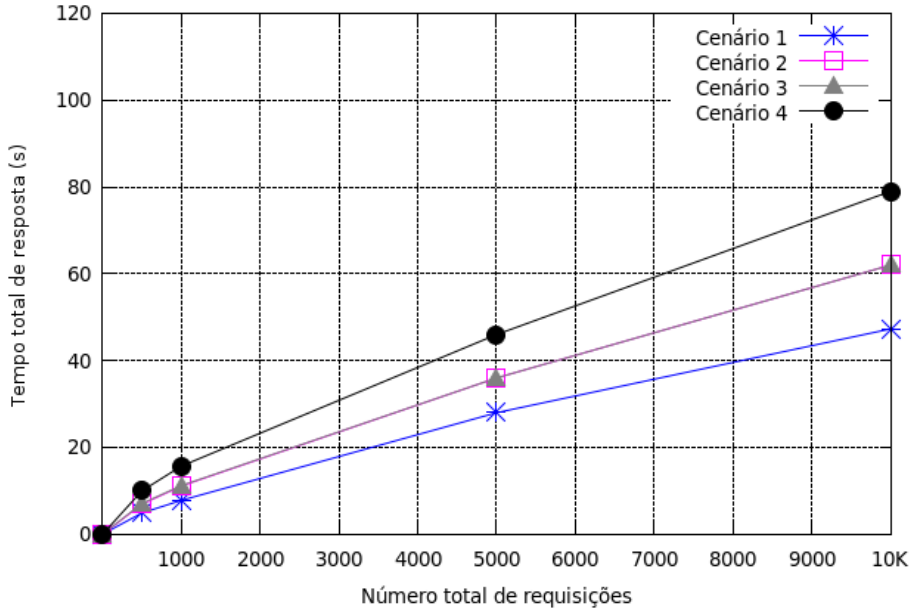


Figura 4.7: Tempos de resposta obtidos nos quatro cenários de testes com a extensão ao WS-Addressing.

Os altos tempos de execução quando a extensão é utilizada refletem o fato de que o elemento *Replicas* é armazenado como texto puro no banco de dados. Assim, sempre que o *handler AORH* consulta o banco de dados para verificar se o serviço possui réplicas, todas as linhas da tabela de réplicas são visitadas. É interessante notar que a simulação assume que cem serviços replicados estão sendo acessados simultaneamente. O teste de 10000 invocações no Tomcat (sem o SimpleRep) com 5 listas de réplicas forneceu, em média, tempos de execução de 43s, enquanto utilizando a distribuição padrão do Axis2 os tempos foram de 38s, uma sobrecarga de 13%.

O cenário 2 fornece uma base de comparação para *frameworks* que não possuem funcionalidades que permitam evitar esta verificação a cada interação. Contudo, o Axis2

fornece objetos do tipo *ServiceContext* que permitem que dados relativos a um serviço sejam armazenados em memória durante as interações. Desta forma, apenas uma variável indicando o fato de o serviço ser replicado já basta para evitar essa consulta ao banco.

O cenário 3 mensura a sobrecarga gerada pelo acesso a um serviço replicado, mesmo que não ocorram falhas. Como é possível notar, a utilização do ATS tem um custo muito baixo, pois os tempos são praticamente idênticos aos obtidos utilizando o *Transport Sender* HTTP padrão do Axis2.

O quarto cenário ilustra a quebra da réplica primária e os subsequentes redirecionamentos das requisições para a nova réplica primária. Os tempos de resposta dependem do momento em que a quebra ocorre, quanto mais cedo a quebra ocorrer, maior será a sobrecarga total, pois maior será o período em que a extensão fará redirecionamentos para a nova réplica primária (ver Figura 4.4). Por este motivo, nos testes a réplica primária quebra após atender trinta requisições apenas, evidenciando a sobrecarga da transparência da recuperação ao cliente.

4.4 Trabalhos relacionados

A extensão proposta foi inspirada pelo elemento <WSG> do projeto FT-SOAP [18]. Entretanto, a utilização desse elemento, conforme proposto em [18] apresenta dois pontos negativos:

- Estende apenas a WSDL, logo, a lista de réplicas só pode ser definida na interface do serviço. Desta forma, mudanças futuras no grupo de réplicas não serão visíveis aos clientes que já tenham gerado o *stub* do serviço. Ao estender o WS-Addressing, é possível enviar novas versões do grupo de réplicas conforme mensagens são trocadas entre o fornecedor e o consumidor do serviço.
- Baseia-se em registros que seguem o padrão *Universal Description Discovery and Integration* (UDDI) como mecanismo de propagação de atualizações do grupo de réplicas. Acreditamos que a utilização de um registro UDDI como parte da infraestrutura de TF apresenta alguns problemas:
  - O padrão, por si só, até o presente momento, não é amplamente adotado. Outros padrões com os mesmos propósitos foram criados, como o ebXML, mas ainda sem grande representatividade [36].
  - Em várias soluções a utilização de um registro UDDI não é necessária.
  - O registro se torna mais um ponto único de falha na arquitetura.

  Entretanto, ao se estender o WS-Addresing, remove-se a necessidade de utilização de UDDI, sem impor qualquer tipo de restrição a sua utilização.

Além do WS-Addressing, a especificação WS-ReliableMessaging [39] também engloba algumas funcionalidades interessantes que poderiam ser reutilizadas ou estendidas para

alcançar os objetivos da extensão. Contudo, essa especificação não sofreu a mesma adoção que o WS-Addressing e tem o seu o escopo restrito a comunicações ponto a ponto. Um estudo sobre a combinação da WS-ReliableMessaging com a extensão proposta neste capítulo pode ser um bom ponto inicial para trabalhos futuros.

4.4.1 Os efeitos da extensão em outros trabalhos

Há muitas propostas de TF em serviços web e grande parte delas pode se beneficiar deste trabalho. O FT-SOAP [18] e o trabalho de Wah [67] utilizam um registro UDDI para atualizar a lista de réplicas. Caso esses trabalhos passem a utilizar o WS-Addressing estendido, seria possível remover o registro UDDI da solução.

O SmartWS [28], o WS-Replication [49] e o trabalho de Osrael et al. [42] interagem com o cliente através de *proxies*[7], que são responsáveis pelas interações com as réplicas. Essa abordagem é transparente ao cliente, mas os *proxies* constituem pontos únicos de falha. Esta deficiência, todavia, pode ser superada por meio da replicação dos *proxies* ou do endereçamento das réplicas diretamente, utilizando a extensão.

O FTWeb [51] e o trabalho de Ye e Shen's [69] utilizam replicação ativa. Assim, é necessário possuir uma lista atualizada de réplicas a cada interação com o cliente. Porém, nenhum desses trabalhos detalha o método utilizado para especificar a lista de réplicas para o cliente. Ambos podem se beneficiar da extensão proposta por este trabalho e o mesmo pode ser dito a respeito de propostas baseadas no modelo de *n-Versões* [3], como o FT-GRID [60], que possuem a mesma necessidade.

O Thema [35] tolera falhas bizantinas [30] em serviços web. Entretanto, ele não pode ser considerado transparente ao cliente, pois este precisa utilizar a biblioteca do *framework* tanto no lado do cliente quanto do servidor.

Alwagait e Ghandeharizadeh [1] propõem uma arquitetura baseada em um servidor tratador de exceções, ao qual eles denominam *registro DeW*. Essa solução não é transparente ao cliente, pois o cliente precisa implementar a lógica responsável pelas interações com o registro DeW.

Por fim, o mais importante a ser notado é o fato de que a extensão pode ser utilizada por quase todos os trabalhos apresentados nesta Seção, conferindo, em alguns casos, até mesmo aumentos no nível de TF do *framework*, caso a lista de réplicas seja armazenada em meio permanente.

4.5 Conclusões e trabalhos futuros

A arquitetura de serviços web tem sofrido expansão significativa nos últimos anos e considerações em relação a sua dependabilidade são de importância fundamental. A proposta deste capítulo fornece uma maneira simples e genérica de criar uma ponte entre o cliente

[7] O trabalho de Osrael et. al utiliza a réplica primária como *proxy*.

e a arquitetura de tolerância a falhas no lado do servidor de maneira transparente. É interessante notar que esta proposta também livra os desenvolvedores de técnicas de replicação e balanceamento de carga da tarefa de definir a maneira de referenciar as réplicas e de implementar o código correspondente.

Com os testes (Seção 4.3.6), o que foi apresentado de fato é a sobrecarga pura gerada pelo uso da extensão. Em um cenário real, com atrasos na rede e tempos de processamento mais longos, tanto no lado do cliente quanto do servidor, a porcentagem de tempo relativa ao processamento da extensão será reduzida.

Considerando as pequenas modificações feitas pela extensão no WS-Addressing, em várias situações os benefícios à dependabilidade dos serviços valem o custo. A possibilidade de enviar novas versões do grupo de réplicas juntamente com os cabeçalhos SOAP permite aumentar o nível de tolerância a falhas nas interações entre os serviços web. Além disso, uma grande qualidade da extensão é o fato de ser independente de framework de TF podendo, inclusive, trabalhar em conjunto com vários dos frameworks já propostos (ver Seção 4.4.1). Caso a extensão seja incorporada ao WS-Addressing ou se uma nova especificação de TF venha a incorporá-la, todas as soluções que estejam em conformidade com o padrão resultante serão capazes de se beneficiar da recuperação transparente ao cliente, incluindo-se neste conjunto os sistemas legados.

Por fim, algumas possibilidades de trabalhos futuros são: permitir a especificação de réplicas também nas interfaces WSDL, para que o cliente não precise realizar ao menos uma interação com o fornecedor do serviço para receber a lista de réplicas inicial; mensurar os impactos da utilização de outros meios de armazenamento no desempenho do protótipo; mensurar os efeitos colaterais que eventualmente possam ser inseridos pela extensão em outros padrões e vice-versa e implementar a replicação ativa.

