Ampliando a Disponibilidade e Confiabilidade em Ambientes de Serviços Web *Stateful*

Igor Nogueira Santos[1], **Daniela Barreiro Claro**[1], **Marcelo Luz**[1]

[1]Laboratório de Sistemas Distribuídos (LaSiD)
Departamento de Ciência da Computação
Instituto de Matemática / Universidade Federal da Bahia (UFBA)
Av. Adhemar de Barros, s/n, Ondina – Salvador – BA – Brasil

igorns@gmail.com, dclaro@ufba.br, celoluz@gmail.com

Abstract. *Web services are being widely used in applications which require high availability and reliability. Several specifications have been created in order to standardize the use of reliable mechanisms on Web services. Web services have been replicated willing to improve its availability. Considering that WS are autonomous and heterogeneous, their replication is even harder when there is state maintenance because web services are developed by different organizations in different ways. This paper evaluates some related work and introduces an hybrid and a passive replication mechanism with state maintenance on web services. Such approach was evaluated throw a local network so as to analyze the overhead obtained and was developed for Axis2 because this engine is widely used for development of web services. Our results presented a satisfactory performance in order to guarantee the replication in stateful web services.*

Resumo. *Os serviços web estão sendo cada vez mais utilizados em aplicações que demandam alta disponibilidade e confiabilidade. Diversas especificações têm sido criadas com o intuito de padronizar a utilização de mecanismos confiáveis para serviços web. Serviços web têm sido replicados, ampliando, consequentemente, a sua disponibilidade. Considerando que os serviços web são autônomos e heterogêneos além de manter o estado (stateful web service), a replicação de serviços web é uma tarefa árdua e complexa, visto que diferentes empresas podem publicar seus serviços em maneiras distintas. O presente trabalho propõe avaliar trabalhos relacionados e introduzir um mecanismo de replicação passiva e híbrida com a manutenção de estados. Este mecanismo foi avaliado em uma rede local com o intuito de analisar o overhead causado, além de ter sido desenvolvido para o ambiente Axis2, visto ser atualmente o ambiente mais utilizado para o desenvolvimento de serviços web. Os resultados apresentaram um desempenho satisfatório em relação à garantia de replicação de serviços web stateful.*

1. Introdução

A crescente utilização de sistemas computacionais leva a uma necessidade de tolerar falhas, especialmente em sistemas críticos. Estes sistemas têm sido, cada vez mais, incrementados com o intuito de serem confiáveis o suficiente para os usuários. A disseminação

das redes, principalmente a Internet, tem aumentado o número de sistemas críticos distribuídos. Alguns destes sistemas utilizam o formato de serviços *web* (Web Services-WS) para publicar as suas funcionalidades, principalmente porque os WS utilizam tecnologias padronizadas (SOAP e WSDL). Além disso, eles são autônomos e heterogêneos, ou seja, são desenvolvidos e publicados independentemente de linguagens, protocolos ou interfaces.

Nos últimos anos, algumas especificações foram desenvolvidas com o intuito de aprimorar as características de confiabilidade nos serviços *web*, ampliando, assim, a sua utilização nestes sistemas críticos. Segurança (WS-Security [Lawrence and Kaler 2004]) e mensagens confiáveis (WS-ReliableMessaging [Iwasa et al. 2004]) são exemplos destas especificações. Alta disponibilidade, um importante critério em sistemas confiáveis (*dependable systems* [Avizienis et al. 2004]), é um dos aspectos que ainda não foram tratados por nenhuma especificação. Um meio de aprimorar a disponibilidade em sistemas distribuídos é replicando cada componente em diferentes servidores. Assim, se um componente falha, outro pode substituí-lo. Especificamente para WS, esta abordagem possui algumas dificuldades devido à heterogeneidade das plataformas onde os mesmos são publicados, impedindo, consequentemente, a determinação de um protocolo de replicação. Além disso, caso o estado de um serviço *web* seja considerado (*stateful web* services), um processo de sincronização de estados deve ser utilizado.

Com o intuito de sobrepor estas limitações, alguns *middlewares* de replicação têm sido propostos, tais como: FT-SOAP [Chen 2007], WS-Replication [Jiménez-Peris et al. 2006], Conectores Tolerantes a Falhas [Fabre and Salatge 2007] e replicação híbrida [Froihofer et al. 2007]. O presente trabalho apresenta um mecanismo transparente de replicação passiva e híbrida para serviços *web stateful* desenvolvidos através do ambiente Axis2 [Axis2 2010]. Este mecanismo garante que os estados dos serviços sejam mantidos, mesmo na presença de falhas. Diferente dos mecanismos propostos anteriormente que implementam replicação passiva ou híbrida e requerem uma modificação específica no WS desenvolvido, nenhuma modificação no WS é requerida nesta proposta. Além disso, este trabalho reduz a computação em cada réplica, visto que ele processa requisições somente na presença de defeitos.

O presente trabalho está organizado como se segue: a seção 2 apresenta alguns aspectos de tolerância a falhas. Na seção 3, são apresentados os trabalhos relacionados com o intuito de melhor posicionar o presente trabalho. A seção 4 apresenta a implementação proposta e os algoritmos desenvolvidos. A seção 5 apresenta os experimentos realizados e os resultados. Seção 6 apresenta as conclusões e os trabalhos futuros.