Capítulo 5

SimpleRep - Incorporação de Tolerância a Falhas em Serviços Web

Conforme apresentado no Capítulo **??**, já existem vários trabalhos que permitem agregar algum tipo de tolerância a falhas (TF) em serviços web. Contudo, a maioria destes trabalhos apresenta algum tipo de acoplamento à solução, sendo os mais comuns:

- Acoplamento ao cliente: o cliente precisa utilizar uma biblioteca do *middleware* de TF ou até mesmo modificar a lógica de negócios.
- Acoplamento ao ambiente de desenvolvimento: o *middleware* só funciona em serviços implementados para uma determinada plataforma, restringindo a liberdade do desenvolvedor de escolher a opção mais adequada para a sua solução.

Este capítulo apresenta o SimpleRep, um *middleware* de tolerância a falhas (TF) em serviços web. Em seu estágio atual, o SimpleRep implementa apenas replicação ativa dos serviços. Entretanto, pautando-se pelas premissas de interoperabilidade entre os serviços web e inspirando-se em trabalhos anteriores, a arquitetura realiza a replicação independentemente da plataforma ou da linguagem de programação utilizadas na implementação dos serviços.

O restante deste capítulo é organizado da seguinte maneira: a Seção 5.1 apresenta uma visão geral da arquitetura. Na Seção 5.2 são apresentados os detalhes dos componentes de um Agente de Replicação (AR), componente chave na arquitetura do SimpleRep. Em seguida, a Seção 5.3 fornece detalhes sobre a implementação da replicação ativa e na Seção 5.4 o problema das requisições duplicadas é tratado. Algumas considerações em relação ao desempenho da solução são feitas na Seção 5.5 e, por fim, as conclusões são apresentadas na Seção 5.6.

5.1 Uma visão geral do SimpleRep

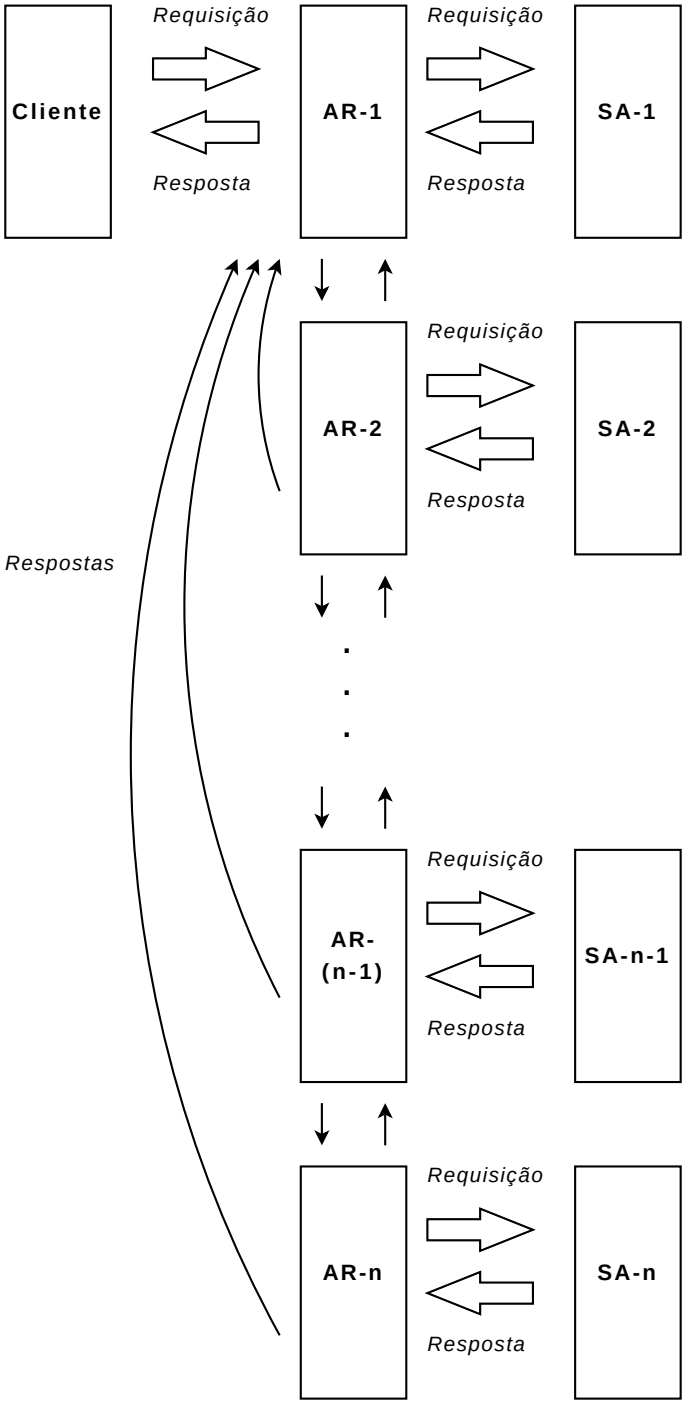


Figura 5.1: Visão de alto nível da arquitetura na Replicação Ativa.

A arquitetura do SimpleRep foi inspirada, basicamente, pela arquitetura do *framework* Apache Axis2 [56]. Conforme exposto na Seção 4.3.1, o Axis2 utiliza abstrações de fluxos

de entrada e saída em que *handlers* podem ser inseridos para manipular os envelopes[1] de requisições e respostas, respectivamente.

Cada réplica do SimpleRep é constituída por dois componentes: um **Servidor de Aplicações (SA)** e um **Agente de Replicação (AR)**. O SA pode ser qualquer componente capaz de fornecer serviços web (Tomcat, Glassfish, JBoss, etc) e é o responsável pelo fornecimento dos serviços. Para cada SA, há um AR correspondente. As invocações são recebidas pelos ARs que, então, comunicam-se entre si e manipulam as mensagens para que estas sejam processadas pelos SAs, conforme ilustrado pela Figura 5.1.

Algo importante que pode ser notado através da Figura 5.1 é que o SimpleRep é completamente independente do SA utilizado, ficando a lógica da replicação encapsulada nos ARs. O SA pode, inclusive, ser executado em uma máquina diferente da máquina em que o AR esteja sendo executado. Desta forma, para a replicação ativa, por exemplo, até mesmo serviços que já estejam em produção podem ser replicados sem modificações (transparência ao servidor). Em relação a esta característica, o SimpleRep segue a proposta de *conectores tolerantes a falhas* de Salatge e Fabre [50] e se junta às soluções que constróem "SOA confiável a partir de serviços não confiáveis".

Conforme já exposto, os ARs são os responsáveis por intermediar a comunicação entre os clientes e os SAs. O fluxo de uma mensagem em um AR, conforme ilustrado pela Figura 5.2, começa no componente *HTTP Proxy*. Logo em seguida, esta mensagem pode ser manipulada através de *handlers* antes de ser enviada ao SA correspondente ao AR. A comunicação entre os ARs é feita utilizando o *framework Samoa* de comunicação distribuída. Detalhes sobre cada um dos componentes de um AR são fornecidos na Seção 5.2. Além disso, uma visão de mais baixo nível sobre o funcionamento do SimpleRep, detalhando as interações entre as *threads* do mesmo, é fornecida no Apêndice A.3.

5.2 Os componentes de um Agente de Replicação

Conforme ilustrado pela Figura 5.2, o SimpleRep é constituído por três grandes componentes: um HTTP Proxy, os *Handlers* e o Samoa. Esses três componentes são detalhados nas seções a seguir.

5.2.1 O HTTP Proxy

O ***HTTP Proxy*** é um *proxy* reverso que interpreta o protocolo HTTP, modificado para interagir com os componentes do SimpleRep, principalmente com o Samoa. Foi utilizado como base a classe *ElementalReverseProxy*, distribuída em conjunto à biblioteca *Apache HttpComponents* [21].

Idealmente, este *proxy* reverso deveria ser genérico e o *framework* deveria fornecer a implementação de vários protocolos de transporte, assim como ocorre no Apache Axis2.

[1] Mensagens trocadas utilizando o padrão SOAP são chamadas "envelopes".

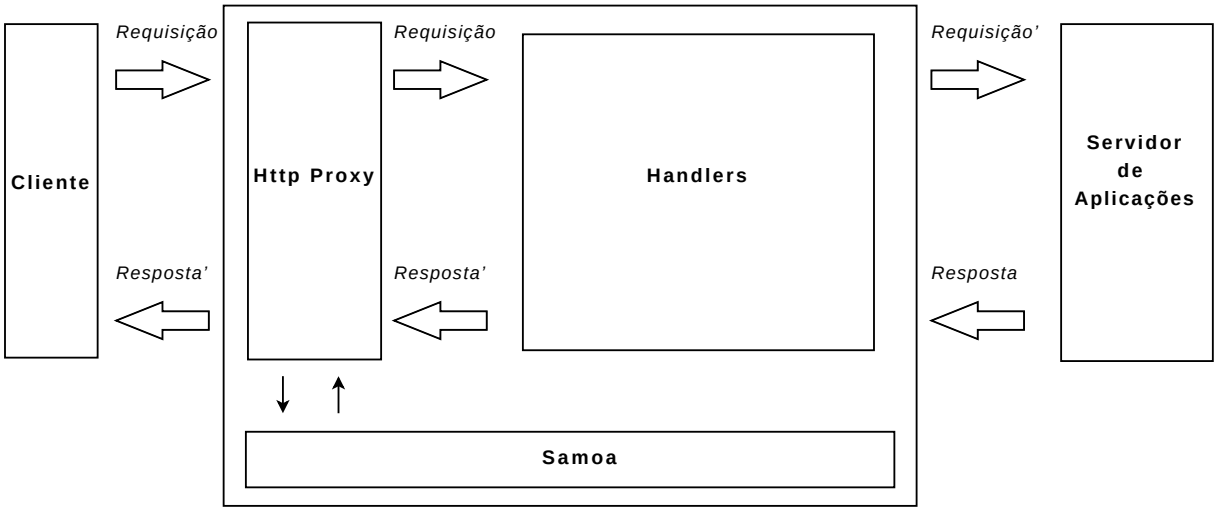


Figura 5.2: Os componentes de um Agente de Replicação em uma interação com um cliente.