2. Aspectos de Tolerância a falhas e Serviços *web*

Confiabilidade (*dependability*) é um critério de qualidade composto por outros critérios tais com Integridade, Manutenibilidade e Disponibilidade [Avizienis et al. 2004]. Estes critérios se tornam mais importantes quando as interações com os WS ficam automáticas. Mais ainda, um defeito pode não somente afetar um único WS, mas uma composição deles. Assim, técnicas de tolerância a falhas têm sido amplamente utilizadas, visto que elas podem garantir a continuidade da execução de um serviço devido ao seu mecanismo de redundância, mesmo na presença de falhas.

Alguns mecanismos de replicação podem ser utilizados para tolerar falhas usando

um conjunto especial de características. Estas características determinam quão adaptado estes mecanismos estão quando são aplicados em um contexto específico. Um contexto específico, por exemplo, um sistema distribuído, é basicamente suportado por processos e comunicações [Cristian 1991]. Técnicas de tolerância a falhas, quando aplicadas a este contexto (sistema distribuído), implementam a sua redundância através do hardware, por exemplo, replicando os servidores em diferentes locais e, consequentemente, os seus *softwares*. Entre os principais modelos de falha que um sistema distribuído pode lidar, destacam-se os seguintes:

1. Falhas Bizantinas: servidores podem ter um comportamento malicioso, provavelmente se unindo a outros servidores falhos;
2. Falhas por Omissão: se um servidor perder mensagens mesmo que o risco de dano seja mínimo, então este servidor pode falhar por omissão;
3. Falhas por Parada (*Crash*): se depois de uma primeira omissão o servidor parar de mandar ou receber mensagens até ele ser reinicializado, então houve uma parada silenciosa neste servidor;
4. Falhas Temporais: ocorre quando uma requisição que deveria receber uma resposta dentro de um intervalo de tempo tem um atraso. Estas falhas ocorrem em sistemas síncronos, quando um tempo máximo é pré-estabelecido. No caso do servidor ultrapassar este limite de tempo, então este servidor é suspeito de falha.

   Neste trabalho, somente as falhas por parada silenciosas (*crash*) são tratadas.

Um mecanismo de replicação requer a manutenção de um estado consistente entre as réplicas. Isso pode assegurar que na presença de uma falha no servidor, o serviço pode continuar sua execução utilizando outra réplica a partir do ponto onde ocorreu a falha. Comunicação em grupo representa um dos principais meios de se construir um sistema distribuído replicado. Neste sentido, um conjunto de processos têm suas atividades coordenadas com o intuito de manter um estado consistente quando suas funções são executadas. Detecção de falhas, entrega *multicast* e ordem das mensagens são características fundamentais de comunicação em grupo e são muito utilizadas no desenvolvimento de mecanismos (*middlewares*) de replicação.

A ordenação das mensagens representa a manipulação de todas as requisições realizadas para um grupo e é essencial para manter o estado consistente dentro de cada réplica. Existem dois tipos básicos de ordenação:

1. Ordenação FIFO: entrega das mensagens para todos os membros segundo ordenação FIFO;
2. Ordenação Total: garante que todos os membros recebem todas as mensagens na mesma ordem, assim que as mensgens são recebidas durante o tempo.

O mecanismo de replicação utiliza um protocolo para garantir a sincronização do estado entre as réplicas. O protocolo de replicação pode ser dividido em dois grupos: protocolo com o processamento moderado e protocolo com o processamento redundante [Défago and Schiper 2001].

2.1. Processamento Moderado (*Parsimonious Processing*)

O principal protocolo que aplica o processamento moderado é a replicação passiva. Nesse protocolo, todas as requisições dos clientes são processadas por uma única réplica - a

réplica primária. As outras réplicas, chamadas *backup*, recebem somente atualizações de estado do servidor primário. Dessa forma, a ordenação FIFO de mensagens é suficiente para garantir a consistência entre as réplicas, já que somente as mensagens do membro primário são ordenadas.

Na falha da réplica primária, outra deve assumir seu lugar. Porém, durante o intervalo de tempo específico no qual o novo membro primário estiver sendo eleito, as novas requisições dos clientes serão perdidas, já que não existirá um servidor primário para processá-las. Nesse sentido, a replicação passiva clássica não é totalmente transparente ao usuário, de forma que o mesmo poderá necessitar re-enviar a mensagem, caso nenhuma resposta seja retornada. Outras abordagens de processamento moderado estão sendo utilizadas para contornar esta limitação de maneira eficiente, dentre elas, destacam-se a *coodination-cohort* e a replicação semi-passiva [Défago and Schiper 2001].

2.2. Processamento Redundante (*Redundant Processing*)

No processamento redundante, todas as requisições são processadas por todas as réplicas de forma que seja garantido um tempo constante de resposta, ainda que na presença de falhas. O principal protocolo desse grupo é a replicação ativa.

Na replicação ativa, todos os gerenciadores de réplicas agem como máquinas de estados que desempenham as mesmas atividades e que estão organizadas em grupos. Uma máquina de estados é formada por variáveis que encapsulam as informações de seu estado e de comandos que modificam esse estado ou produzem uma resposta [Schneider 1990]. Nesse modelo, cada comando consiste em um programa determinístico cuja execução é atômica em relação a outros comandos, de modo que a execução de uma máquina é equivalente a efetuar operações em uma ordem estrita. Dessa forma, o estado de uma máquina é uma função determinística de seus estados iniciais e da sequência de comandos neles aplicados.