Entretanto, a implementação deste componente da arquitetura, para outros protocolos diferentes do HTTP, foi deixada como sugestão para trabalhos futuros.

5.2.2 Os *Handlers*

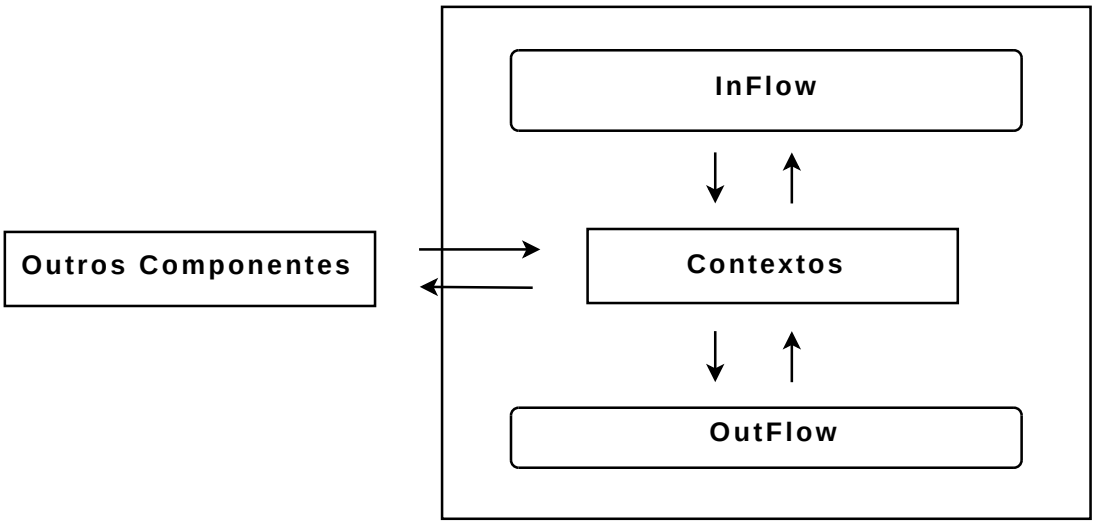


Figura 5.3: Os fluxos do SimpleRep.

Os ***handlers***, assim como no Axis2, permitem que tanto os envelopes de requisições quanto de respostas sejam manipulados. Uma vez implementado, um *handler* precisa ser inserido no fluxo de requisições (*InFlow*), no fluxo de respostas (*OutFlow*) ou até mesmo em ambos para que passe a realizar o processamento das mensagens. Um exemplo de *handler* utilizado na implementação do SimpleRep é o *AddressingReplicasOutHandler*. O mesmo foi inserido no *OutFlow* e é responsável pela inserção do elemento *Replicas* nas

mensagens de resposta. Assim, serviços replicados através do SimpleRep passam a gerar o elemento *Replicas* independentemente de o SA suportar ou não a extensão ao WS-Addressing proposta neste trabalho. Um exemplo de implementação de um *handler* é apresentado no Apêndice A.2.

Os ***Contextos*** são áreas de armazenamento de dados que podem ser acessadas tanto por *handlers* quanto pelo outros componentes de um AR, conforme ilustrado pela Figura 5.3. Para o SimpleRep, apenas um subconjunto das abstrações de contexto criadas pelo Axis2 foi implementado. São os seguintes:

- Contexto de Mensagem (*MessageContext*): cada envelope recebido é encapsulado em um Contexto de Mensagem. Dados pertinentes apenas ao processamento da requisição ou da resposta devem ser armazenados no Contexto de Mensagem, pois eles serão visíveis apenas no *InFlow* (para requisições) ou no *OutFlow* (para respostas).
- Contexto de Operação (*OperationContext*): permite o armazenamento de dados pertinentes ao processamento de toda a operação (requisição e resposta). Assim sendo, tanto os *handlers* no *InFlow* quanto no *OutFlow* têm acesso a esses dados.
- Contexto de Serviço (*ServiceContext*): permite o armazenamento de dados pertinentes a todas as interações com um determinado serviço. Os dados inseridos neste contexto são armazendos enquanto um cliente estiver interagindo com o serviço, sendo útil para um processamento que necessite de dados obtidos em outras operações.

5.2.3 O Samoa

Conforme visto na Seção 5.2, cada AR possui também uma instância do Samoa [70]. O Samoa é um *framework* que fornece várias abstrações para comunicação entre aplicações, sendo a de maior interesse para o SimpleRep, na replicação ativa, o Atomic Broadcast (ver Seção 5.3).

O Samoa, ao contrário da maior parte dos *frameworks* de comunicação distribuída, não utiliza eventos para interações locais entre os módulos de protocolos, mas sim uma abstração de serviços. Ele também implementa funcionalidades avançadas, como a atualização e troca dinâmica de protocolos. Apesar de o Samoa ter sido utilizado em sua versão mais simples no SimpleRep, nada impede que trabalhos futuros sejam baseados apenas na customização de parâmetros do mesmo e na implementação de protocolos de replicação mais avançados, como a replicação semi-passiva [15], em busca de soluções mais eficientes.

5.3 A Replicação Ativa

Ao empregar o estilo de replicação ativa, tem-se como pressuposto a existência de um grande número de serviços que possuam comportamento determinístico, nos quais as mudanças de estado e a as respostas geradas dependam apenas do estado atual das réplicas,

das requisições e da ordem em que as requisições chegam, independentemente de outras atividades no sistema.

Para o caso de replicação de serviços, o evento não-determinístico a ser sincronizado é a entrega (*delivery*) das requisições à aplicação. No modelo *crash-stop*, para preservar a **consistência do estado interno do serviço**, as entregas das requisições aos SAs devem possuir as seguintes propriedades [29]:

- **Acordo na Entrega (*Delivery Agreement*):** Se uma réplica correta p entrega a requisição r, então a réplica q, em algum momento futuro, entrega r ou q quebrou.
- **Ordenação Uniforme na Entrega (*Uniform Delivery Order*)**: Se uma réplica correta p entrega as requisições r e r' e entrega primeiro r, então a réplica q (correta ou não) não entrega a requisição r' a não ser que r já tenha sido entregue.

No SimpleRep, ambas propriedades acima são garantidas pela utilização do *Atomic Broadcast* do Samoa.

Além das propriedades relativas à entrega das mensagens nas réplicas, para se manter a **consistência do estado do sistema** é necessário também que a propriedade de **causalidade** seja satisfeita [29, 52]:

- **Causalidade**: a ordem de entrega das requisições dos clientes devem respeitar suas potenciais relações causais:
  - Se um cliente c realiza uma requisição r e após isto ele realiza uma requisição r', então a réplica p não entrega r' a não ser que já tenha entregue r, ou p não é correta.
  - Se o fato de uma requisição r realizada por um cliente c a uma réplica p puder ser a causa da realização de uma requisição r' por um cliente c', então p processa r antes de r'

Ao assumir que as interações com os serviços serão realizadas apenas de maneira bloqueante, garante-se, trivialmente, que a causalidade será satisfeita.

É importante notar que o SimpleRep implementa uma variação da replicação ativa em que o cliente interage com apenas uma das réplicas (AR). Na abordagem clássica [52], uma requisição é feita pelo cliente a todas as réplicas e o mesmo deve tratar todas as respostas geradas. A abordagem adotada pelo SimpleRep facilita a construção de soluções transparentes ao cliente e mostra-se mais eficaz em ambientes em que o cliente não faz parte do grupo de réplicas (*open group model*), como a Internet [52].

Além de satisfazer as propriedades básicas da replicação ativa, o SimpleRep também implementa um mecanismo simples de melhoria de desempenho. Sempre que uma mensagem é processada por uma réplica, a mesma envia o resultado do processamento para a réplica que recebeu a requisição do cliente, conforme ilustrado pela Figura 5.4. Assim, se por algum motivo um determinado SA se tornar mais lento que os demais, os clientes

não perceberão a degradação no serviço. Na Figura 5.4, por exemplo, o SA-1 está mais lento que o SA-2 e, por este motivo, a resposta (*Resp1*) gerada pelo SA-2 é enviada ao cliente, enquanto a resposta gerada pelo SA-1 é descartada. Esta otimização foi implementada através de modificações no *HTTP Proxy* e com a utilização do serviço de troca de mensagens UDP fornecido pelo Samoa.

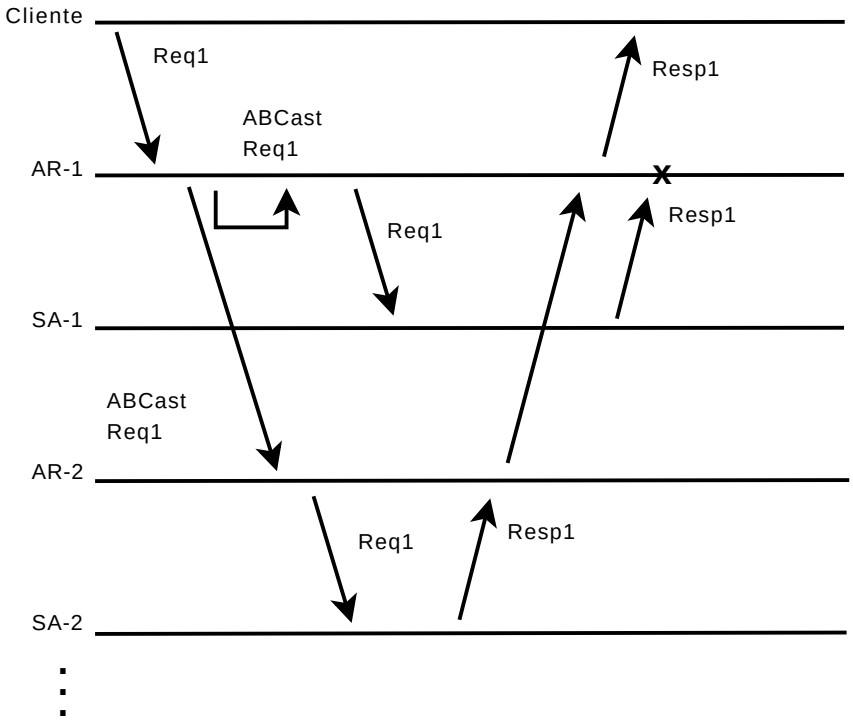


Figura 5.4: Otimização no processamento de mensagens na replicação ativa.

Exemplos de interações da replicação ativa no SimpleRep são ilustrados na Figura 5.5. Na Figura 5.5-a, o SA-1 gerou a resposta (*Resp1*) mais rapidamente e, por consequência, o cliente teve a sua requisição (*Req1*) atendida por SA-1. Já na Figura 5.5-b, o SA-2 gerou a resposta em um menor tempo e teve sua resposta enviada ao cliente.

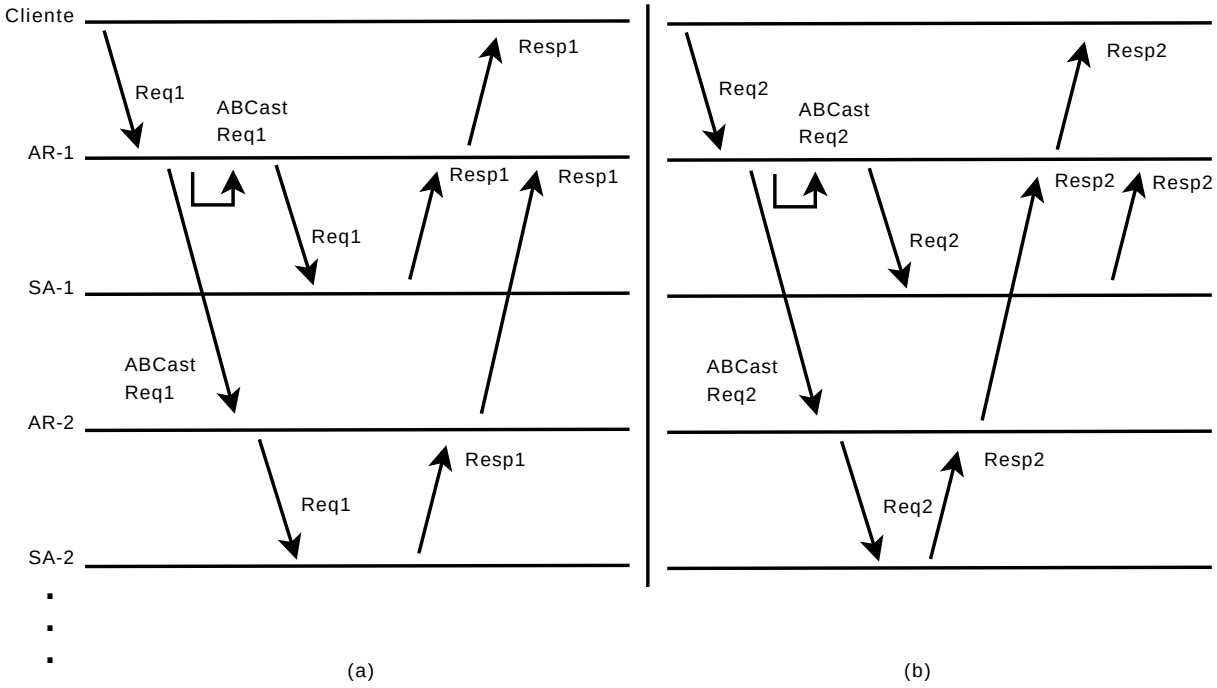


Figura 5.5: Processamento de uma requisição na replicação ativa. (a) Após o *Atomic Broadcast* da requisição (*Req1*), o AR-1 gerou a resposta (*Resp1*) mais rapidamente. (b) Após o *Atomic Broadcast* da requisição (*Req2*), o AR-2 gerou a resposta (*Resp2*) mais rapidamente.

Por fim, os estados assumidos por um AR durante o seu funcionamento na replicação ativa são ilustrados pelo diagrama da Figura 5.6.

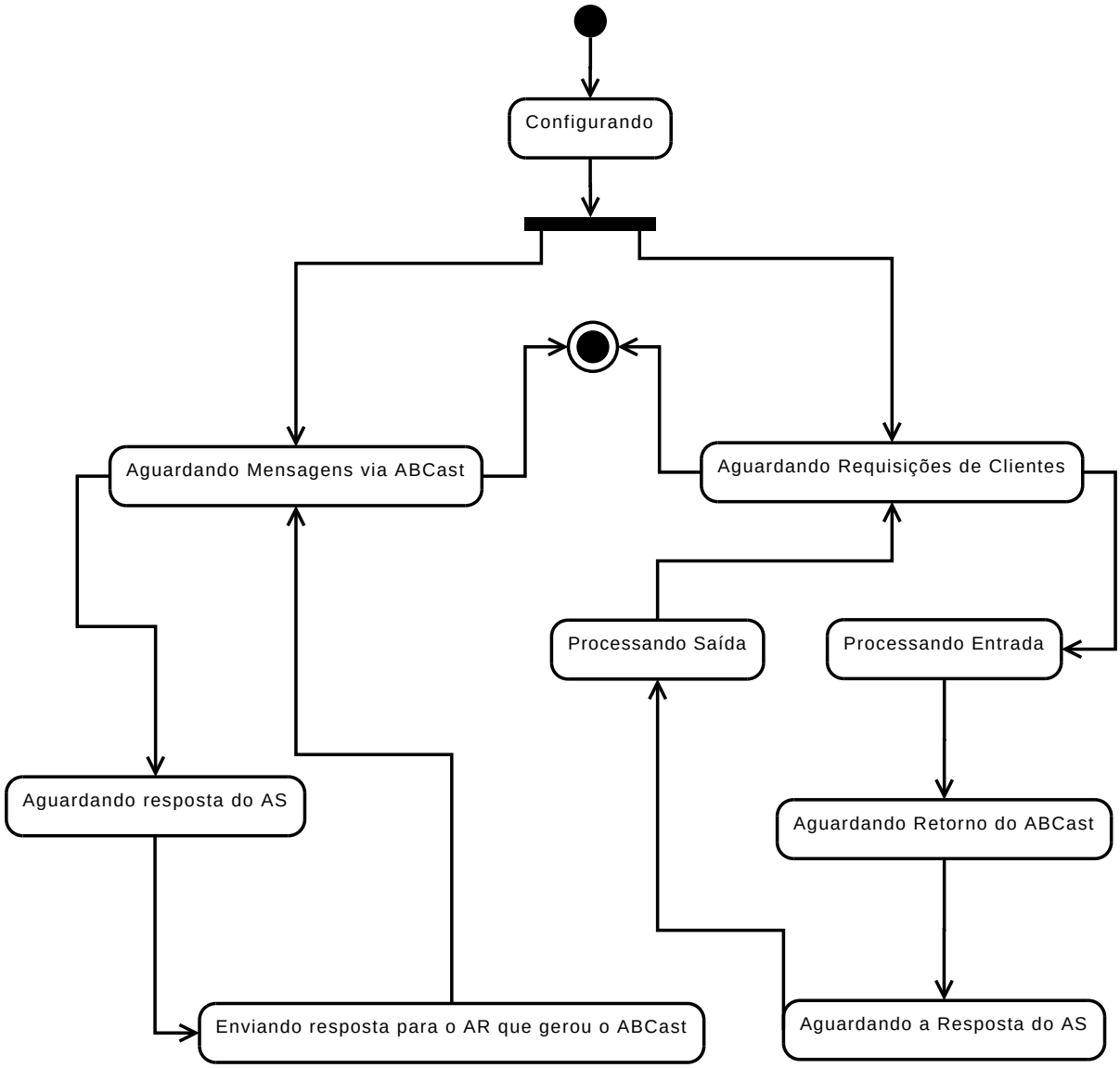


Figura 5.6: Estados assumidos por um AR na replicação ativa.

5.4 Tratamento de requisições duplicadas

Os padrões básicos de serviços web não prevêem a retransmissão de mensagens consideradas perdidas pelos clientes. No cenário comum assumido neste texto, o problema de recebimento de mensagens duplicadas não precisaria ser considerado. Entretanto, a extensão proposta no Capítulo 4 realiza a reinvocação de um serviço à próxima réplica da lista quando uma quebra é identificada. Contudo, esta funcionalidade possibilita a ocorrência de cenários em que uma mensagem possa vir a ser processada mais de uma vez pelas réplicas, veja a Figura 5.7.