A utilização desse modelo como protocolo de replicação só é possível se cada réplica começar no mesmo estado inicial e executar a mesma série de comandos, na mesma ordem, de forma que cada máquina produzirá os mesmos resultados para entradas iguais. Assim, a replicação ativa requer que o processamento de mensagens seja determinístico, o que impede, por exemplo, a utilização desse mecanismo em aplicações *multithreading*.

Na replicação ativa, todas as réplicas podem ser diretamente invocadas pelos usuários, portanto, as mensagens devem ser ordenadas em relação a todas as réplicas, à medida em que são recebidas. A ordenação total garante essas características.

Com a crescente utilização de WS em sistemas que demandam alta confiabilidade, a aplicação das técnicas de tolerância a falhas nesses componentes é cada vez mais comum. O presente trabalho visa incorporar no Axis2 um mecanismo de replicação passiva e híbrida transparente para serviços *web stateful*.

2.3. Processamento Híbrido

Além dos esquemas vistos, é possível construir novas abordagens que combinam elementos de processamento moderado com elementos de processamento redundante. O modelo de replicação aplicado em [Froihofer et al. 2007] é um exemplo deste aspecto.

3. Trabalhos relacionados

O requisito de disponibilidade, um dos principais componentes da confiabilidade de um sistema, é particularmente difícil de atingir no ambiente heterogêneo em que serviços *web* habitualmente operam. A tarefa de construir modelos de falha, por exemplo, é dificultada pela natureza da publicação dos serviços, já que a descrição de um serviço, obrigatoriamente, especifica apenas informações básicas necessárias à sua invocação, de modo que informações sobre aspectos não-funcionais, tais como disponibilidade e desempenho, não são publicadas. Dessa forma, determinar quais as possíveis falhas que um determinado serviço pode vir a apresentar pode ser impossível sem as informações necessárias sobre cada serviço que o compõe. De fato, a disponibilidade de um serviço pode ser até menor que qualquer um dos componentes que lhe dão forma [Moser et al. 2007].

Apesar da evidente necessidade, nenhuma especificação de confiabilidade no que tange à disponibilidade de serviços ainda foi desenvolvida. Isso ocorre, em grande parte, devido à dificuldade de estender as técnicas de replicação além das fronteiras de uma única organização, já que as tecnologias que elas usam no desenvolvimento de seus serviços são, possivelmente, não equivalentes. Nos últimos anos, vários *middleware* de replicação em serviços *web* foram desenvolvidos com o propósito de contornar essa limitação, dentre eles FT-SOAP [Chen 2007], WS-Replication [Jiménez-Peris et al. 2006], Conectores de Tolerância a Falhas [Fabre and Salatge 2007] e a replicação híbrida adotada em [Froihofer et al. 2007].

3.1. FT-SOAP

FT-SOAP [Chen 2007] é um mecanismo de replicação passiva. Os componentes básicos dessa especificação são: o **Gerenciamento de Falhas** que é utilizado para realizar o monitoramento de estado das réplicas, o **Mecanismo de Log e Recuperação**, responsável por fazer o *log* das invocações, de forma que elas não sejam perdidas na falha do gerenciador primário, e o **Gerenciador de Replicação**, que tem a função de monitorar e constituir os grupos de réplicas.

O uso de componentes centralizados adiciona novos pontos de falhas ao sistema, de forma que até os componentes do próprio *middleware* devem ser replicados.

No mecanismo proposto neste artigo, cada réplica funciona como um mecanismo de replicação independente, evitando a centralização encontrada na proposta FT-SOAP.

3.2. WS-Replication

WS-Replication [Jiménez-Peris et al. 2006] implementa replicação ativa no contexto dos serviços *web*. O protocolo de replicação aplicado é baseado no componente WS-Multicast que utiliza tecnologias básicas de WS (SOAP e WSDL) para promover sincronia entre as réplicas. Ao utilizar o WS-Multicast em sua estrutura, WS-Replication mantém na estrutura de replicação a independência de plataformas inerente à arquitetura SOA, escalando o protocolo a operar diretamente sobre os serviços na Internet. Além disso, como nenhuma modificação é imposta à implementação dos serviços, a replicação é totalmente transparente aos usuários.

O mecanismo de replicação implementado neste artigo, de maneira semelhante ao WS-Replication, é transparente aos usuários. No entanto, ao aplicar a replicação passiva, o mecanismo proposto requer menor processamento.

3.3. Conectores

Este mecanismo [Fabre and Salatge 2007] parte do princípio de que criar serviços *web* confiáveis é difícil porque eles geralmente dependem de outros serviços que não são confiáveis, de forma que é necessário encontrar meios externos ao serviço para prover mecanismos adicionais de tolerância a falhas. A idéia básica dessa infraestrutura é realizar a comunicação com os serviços *web* através de conectores que implementam características de replicação.