Na Figura 5.7 a resposta para a requisição *Req1* do cliente demorou mais tempo (*timeout*) para ser processada do que o tempo de espera que a extensão permite. Por

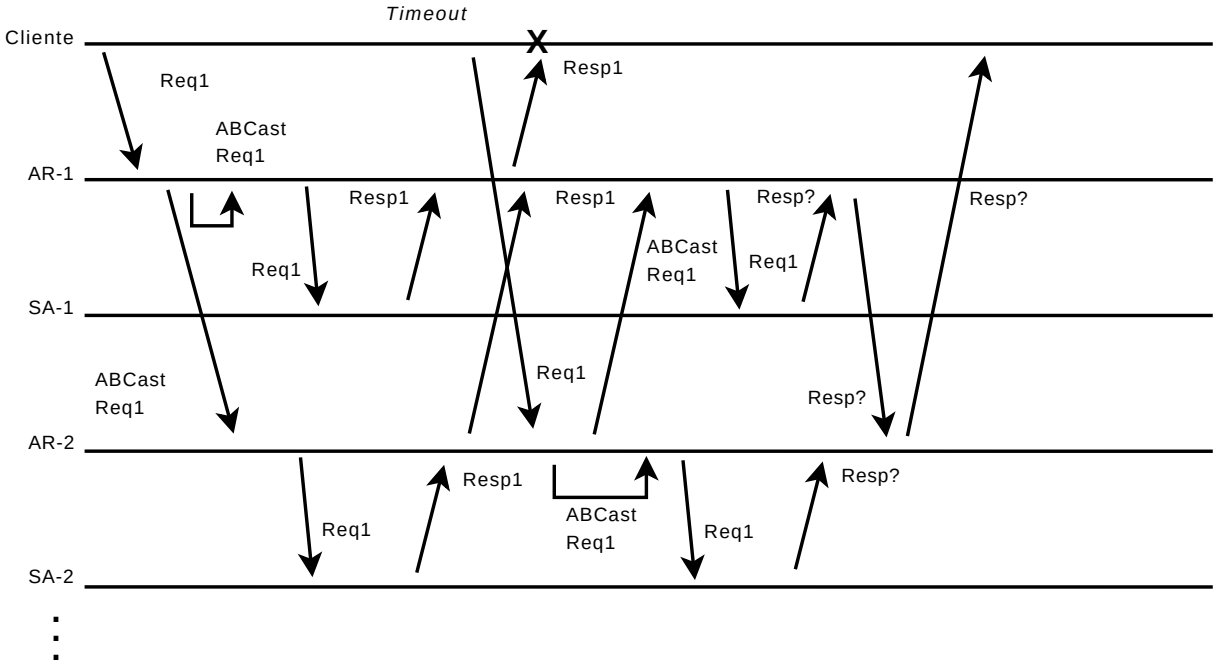


Figura 5.7: Processamento errôneo de mensagem retransmitida pela extensão.

consequência, o módulo do WS-Addressing estendido invoca a próxima réplica (AR2) da lista de réplicas. A mensagem, entretanto, foi processada pelas réplicas e quando AR2 recebe *Req1* novamente, segue o procedimento padrão de processamento de mensagens da replicação ativa. O problema deste comportamento é que o primeiro processamento de *Req1* pode ter mudado o estado das réplicas, invalidando a segunda resposta gerada (Resp?).

Duas possibilidades podem ser visualizadas para sanar o problema das requisições duplicadas. A primeira delas é utilizar a propriedade *MessageId* do WS-Addressing. Por se tratar de um identificador único de cada mensagem, uma possível solução seria armazenar todos os *MessageIds* recebidos e verificar a presença do *MessageId* de cada mensagem recebida neste conjunto. Esta solução é de simples implementação, mas bastante dispendiosa em termos de recursos computacionais quando se tem vários clientes acessando simultaneamente os serviços. Uma solução mais eficiente é atribuir um número de sequência para as mensagens, bastando então armazenar apenas o último número de sequência recebido. Esta é a solução utilizada pelo WS-ReliableMessaging, por exemplo, mas como este padrão não é um pré-requisito para o funcionamento do SimpleRep ou da extensão ao WS-Addressing, não podemos utilizar o campo definido pelo mesmo. Assim sendo, atribuir um número de sequência às mensagens através da própria extensão ao WS-Addressing foi o método adotado para solucionar o problema.

É importante ressaltar que o problema do reprocessamento em retransmissões só acontece com clientes que utilizam a extensão, devido à retransmissão de mensagens em caso de detecção de quebra. Clientes que não utilizam a extensão só podem vir a realizar retransmissões caso estas venham a ser implementadas na camada de negócios. Neste caso, o problema de recebimento de mensagens duplicadas também ocorre no cenário em que o

serviço não foi replicado, devendo portanto ser tratado na lógica de negócios do serviço.

O AR também é capaz de descartar mensagens duplicadas que venham a ser recebidas de um cliente que esteja utilizando a extensão do WS-Addressing.

5.5 Considerações em relação ao desempenho

Nos testes do SimpleRep foram utilizadas as mesmas configurações de servidores e de invocação de serviço definidas na Seção 4.3.6. Desta vez, três cenários foram testados:

1. **Cenário 1**: O SimpleRep não foi utilizado e o serviço foi invocado utilizando o Axis2 não modificado no cliente, sem a extensão.
2. **Cenário 2**: O SimpleRep foi utilizado, mas ainda sem a extensão no cliente.
3. **Cenário 3**: O serviço foi invocado utilizando a extensão juntamente ao SimpleRep.

Os resultados obtidos são ilustrados pela Figura 5.8. Conforme o esperado, a utilização do SimpleRep, de fato, ainda gera uma sobrecarga considerável de 25%, sem a extensão, e de 69% com a extensão. Podem ser apontadas, além das já citadas na Seção 4.3.6, como principais fontes de sobrecarga nos cenários que utilizam o SimpleRep:

- Replicação Ativa: para que uma resposta seja gerada ao cliente, há troca de mensagens na rede entre as réplicas para que o *Atomic Broadcast* funcione.
- Processamento de XML: os cabeçalhos SOAP das mensagens são sempre processados nos ARs (lista de réplicas e *SequenceID*).
- HTTP Proxy: de acordo com o próprio autor do componente, a implementação é rudimentar, disponibilizada para fins educativos.

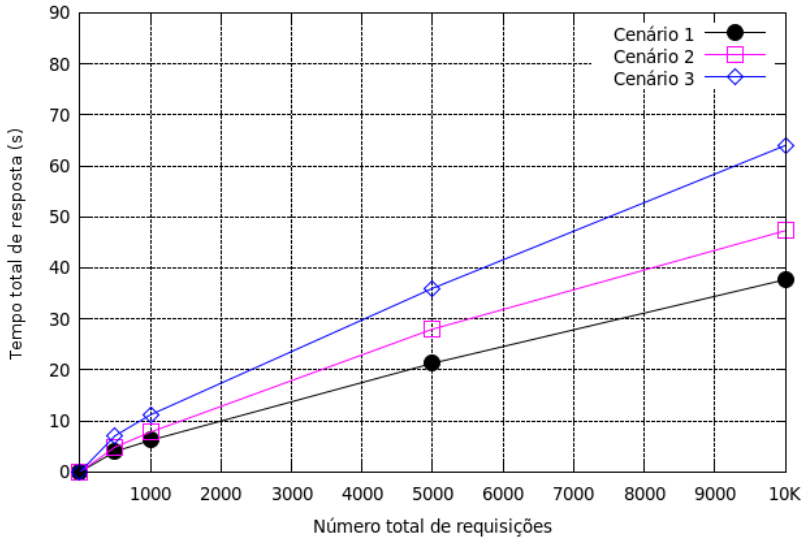


Figura 5.8: Tempos de resposta obtidos nos três cenários de testes com o SimpleRep.

5.6 Conclusão

Este capítulo mostrou que é possível implementar um *middleware* de TF em serviços web independentemente da plataforma ou da linguagem de programação utilizadas na implementação do serviço. Os resultados em relação ao desempenho, apesar de se tratar de uma implementação de prova de conceito, podem levantar questionamentos sobre o custo/benefício de se utilizar uma solução desse gênero. Entretanto, é preciso notar que todas as soluções de replicação possuem custos inerentes, sejam de troca de mensagens, trocas de estados entre as réplicas, escrita em *logs* e vários outros mecanismos necessários para se manter a consistência entre as réplicas.

O SimpleRep, atualmente, implementa apenas a replicação ativa. Contudo, o mesmo fornece uma base de código para a implementação de novos protocolos de replicação e, neste sentido, tanto a arquitetura do *middleware* quanto a utilização do Samoa podem se mostrar vantajosas no sentido de facilitar a construção de novas soluções. Estudos mais detalhados, em cenários mais realísticos, fazem-se necessários para se chegar a conclusões mais definitivas.

Capítulo 6

Considerações Finais

Todo este trabalho foi pautado pela busca de uma solução de tolerância a falhas para serviços web que fosse:

- **simples**: a implementação da solução não deve requerer conhecimentos profundos de TF por quem venha a utilizá-la.
- **genérica**: a maior promessa dos Serviços Web é a independência da plataforma de desenvolvimento e da linguagem de programação. Desta forma, é de grande interesse que um *framework* que vise a este tipo de arquitetura também o seja. Dentre os principais benefícios desta abordagem, podemos citar:
  - Escreve-se apenas um framework de tolerância a falhas
  - Fica-se imune ao esforço de troca de *framework* em caso de troca de tecnologia de implementação dos serviços.
  - Permite-se que a tecnologia mais adequada para o desenvolvimento do serviço seja utilizada.
- **transparente ao cliente**: em um ambiente tão heterogêneo, não é interessante realizar modificações nas várias implementações existentes.

Obviamente, outras características também possuem grande importância. Dentre elas, uma que geralmente possui grande peso é o desempenho. Entretanto, com os avanços no poder de processamento dos últimos anos, o fator desempenho vem, gradualmente, tendo sua importância reduzida. Uma evidência disto é o fato de a própria arquitetura de serviços web ser criticada por alguns, justamente por se utilizar de um padrão textual como o XML em seu alicerce, mas a sua adoção só tem crescido.

Desta forma, as duas soluções apresentadas neste trabalho fornecem uma proposta muito próxima dos objetivos principais perseguidos durante a pesquisa. A extensão ao WS-Addressing permite, de fato, a recuperação de falhas de forma transparente ao cliente. O que, por si só, já pode ser considerado um avanço em relação ao estado atual dos padrões de Serviços Web. Além disso, o SimpleRep consegue ser o mais genérico possível,

permitindo a sua utilização independentemente da plataforma de desenvolvimento do serviço.

O SimpleRep ainda não é uma solução completa e robusta, porém a arquitetura básica já é funcional. A tentativa de fornecer TF independemente de plataforma de desenvolvimento mostra-se promissora.