A noção de conector é baseada no conceito de ADLs (*Architecture Description Language*), que permite customizar interações entre componentes. Um conector específico de tolerância a falhas (*Specific Fault Tolerant Connector*) é utilizado para interceptar invocações a um WS e realizar ações características de mecanismos de replicação, funcionando como um componente intermediário entre um serviço *web* construído de forma não-confiável e um cliente que representa uma aplicação SOA crítica (*Critical SOA Based application*).

A presença de conectores em cada réplica de um serviço torna esse *middleware* bastante flexível, de maneira que é possível configurá-lo para aplicar replicação passiva ou ativa, além de possibilitar a utilização de réplicas não idênticas de uma mesma funcionalidade. Na replicação passiva, porém, para que haja manutenção de sincronia de estado entre as réplicas, os serviços devem implementar funções de manipulação de estado e publicá-las conjuntamente às funcionalidades propriamente ditas.

Diferentemente da proposta apresentada em [Fabre and Salatge 2007], no mecanismo desenvolvido e acoplado à plataforma Axis2, nenhuma alteração é imposta à codificação dos serviços replicados.

3.4. Modelo Híbrido de Replicação

Esse *middleware* [Froihofer et al. 2007] de replicação considera serviços publicados sobre infraestruturas homogêneas, de forma que é possível adotar medidas de replicação comumente utilizadas em objetos distribuídos. O modelo de replicação adotado tem o seu funcionamento básico baseado nos protocolos de replicação passiva e ativa.

Da passiva, ele herda a característica de todas as mensagens serem enviadas somente ao gerenciador primário. Da ativa, ele implementa o processamento redundante. O esquema de funcionamento pode ser observado na figura 1.

A mensagem chega à interface de transporte do gerenciador primário (passo 1), segue o fluxo de entrada até ser interceptada (passo 2). Nesse momento, a invocação é serializada e enviada a todas as réplicas através do *toolkit* SPREAD (passo 3). Ao receber a invocação, o interceptor a remonta (passo 4) e a envia para o início do fluxo de entrada (passo 5), de forma que a réplica processe a mesma invocação. Para garantir a sincronia, a réplica primária interrompe o fluxo de processamento (passo 2) e só o reinicia depois que a invocação é recebida novamente, uma vez que a réplica primária, necessariamente, também executa os passos 4 e 5.

O mecanismo de interceptação deste último *middleware* é implementado na *engine* do Axis, de forma que o protocolo de replicação não precisa ser implementado nos serviços em si. O mecanismo, porém, só é aplicado sobre serviços publicados através

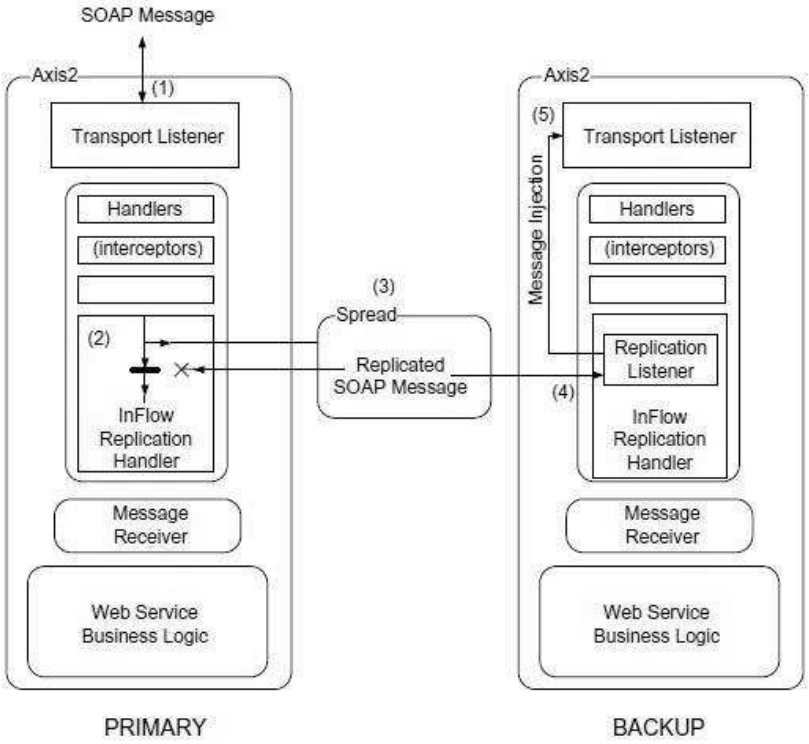

Figura 1. Arquitetura da Replicação Híbrida [Froihofer et al. 2007].

dessa ferramenta. Apesar desta limitação, a plataforma Axis é uma das principais formas de publicação de serviços *web* da atualidade, o que facilita a utilização deste mecanismo.

O mecanismo de interceptação e replicação deste *middleware* serviu de base para o desenvolvimento da proposta do presente trabalho. O principal diferencial foi a incorporação da replicação passiva clássica para serviços e, com isso, foi obtida uma diminuição no processamento de cada réplica. Na proposta implementada, porém, ao momento da interceptação da invocação e *multicast* da informação, não somente a invocação é enviada às demais réplicas, mas todo o contexto da invocação, de forma que ao receber esse contexto, as réplicas backup não necessitam re-inserir a invocação no início dos fluxos de entrada, mas somente dar prosseguimento ao processo do ponto de onde ele parou na réplica primária. A implementação desta replicação passiva é detalhada na próxima seção.