Um outro importante fator a se considerar é que, atualmente, os vários padrões de serviços web (WS-*) possuem uma implementação para cada plataforma/servidor de aplicação, gerando heterogeneidade entre as versões. Não é raro encontrar plataformas que ainda não suportam as últimas versões de determinados padrões, restringindo a sua utilização por usuários finais e desenvolvedores. Soluções como o SimpleRep permitem que o padrão possa ser implementado uma única vez e utilizado pelos mais variados SAs, um avanço interessante em relação ao estado atual.

Por fim, o SimpleRep, diferentemente da maioria das soluções existentes, foi concebido também com o foco na extensibilidade. Ao se basear na arquitetura do Axis2, o SimpleRep fornece abstrações (fluxos, *handlers* e contextos) que facilitam tanto a manipulação das mensagens quanto a agregação de novas funcionalidades de TF pelos desenvolvedores.

6.1 Contribuições

As duas principais contribuições deste trabalho são:

- A proposta de extensão ao WS-Addressing, adicionando o elemento *Replicas*, assim como uma implementação de referência, utilizando o Axis2.
- Uma proposta de middleware de tolerância a falhas em serviços web independente de plataforma de desenvolvimento, o SimpleRep.

Uma contribuição secundária deste trabalho foi a revisão bibliográfica sobre os principais trabalhos de tolerância a falhas em serviços web, assim como um comparativo entre eles.

6.2 Dificuldades Encontradas

Como o trabalho envolveu implementação, as maiores dificuldades envolveram principalmente a concretização das propostas em código.

A primeira dificuldade encontrada foi aprender como o código do Axis2 funciona. Uma característica comum entre os projetos de software livre é a falta de documentação técnica para sua extensão.

A segunda dificuldade foi aprender a utilizar o Samoa. O código do projeto é fruto, basicamente, do trabalho realizado durante o doutorado de Olivier Rütti [48], da École

Polytechnique Fédérale de Lausanne. O projeto do SimpleRep começou quando o Samoa ainda estava na versão 1.2, ainda com algumas falhas de implementação. Em determinados momentos foi necessário entrar em contato com o autor para que o projeto pudesse continuar caminhando. Entretanto, essas interações foram positivas, tanto para o SimpleRep quanto para o Samoa (um *patch* foi aceito pelo autor). Neste sentido, só podemos agradecer a sua disposição e receptividade.

6.3 Trabalhos Futuros

Além do que já foi exposto, é possível enxergar algumas outras possibilidades de evolução do estado atual do projeto que o tornariam mais próximo de ser utilizado na prática:

- Implementação de outros estilos de replicação, como a passiva e a semi-passiva.
- Tornar o *proxy* reverso modular e independente de protocolo de transporte (ver Seção 5.1), para permitir a implementação de outros protocolos de transporte (SMTP, JMS, etc).
- Permitir que os clientes realizem requisições não-bloqueantes.
- Adicionar ao SimpleRep uma funcionalidade de instanciação de serviços nas réplicas. Atualmente, o mesmo serviço precisa ser instanciado em cada SA, um por vez.
- Processar requisições de documentos WSDL. Todos os testes realizados neste projeto já possuíam clientes que acessavam os serviços antes da replicação. Em um cenário real, é necessário que as descrições dos serviços apontem para os endereços dos ARs, e não para os SAs. Tornar o SimpleRep capaz de alterar os endereços das WSDLs dos serviços dinamicamente é fundamental para a sua utilização na prática.
- Implementar funcionalidades de balanceamento de carga na replicação ativa.
- Implementar funcionalidades que facilitem a implementação de replicação *n-Version* (ver Seção 3.2.9).

Referências Bibliográficas

[1] Esam Alwagait e Shahram Ghandeharizadeh. DeW: A Dependable Web Services Framework. In *RIDE '04: Proceedings of the 14th International Workshop on Research Issues on Data Engineering: Web Services for E-Commerce and E-Government Applications (RIDE'04)*, páginas 111–118. IEEE Computer Society, Washington, DC, USA, 2004. ISBN 0-7695-2095-2.

[2] Yair Amir, Claudiu Danilov, e Jonathan Stanton. A Low Latency, Loss Tolerant Architecture and Protocol for Wide Area Group Communication. In *FTCS 2000*. 2000.

[3] Algirdas Avizienis. The N-Version Approach to Fault-Tolerant Software. *IEEE Transactions on Software Engineering*, SE-11(12):1491–1501, dezembro 1985.

[4] Bela Ban. JGroups: A Toolkit for Reliable Multicast Communication. Disponível em: http://www.jgroups.org. Acesso em: 13 out. 2010.

[5] Alberto Bartoli, Ricardo Jiménez-Peris, Bettina Kemme, Cesare Pautasso, Simon Patarin, Stuart Wheater, e Simon Woodman. The Adapt Framework for Adaptable and Composable Web Services. *IEEE Distributed Systems Online*, setembro 2005.

[6] Ken Birman, Robbert van Renesse, e Werner Vogels. Adding High Availability and Autonomic Behavior to Web Services. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, páginas 17–26. IEEE Computer Society, Washington, DC, USA, 2004. ISBN 0-7695-2163-0.

[7] Kenneth P. Birman. *Reliable Distributed Systems: Technologies, Web Services and Applications*. Springer, New York, 2005.

[8] Kenneth P. Birman. The Untrustworthy Web Services Revolution. *Computer*, 39(2):98, 2006. ISSN 0018-9162.

[9] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, e Sam Toueg. The Primary-Backup Approach. In *Distributed systems (2nd Ed.)*, páginas 199–216. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993. ISBN 0-201-62427-3.

[10] Miguel Castro, Rodrigo Rodrigues, e Barbara Liskov. BASE: Using Abstraction to Improve Fault Tolerance. *ACM Transactions on Computer Systems (TOCS)*, 21(3):236–269, 2003. ISSN 0734-2071.

[11] Tushar Deepak Chandra e Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal Of The ACM*, 43(2):225–267, 1996. ISSN 0004-5411.

[12] Martinho Correia e Jorge Cardoso. Aumento da resiliência dos web services com uma infra-estrutura peer-to-peer. In *6ª Conferência da Associação Portuguesa de Sistemas de Informação*. outubro 2005. ISBN 972-95246-3-7.

[13] George Coulouris, Jean Dollimore, e Tim Kindberg. *Distributed Systems: Concepts and Design*. International Computer Science Series. Addison Wesley, Reading, Massachusetts, 4ª edição, maio 2005. ISBN 0321263545.

[14] José Ricardo da Silva e Irineu Sotoma. Failover Transparente ao Cliente em Serviços Web: uma Extensão ao WS-Addressing. In *Workshop de Testes e Tolerância a Falhas (WTF 2009)*. João Pessoa - PB, Brazil, agosto 2009.

[15] Xavier Défago e André Schiper. Semi-passive replication and lazy consensus. *Journal of Parallel and Distributed Computing*, 64(12):1380–1398, dezembro 2004.

[16] Glen Dobson. Using WS-BPEL to Implement Software Fault Tolerance for Web Services. In *EUROMICRO '06: Proceedings of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications*, páginas 126–133. IEEE Computer Society, Washington, DC, USA, 2006. ISBN 0-7695-2594-6.

[17] Robert A. Van Engelen e Kyle A. Gallivan. The gSOAP Toolkit for Web Services and Peer-to-Peer Computing Networks. In *CCGRID '02: Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, página 128. IEEE Computer Society, Washington, DC, USA, 2002. ISBN 0-7695-1582-7.

[18] Chen-Liang Fang, Deron Liang, Fengyi Lin, e Chien-Cheng Lin. Fault Tolerant Web Services. *Journal of Systems Architecture*, 53(1):21–38, 2007. ISSN 1383-7621.

[19] Michael J. Fischer, Nancy A. Lynch, e Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985. ISSN 0004-5411.

[20] Ian Foster, Savas Parastatidis, Paul Watson, e Mark Mckeown. How do I model state?: Let me count the ways. *Communications Of The ACM*, 51(9):34–41, 2008. ISSN 0001-0782.

[21] The Apache Software Foundation. Apache HttpComponents. Disponível em: http://hc.apache.org. Acesso em: 13 out. 2010.

[22] Geoffrey Fox, Shrideep Pallickara, e Savas Parastatidis. Towards Flexible Messaging for SOAP Based Services. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, página 8. IEEE Computer Society, Washington, DC, USA, 2004.

[23] Seth Gilbert e Nancy Lynch. Brewer's conjecture and the feasibility of consistent, availabel, partition-tolerant web services. *ACM SIGACT News Distributed Computing*, 33(2):51–59, maio 2002.

[24] Mark Hansen. *Soa Using Java(Tm) Web Services*. Prentice Hall PTR, Upper Saddle River, 2007. ISBN 0130449687.

[25] Mark Hayden e Kenneth Birman. Probabilistic Broadcast. Relatório técnico, Cornell University, Ithaca, NY, USA, 1996.

[26] Michael N. Huhns e Munindar P. Singh. Service-oriented computing: Key concepts and principles. *IEEE Internet Computing*, 9(1):75–81, fevereiro 2005. ISSN 1089-7801.

[27] Deepal Jayasinghe. FAWS for SOAP-based Web services. Disponível em: http://www-128.ibm.com/developerworks/webservices/library/ws-faws. Acesso em: 13 out. 2010.

[28] José Geraldo R. Junior, Glauber Tadeu S. Carmo, e Marco Tulio O. Valente. Invocation of Replicated Web Services Using Smart Proxies. In *WebMedia '06: Proceedings of the 12th Brazilian symposium on Multimedia and the web*, páginas 138–147. ACM, New York, NY, USA, 2006. ISBN 85-7669-100-0.

[29] Christos T. Karamanolis e Jeffrey N. Magee. Client-access protocols for replicated services. *IEEE Trans. Softw. Eng.*, 25(1):3–21, 1999. ISSN 0098-5589.