4. Implementação

O mecanismo proposto neste trabalho foi implementado para a versão mais nova do Axis, *engine* Axis2, com o intuito de garantir que nenhuma modificação no serviço *web* precisa ser realizada. Dois modelos de replicação (passiva e híbrida) foram implementados e testados para serviços *web stateful*, levando em consideração a manutenção de consistência de estado entre as réplicas. Os serviços testados foram desenvolvidos em Axis2 [Axis2 2010] e as garantias de comunicação em grupo entre as réplicas foram implementadas através da ferramenta (*toolkit*) JGroups [Ban 2008].

JGroups é um *toolkit* Java para comunicação *multicast* confiável. Sua arquitetura

consiste de três partes: uma API Canal (*Channel*) que provê as funcionalidades básicas de acesso e criação de grupos de réplicas; Blocos de Implementação (*Building Blocks*) que provêm uma abstração mais refinada de utilização do canal; e Pilha de Protocolos (*Protocol Stack*) que é formada pelos componentes que implementam as garantias de confiabilidade e ordenação da entrega de mensagens.

Já o Axis2 é um *engine* de implementação da especificação SOAP. Seu funcionamento consiste, basicamente, em enviar e receber mensagens XML. Para tanto, a arquitetura do Axis2 mantém dois fluxos que executam essas atividades. Esses fluxos são implementados pelo mecanismo Axis (*Axis Engine*) através de dois métodos: enviar (*send*) e receber (*receive*). Os dois fluxos são chamados, respectivamente, Fluxo de Entrada (*InFlow*) e Fluxo de Saída (*OutFlow*).

Nesse modelo, cada fluxo é dividido em fases (*phases*) e cada fase é formada de *handlers* que agem como interceptadores processando partes da mensagem e provendo qualidade de serviço. Quando uma mensagem SOAP está sendo processada, os *handlers* registrados nas fases são executados. As informações de execução desses *handlers* são armazenadas em objetos de contexto (*context*) que têm por finalidade manter dados que podem ser compartilhados entre várias invocações ou entre *handlers* de uma única invocação como, por exemplo, a sessão. Dessa forma, para adicionar funcionalidades ao processamento, é necessário registrar novos *handlers* nas fases de execução pré-existentes na arquitetura, ou em novas fases criadas pelo usuário.

Dois modelos de replicação foram implementados e testados: replicação passiva clássica e um modelo híbrido. O esquema de interceptação foi baseado no trabalho desenvolvido em [Froihofer et al. 2007], porém foi modificado a fim de melhorar a distribuição de mensagens no grupo.

4.1. Replicação Híbrida

Uma nova fase foi adicionada aos fluxos de entrada e saída intitulada *FaseReplicacaoHibrida*. O funcionamento do protótipo, no fluxo de entrada, pode ser observado na figura 2.

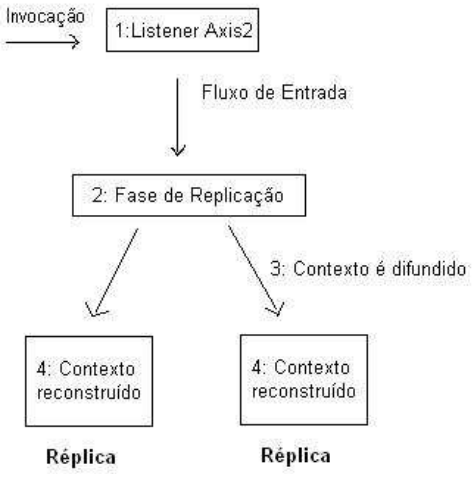

Figura 2. Funcionamento da Replicação Híbrida.

1. A invocação do cliente, interceptada pelo *listener*, chega à interface de transporte da réplica primária e é encaminhada às fases posteriores do fluxo de entrada. Durante as fases, cada *handler* recebe um objeto do tipo contexto e adiciona ou modifica informações no mesmo. Basicamente, um contexto contém informações sobre a configuração da engine Axis2, da sessão e o envelope SOAP em si;
2. Ao atingir a fase de replicação, o contexto é serializado e não somente a invocação, o que difere da solução proposta em [Froihofer et al. 2007]. Essa modificação é feita porque a fase de replicação foi inserida imediatamente antes da fase de processamento, de forma que o contexto ao qual o *handler* implementado tem acesso está completo e pronto para dar início ao processamento da requisição, tornando desnecessária a reconstrução do mesmo em cada réplica.
3. Serializado o contexto, a mensagem de atualização é enviada a todas as réplicas, exceto a réplica primária, que já possui a informação construída. Essa aproximação é possível, porque o uso do JGroups possibilita à réplica primária esperar por nenhuma, a primeira, a maioria ou todas as confirmações de recebimento das réplicas, de forma que a consistência de estados é garantida pela comunicação em grupo, dispensando o envio do contexto serializado para a mesma réplica que o gerou. O envio *multicast* é feito utilizando o bloco de implementação *RPCDispatcher* que permite a invocação remota de métodos.
4. No recebimento do contexto, as réplicas executam o algoritmo 1.