[30] Leslie Lamport, Robert Shostak, e Marshall Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982. ISSN 0164-0925.

[31] Jean-Claude Laprie e Brian Randell. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004. ISSN 1545-5971. Fellow-Algirdas Avizienis and Senior Member-Carl Landwehr.

[32] Nik Looker e Malcolm Munro. WS-FTM: A Fault Tolerance Mechanism for Web Services. Relatório técnico, University of Durham, University of Durham, março 2005.

[33] Nik Looker, Malcolm Munro, e Jie Xu. WS-FIT: A Tool for Dependability Analysis of Web Services. In *COMPSAC '04: Proceedings of the 28th Annual International Computer Software and Applications Conference - Workshops and Fast Abstracts - (COMPSAC'04)*, páginas 120–123. IEEE Computer Society, Washington, DC, USA, 2004. ISBN 0-7695-2209-2-2.

[34] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, San Franciso, California, 1996. ISBN 1-55860-348-4.

[35] Michael G. Merideth, Arun Iyengar, Thomas Mikalsen, Stefan Tai, Isabelle Rouvellou, e Priya Narasimhan. Thema: Byzantine-Fault-Tolerant Middleware for Web-Service Applications. In *SRDS '05: Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)*, páginas 131–142. IEEE Computer Society, Washington, DC, USA, 2005. ISBN 0-7695-2463-X.

[36] Anton Michlmayr, Florian Rosenberg, Christian Platzer, Martin Treiber, e Schahram Dustdar. Towards Recovering the Broken SOA Triangle: A Software Engineering Perspective. In *IW-SOSWE '07: 2nd International Workshop on Service Oriented Software Engineering*, páginas 22–28. ACM, New York, NY, USA, 2007. ISBN 978-1-59593-723-0.

[37] Microsoft Corporation. Microsoft Launches XML Web Services Revolution With Visual Studio .NET and .NET Framework, 2002. Disponível em: http://www.microsoft.com/presspass/press/2002/feb02/02-13RevolutionPR.mspx. Acesso em: 12 out. 2010.

[38] L. E. Moser, P. M. Melliar-Smith, e Wenbing Zha. Making Web Services Dependable. In *ARES '06: Proceedings of the First International Conference on Availability, Reliability and Security (ARES'06)*, páginas 440–448. IEEE Computer Society, Washington, DC, USA, 2006. ISBN 0-7695-2567-9.

[39] OASIS. Web Services Reliable Messaging (WS-ReliableMessaging), fevereiro 2007. Disponível em: http://docs.oasis-open.org/ws-rx/wsrm/200702/wsrm-1.2-spec-os.html. Acesso em: 13 out. 2010.

[40] OGM. Object Management Group, Fault Tolerant CORBA specification, OMG Technical Comittee Document. Capítulo 23., 2004. Disponível em: http://www.omg.org/cgi-bin/doc?formal/2004-03-12. Acesso em: 24 out. 2010.

[41] Johannes Osrael, Lorenz Froihofer, e Karl M. Goeschka. What Service Replication Middleware Can Learn from Object Replication Middleware. *Proceedings of the 1st workshop on Middleware for Service Oriented Computing (MW4SOC 2006)*, páginas 18–23, dez. 2006. Institute of Information Systems, Vienna University of Technology.

[42] Johannes Osrael, Lorenz Froihofer, Martin Weghofer, e Karl M. Goeschka. Axis2-based Replication Middleware for Web Services. *IEEE International Conference on Web Services (ICWS)*, páginas 591–598, julho 2007.

[43] M. P. Papazoglou e D. Georgakopoulos. Service-oriented Computing. *Communications of the ACM*, 46(10):25–28, 2003. ISSN 0001-0782.

[44] C. Partridge, T. Mendez, e W. Milliken. Host Anycasting Service. RFC 1546, novembro 1993.

[45] PASOA Consortium. PreServ, 2007. Disponível em: http://www.pasoa.org. Acesso em: 13 out. 2010.

[46] Srinath Perera, Chathura Herath, Jaliya Ekanayake, Eran Chinthaka, Ajith Ranabahu, Deepal Jayasinghe, Sanjiva Weerawarana, e Glen Daniels. Axis2, Middleware for Next Generation Web Services. In *ICWS '06: Proceedings of the IEEE International Conference on Web Services (ICWS'06)*, páginas 833–840. IEEE Computer Society, Washington, DC, USA, 2006. ISBN 0-7695-2669-1.

[47] Pervasive Technology Labs, Indiana University. The NaradaBrokering Project. Disponível em: http://www.naradabrokering.org. Acesso em: 24 out. 2010.

[48] Olivier Rütti. *Concurrency and Dynamic Protocol Update for Group Communication Middleware*. Tese de Doutoramento, École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland, 2009. Number 4244.

[49] Jorge Salas, Francisco Perez-Sorrosal, Marta Patiño-Martínez, e Ricardo Jiménez-Peris. WS-Replication: A Framework For Highly Available Web Services. In *WWW '06: Proceedings of the 15th international conference on World Wide Web*, páginas 357–366. ACM Press, New York, NY, USA, 2006. ISBN 1-59593-323-9.

[50] Nicolas Salatge e Jean-Charles Fabre. Fault tolerance connectors for unreliable web services. In *DSN '07: Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, páginas 51–60. IEEE Computer Society, Washington, DC, USA, 2007. ISBN 0-7695-2855-4.

[51] Giuliana Teixeira Santos, Lau Cheuk Lung, e Carlos Montez. FTWeb: A Fault Tolerant Infrastructure for Web Services. In *EDOC '05: Proceedings of the Ninth IEEE International EDOC Enterprise Computing Conference (EDOC'05)*, páginas 95–105. IEEE Computer Society, Washington, DC, USA, 2005. ISBN 0-7695-2441-9.

[52] Fred B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990. ISSN 0360-0300.

[53] Abraham Silberschatz, Peter Baer Galvin, e Greg Gagne. *Operating Systems Concepts with Java*. Wiley, Hoboken, New Jersey, 7ª edição, 2007. ISBN 978-0-471-76907-1.

[54] José Airton F. Silva e Nabor C. Mendonça. Implementation and Empirical Evaluation of Server Selection Policies for Accessing Replicated Web Services. In *Anais do XXV Congresso da Sociedade Brasileira de Computação (SBC'05), 2005.*, XVIII Concurso de Teses e Dissertações (CTD'05), páginas 75–79. 2005.

[55] Sun Microsystems, Inc. JXTA Technology. Disponível em: http://www.sun.com/software/jxta. Acesso em: 24 out. 2010.

[56] The Apache Software Foundation. Apache Axis2/Java - Next Generation Web Services. Disponível em: http://ws.apache.org/axis2. Acesso em: 12 out. 2010.

[57] The Apache Software Foundation. Apache Derby. Disponível em: http://db.apache.org/derby. Acesso em: 24 out. 2010.

[58] The Apache Software Foundation. Apache ODE. Disponível em: http://ode.apache.org. Acesso em: 13 out. 2010.

[59] The Apache Software Foundation. Web Services - Axis. Disponível em: http://ws.apache.org/axis. Acesso em: 12 out. 2010.

[60] Paul Townend, Paul Groth, Nik Looker, e Jie Xu. FT-Grid: A Fault-Tolerance System for e-Science. In *Proceedings of the UK e-Science All Hands Meeting 2005*. Nottingham UK, setembro 2005.

[61] Universidad Politécnica de Madrid. ADAPT Middleware Technologies for Adaptive and Composable Distributed Components (IST-37126), 2007. Disponível em: http://adapt.ls.fi.upm.es/adapt. Acesso em: 24 out. 2010.

[62] W3C. Web Services Addressing 1.0 - Core, 2006. Disponível em: http://www.w3.org/TR/ws-addr-core. Acesso em: 03 nov. 2007.

[63] W3C. Web Services Addressing 1.0 - SOAP Binding, 2006. Disponível em: http://www.w3.org/TR/ws-addr-soap. Acesso em: 02 dez. 2008.

[64] W3C. WS-Addressing 1.0 Latest XML Schema, 2006. Disponível em: http://www.w3.org/2006/03/addressing/ws-addr.xsd. Acesso em: 17 nov. 2008.

[65] W3C. Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language, jun. 2007. Disponível em: http://www.w3.org/TR/wsdl20.

[66] W3C. Extensible Markup Language (XML), 2008. Disponível em: http://www.w3.org/XML. Acesso em: 12 out. 2010.

[67] Chan Pik Wah. Reliable Web Services by Fault Tolerant Techniques: Methodology, Experiment, Modeling and Evaluation. Relatório técnico, The Chinese University of Hong Kong Shatin, Hong Kong, 2006.

[68] Sanjiva Weerawarana, Francisco Curbera, Frank Leymann, Tony Storey, e Donald F. Ferguson. *Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More*. Prentice Hall, Upper Saddle River, 2005.

[69] Xinfeng Ye e Yilin Shen. A Middleware for Replicated Web Services. In *ICWS'05: Proceedings of the IEEE International Conference on Web Services (ICWS'05)*, páginas 631–638. IEEE Computer Society, Washington, DC, USA, 2005. ISBN 0-7695-2409-5.

[70] École Polytechnique Fédérale de Lausanne. Samoa Project. Disponível em: http://lsrwww.epfl.ch/page13488.html. Acesso em: 24 out. 2010.

Apêndice A

Detalhes de utilização

Os capítulos anteriores apresentaram os principais conceitos das propostas feitas por este trabalho (a extensão e o SimpleRep), fornecendo uma visão de alto nível das soluções. Este apêndice visa a fornecer mais detalhes técnicos sobre a implementação dos respectivos protótipos e também sobre a utilização dos mesmos, servindo de referência tanto para futuros utilizadores quanto para futuros colaboradores. A Seção A.1 fornece um exemplo de configuração do SimpleRep e a Seção A.2 apresenta um exemplo de *handler* que pode ser adicionado ao SimpleRep para o tratamento de mensagens.