Algorithm 1 Recebimento do contexto pelas réplicas *backup*

```
1: context = unserialize(serializedContext);
2: context.activate;
3: AxisEngine.resume(context)
```

Ao ser serializado, o contexto é transformado em uma cadeia de caracteres codificados a fim de garantir a segurança da informação no ínterim em que trafega pela rede. Com o contexto reconstruído (linha 1), faz-se necessário reativá-lo para que o processamento seja reiniciado. No mecanismo implementado, o processamento da requisição é retomado de uma maneira otimizada, pois, não é necessária a reinserção do invocação no início do fluxo como faz a aproximação adotada em [Froihofer et al. 2007], de forma que a operação **AxisEngine.resume** dispara, imediatamente, a fase de processamento na réplica em questão, como pode ser observado na linha 3. Dessa forma, o processamento redundante é atingido.

Caso o gerenciador primário falhe, o novo líder é determinado através da nova lista de membros criada pelo JGroups. Para tanto, todas as réplicas têm acesso à listas de membros (*views*) idênticas, de forma que o novo líder escolhido é aquele que ocupa a primeira posição (posição zero) na *view*. Como cada réplica atualiza o seu estado ao recebimento de cada requisição, o estado é mantido e, dessa forma, o novo líder eleito está consistente para tratar as novas requisições.

Assim, o protótipo desenvolvido mantém o estado entre as réplicas, já que todas as requisições são processadas por todos os membros do grupo de replicação, apesar de somente o membro primário receber e responder às invocações dos clientes. A consistência de estado é garantida através da utilização do protocolo FIFO na pilha do JGroups.

4.2. Replicação Passiva Clássica

Semelhante ao protótipo de replicação híbrida, também foi adicionada uma nova fase ao fluxo de entrada (*FaseReplicacaoPassiva*). As etapas pelas quais a replicação passiva realiza suas atividades são bastante próximas da replicação híbrida, exceto que as réplicas agem de forma distinta ao recebimento do contexto serializado. Os passos que a replicação passiva aplica no fluxo de entrada são os seguintes:

1. Antes da fase de processamento, o contexto é serializado e enviado às réplicas de forma semelhante aos passos 2 e 3 do fluxo de entrada no esquema híbrido. Como o processamento é moderado, nenhum mecanismo de registro de espera por confirmação de execução é necessário. Além do contexto em si, o nome do serviço que está sendo invocado também é enviado;
2. No recebimento do contexto, as réplicas executam os passos enumerados no algoritmo abaixo:

Algorithm 2 Recebimento do contexto pelas réplicas *backup*

```
1: serviceName = context.getServiceName
2: contextState = contexto.getSerializedValue;
3: historic.save(serviceName, contextState);
```

A mensagem de atualização para a réplica contém dois campos, sendo o primeiro deles o nome do serviço e o segundo o contexto serializado. Na linha 3, o contexto serializado é armazenado em um *hashmap* mantido em cada réplica. É possível notar que, a cada requisição a um determinado serviço, seu histórico é sobrescrito com o novo contexto. Isto é pertinente porque as informações de sessão são mantidas no contexto do Axis2, de forma que o contexto mantém as modificações feitas aos estados da sessão. Para suportar persistência de estado em dispositivos secundários como arquivos ou bancos de dados, o histórico pode ser facilmente estendido de forma a armazenar uma lista de contextos em cada serviço. Cada contexto corresponderia a uma invocação e assim, na recuperação de estado, todas as entradas do histórico teriam que ser reconstruídas e processadas na nova réplica primária antes que a nova requisição fosse processada.

Dessa forma, em cada réplica, antes da execução do fluxo acima, o histórico é verificado à procura de entradas pelo nome do serviço. Como a réplica primária nunca recebe as mensagens de atualização que envia, necessariamente, o seu histórico sempre estará vazio. Na ocorrência de falhas, quando o novo primário é eleito e recebe uma invocação, ele busca no histórico algum contexto associado ao nome do serviço. Caso encontre, esse contexto é reconstruído e processado antes que a nova requisição do usuário seja atendida.

Por utilizar o processamento moderado, o tempo de recuperação de estado é proporcional ao número de requisições que devem ser processadas antes da execução da nova invocação. Se for considerado o contexto do Axis2, então esse tempo é fixo e igual a uma requisição a mais. Caso o histórico seja modificado a fim de manter todo o conjunto de requisições, então esse valor pode ser bem maior dependendo do intervalo de tempo decorrido entre a falha do servidor primário e o início do recebimento das invocações.

A ordenação FIFO de mensagens é suficiente para manter a consistência de estado entre as réplicas. O estado é mantido através do *hashing* de contextos replicado

em cada réplica. Na falha do líder, o novo gerenciador primário irá recuperar o estado reaplicando as requisições feitas a cada serviço, isoladamente, de forma que o mínimo de processamento seja feito na fase de recuperação. Dessa forma, cada membro do grupo de replicação funciona como um mecanismo de replicação em si, dispensando a criação de componentes centralizados e a conseqüente adição de novos pontos de falha ao modelo.

5. Experimentos e resultados

Com o intuito de avaliar o funcionamento dos protótipos desenvolvidos foram realizados testes de desempenho para determinar o *overhead* da utilização do mecanismo no que tange ao trabalho implementado e a utilização da comunicação em grupo. Também foram validadas a manutenção e consistência de estado entre as réplicas.

5.1. Configuração do Ambiente

Os experimentos iniciais foram realizados de forma a considerar somente o tempo de execução dos protótipos sem influência da latência de redes. Para tanto, foram criadas quatro instâncias independentes do servidor Tomcat, versão 6.0.18, em um PC Pentium D 2.80 GHz, 1.5 GB de RAM, *Microsoft Windows XP Professional* 2002 *Service Pack 2*. Cada instância Tomcat publica uma instância Axis2.1.4.1. Já a versão do JGroups toolkit é a 2.6.4 e a pilha padrão de protocolos foi utilizada.

Para a realização dos testes foi desenvolvido um serviço *web* simples, que mantém um estado (*stateful*). As operações básicas oferecidas pelo serviço estão listadas a seguir:

1. Criar sessão: Método responsável por criar um vetor na sessão;
2. Adicionar elemento: Adiciona um elemento ao vetor na sessão;
3. Listar número de elementos: Retorna a quantidade de itens no vetor;
4. Listar elementos: Retorna os elementos na sessão em uma string.

Nos experimentos, cada requisição adicionou um elemento ao vetor da forma ”elemento i”, onde i cresce de acordo com o número da requisição, variando, dessa forma, de ”elemento 0”a ”elemento 2999”.

Esse serviço foi publicado em cada instância Axis2 instalada no computador. Para o teste de desempenho, foi feita uma série de 3000 requisições à réplica primária e calculados a média e o desvio padrão para os seguintes cenários:

1. Um serviço, sem qualquer adição de replicação;
2. Replicação com 1, 2, 3 e 4 réplicas.

5.2. Conjunto de testes - Número de réplicas

Para avaliar o overhead dos mecanismos em si, foram aplicados testes nos protótipos na presença de uma única réplica, de forma a evitar o overhead da comunicação em grupo. Os valores obtidos estão de acordo com o gráfico da figura 3.

Os testes foram executados 5 vezes a fim de determinar o intervalo de confiança do experimento. Para um nível de 95%, a média de execução para o cenário sem replicação, foi de 46,77 milissegundos com um desvio padrão de 0,11 millesegundos. A média para a replicação passiva foi de 50,49 milissegundos para o intervalo de 0,44 milissegundos, impondo um *overhead* de 7,9%. Já o valor da replicação híbrida, teve média 55,79 milissegundos para o intervalo de 0,27 aumentando o tempo médio em 19,2%.

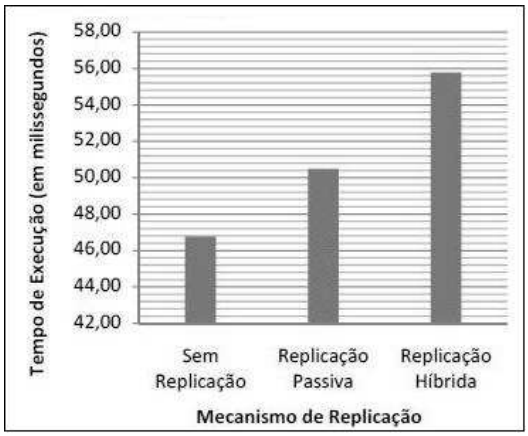

Figura 3. Gráfico comparativo dos protótipos de replicação.

O aumento no tempo de execução imposto pelo mecanismo é justificável porque a adição de uma fase representa um aumento na execução geral de mensagens, pois cada fase adicionada ao fluxo representa mais um passo de execução na *engine* do Axis2. Outro aspecto importante é a serialização do contexto. À medida que as invocações são feitas e mais informações são adicionadas à sessão, o contexto aumenta de tamanho de forma que a sua serialização representa um dos principais aspectos de consumo de tempo no modelo proposto e explica o desvio padrão relativamente alto, já que as últimas requisições da série demandaram um maior esforço de serialização em relação às requisições iniciais.

Testes foram realizados na presença de mais de uma réplica para que o overhead da comunicação em grupo pudesse ser determinado. Assim, com a adição de réplicas, o mecanismo de comunicação em grupo é efetivamente utilizado, de forma que o tempo de execução de cada mensagem, além dos fatores considerados anteriormente, é acrescido pelo envio das mensagens de atualização de estado para cada réplica, além do mecanismo de monitoração de estado dos membros do grupo, que cresce conjuntamente ao número de cópias. Nesse modelo, o *overhead* máximo foi atingido na presença de 4 réplicas, ou seja, 1 réplica primária e 3 réplicas backups. Para esse cenário com 4 cópias, o tempo médio foi de $68{,}70 \pm 5{,}91$ milissegundos, o que implica em um aumento de 14,5% em relação à replicação passiva com o mesmo número de réplicas e 46,9% em relação ao modelo sem replicação.

Por último, foram realizados testes em uma rede LAN, de forma que um teste mais próximo de um ambiente real fosse mensurado. O resultado dos testes pode ser observado na figura 4.

A média para a replicação passiva com duas réplicas foi 267,09 milissegundos, enquanto que o resultado para três réplicas foi 274,14 milissegundos , o que representa um aumento de 2,64% entre as configurações. Já a replicação híbrida teve média de 270,56 milissegundos para duas réplicas e 280,26 milissegundos para três. A diferença, nesse contexto, foi de 3,59%,demonstrando que o tempo de execução dos protótipos não aumenta bruscamente, em rede, com a adição de réplicas.