A.1 Configurando o SimpleRep

O SimpleRep precisa que alguns parâmetros de inicialização sejam definidos para que o mesmo possa ser executado. Os parâmetros são:

- **port**: a porta TCP pela qual o AR irá receber requisições.
- **transportProtocol**: protocolo da camada de transporte. No protótipo deste trabalho, apenas o HTTP está implementado.
- **timeout**: intervalo (em milissegundos) de detecção de falhas (utilizado pelo detector de falhas do Samoa).
- **replicas**: endereços dos outros ARs. Formato: [endereço IP ou *hostname*]:[porta].
- **style**: tipo de replicação. Pode ser ativa (*active*) ou passiva (*passive*).
- **samoa**: endereço da instância local do Samoa.
- **appserver**: endereço do servidor de aplicação responsável pelo fornecimento do serviço.

A definição destes parâmetros é feita em um único arquivo XML. Assim, cada agente de replicação (instância do SimpleRep) deve possuir um arquivo de configuração semelhante ao da Figura A.1.

```
<simplerepconf>

  <!-- AR conf -->
  <port>8081</port>
  <transportProtocol>http</transportProtocol>
  <timeout>1000</timeout>

  <!-- Endereço das instâncias do Samoa -->
  <replicas style="active">
    <replica>10.87.6.156:6666</replica>
    <replica>10.87.6.157:6666</replica>
    <replica>10.87.6.158:6666</replica>
  </replicas>

  <proxies>
    <proxy>http://10.87.6.158:8081</proxy>
    <proxy>http://10.87.6.157:8081</proxy>
    <proxy>http://10.87.6.156:8081</proxy>
  </proxies>

  <!-- Samoa configuration for this instance -->
  <samoa>
    <host>10.87.6.158</host>
    <port>6666</port>
  </samoa>

  <!-- Endereço do SA -->
  <appserver>
    <host>localhost</host>
    <port>8080</port>
  </appserver>
</simplerepconf>
```

Figura A.1: Arquivo de configuração de exemplo.

A.2 Um exemplo de *handler*

Suponha que o número de sequência de uma requisição seja necessário para manipular a sua resposta no SimpleRep (através de um *handler* localizado no *OutFlow*). A Figura A.2 fornece um exemplo de *handler* que poderia ser implementado para realizar a tarefa de armazenar o número de sequência no Contexto de Operação, que posteriormente poderá ser recuperado por um *handler* no *OutFlow*.

```
package br.ufms.dct.simplerep.handlers;

import java.util.Iterator;
import org.apache.axiom.soap.SOAPHeader;
import org.apache.axiom.soap.SOAPHeaderBlock;
import br.ufms.dct.simplerep.ar.MessageContext;
import br.ufms.dct.simplerep.ar.ProcessingStatus;
import br.ufms.dct.simplerep.enums.AddressingConstants;
import br.ufms.dct.simplerep.enums.SimpleRepConstants;

public class AddressingInSeqIdHandler implements AbstractHandler {
   public ProcessingStatus invoke(MessageContext context) {
      System.out.println("[AddressingInSeqIdHandler] invoke()");

      SOAPHeader soapHeader = context.getEnvelope().getHeader();
      int seqId = -1;

      String wsa = AddressingConstants.ADDRESSING_NAMESPACE;
      Iterator headers = soapHeader.getHeadersToProcess(null, wsa);

      while (headers.hasNext()) {
            SOAPHeaderBlock soapHeaderBlock = (SOAPHeaderBlock)headers.next();
            String localName = soapHeaderBlock.getLocalName();

            if (localName.equals("SequenceID")) {
               seqId = Integer.parseInt(soapHeaderBlock.getText());
            }
      }

      if (seqId < 0) {
         // O número de sequência não foi recebido
         // O cliente não está usando a extensão ao WS-Addressing
         return ProcessingStatus.CONTINUE;
      }

      // Vamos precisar do número de sequência no OutFlow
      // Armazenando no Contexto de Operação
      OperationContext opCtxt = context.getOperationContext();
      opCtxt.set(SimpleRepConstants.RECEIVED_SEQUENCE_ID, new Integer(seqId));
  }
}
```

Figura A.2: Um *handler* que armazena o número de sequência no Contexto de Operação.

A.3 As threads do SimpleRep

Visando a facilitar a compreensão do funcionamento do código do SimpleRep, nesta seção são apresentadas as principais *threads* do *middleware* e o fluxo de mensagens entre elas durante o processamento de uma requisição na replicação ativa. Uma visão geral é ilustrada pela Figura A.3.

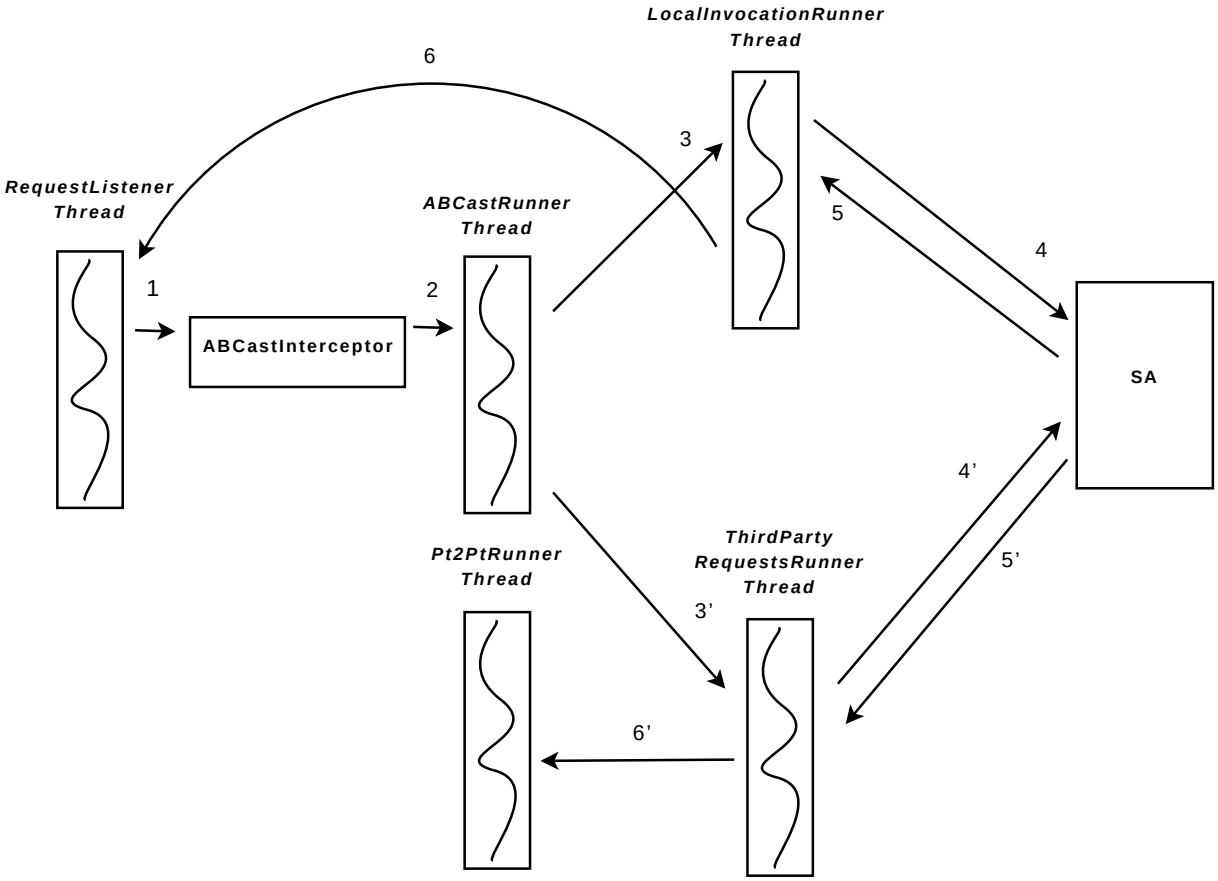


Figura A.3: As principais *threads* do SimpleRep.

Na replicação ativa, há dois comportamentos possíveis para um AR. Suponha que o sistema possua 5 réplicas e, por consequência, 5 ARs (AR-1, AR-2, ..., AR-5). Se a solicitação do cliente é realizada ao AR-1, por exemplo, o caminho percorrido pela mensagem será o indicado pelos números de 1 a 6 na Figura A.3. Neste caso, todos os outros ARs (AR-[2-5]) processarão a mensagem seguindo o caminho de 3' a 6'.

Seguindo o cenário descrito anteriormente, cada instante, ilustrado por um número na Figura A.3, é detalhado a seguir:

- 1: Uma requisição chega do cliente ao AR-1 através da *thread RequestListener*.
- 2: No início do processamento da requisição, o componente *ABCastInterceptor*[1] realiza o *Atomic Broadcast* da mensagem. Ao término do *Atomic Broadcast* uma

[1]A biblioteca *HttpComponents* permite que interceptadores sejam adicionados ao *Proxy HTTP*.

rotina de *callback* é invocada. Este *callback*, em tempo de execução, determina se o AR é o AR que recebeu a requisição do cliente. Em caso afirmativo, segue-se para o passo 3. Caso contrário, segue-se para o passo 3'.

- 3: O serviço é requisitado (mensagens 4 e 5) ao SA correspondente ao AR que recebeu a invocação do cliente.
- 6: A resposta do SA é enviada ao cliente, caso uma resposta de outro AR não tenha ainda sido recebida (ver a otimização feita no SimpleRep na Seção 5.3 e na Figura 5.4).
- 3': O serviço é requisitado (mensagens 4' e 5') aos SAs correspondentes a cada um dos outros ARs, que receberam a requisição via *Atomic Broadcast*.
- 6': A resposta é enviada ao AR que recebeu a requisição do cliente (AR-1, no cenário de exemplo).