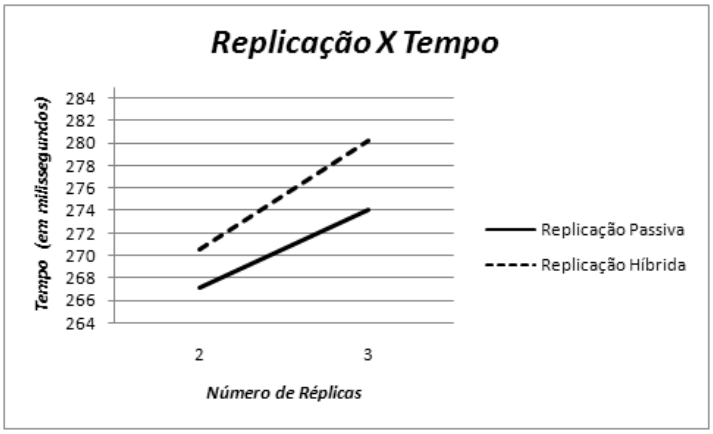

Figura 4. Resultado dos testes em rede.

5.3. Conjunto de Testes - Manutenção do Estado e Consistência

Para o teste de manutenção de estado, a análise foi feita fazendo a réplica primária falhar continuamente e aplicando a operação de listar o número de itens no novo primário eleito. Para o teste de consistência, a listagem de elementos foi invocada em cada novo primário e armazenada para posterior comparação.

O teste de manutenção de estado funcionou dentro do esperado, de forma que cada réplica apresentou 3000 itens em seus vetores após a falha do servidor primário.

O teste de consistência também foi bem sucedido. Para comparar os estados, após as invocações, a operação de listagem de conteúdo foi invocada em cada réplica e seus resultados armazenados para que pudessem ser comparados através da API Java.

Os testes demonstraram que o maior *overhead* da replicação foi a comunicação em grupo. A pilha de protocolo utilizada nos testes foi a pilha padrão do JGroups, de forma que é necessário estudá-la melhor e customizá-la a fim de apresentar um desempenho mais satisfatório.

Como todas as modificações foram feitas na arquitetura Axis2, nenhuma mudança foi necessária diretamente no serviço *web*, de forma que, nesse aspecto, este trabalho garante que serviços *web* autônomos e heterogêneos sejam replicados de uma maneira transparente.

6. Conclusão e Trabalhos Futuros

Este trabalho propôs analisar a maneira pela qual os métodos de replicação são adotados em conjunto aos serviços web. Os protótipos implementados demonstraram que a replicação em serviços é praticável, inclusive no que se refere à manutenção e consistência de estado entre as réplicas. A principal fonte de latência encontrada foi a comunicação em grupo, demonstrando que a primitiva *multicast* e o controle de membros de grupo podem ser bastante dispendiosos.

Dentre os trabalhos futuros, destaca-se uma melhor análise da pilha de protocolos do JGroups para obter um melhor desempenho, além de analisar a escalabilidade da solução proposta. Outrossim é a implementação de outros protocolos de replicação, tais como replicação semi-passiva e ativa.

Referências

Avizienis, A., Landwher, C., and Laprie, J.-C. (2004). Basic concepts and taxonomy of dependable and secure computing. IEEE Transactions on Dependable and Secure Computing.

Axis2 (2010). Apache axis2. Disponível em: http://ws.apache.org/axis2/. Último Acesso em 31 de março de 2010.

Ban, B. (2008). Reliable multicasting with the jgroups toolkit. [S.l].

Chen, Chyouhaw; Fang, C.-L. L. D. (2007). Ft-soap: A fault-tolerant web service. Journal of Systems Architecture: the EUROMICRO Journal.

Cristian, F. (1991). Understanding fault tolerant distributed systems. Communication of ACM.

Défago, X. and Schiper, A. (2001). Specification of replication techniques, semi-passive replication, and lazy consensus. Academic Press.

Fabre, J.-C. and Salatge, N. (2007). Fault tolerance connectors for unreliable web services. 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks.

Froihofer, L., Goeschka, K., Osrael, J., and Weghofer, M. (2007). Axis2-based replication middleware for web services. IEEE International Conference on Web Services (ICWS 2007).

Iwasa, K., Durand, J., Rutt, T., Peel, M., Kunisetty, S., and Bunting, D. (2004). Web services reliable messaging tc ws-reliability 1.1. OASIS Open 2003-2004.

Jiménez-Peris, R., Patino-Martinez, M., Pérez-Sorrosal, F., and Salas, J. (2006). Ws-replication: A framework for highly available web services. WWW 2006 - International World Wide Web Conference, Edinburgh,Scotland.

Lawrence, K. and Kaler, C. (2004). Web services security: Soap message security 1.1. http://docs.oasis-open.org/wss/v1.1/wss-v1.1-spec-pr-SOAPMessageSecurity-01.htm.Último acesso em: 03 de março de 2009.

Moser, L. E., Smith, P., and Zhao, W. (2007). Building dependable and secure web services. JOURNAL OF SOFTWARE, VOL. 2, NO. 1.

Schneider, F. B. (1990). Replication management using the state-machine approach. ACM Press/Addison-Wesley Publishing Co.