Auto-adaptação de requisitos de tolerância a faltas através de contratos

André Luiz Barbosa Rodrigues[1], Alexandre Sztajnberg[1], Orlando Loques[2]

[1]DICC/IME e PEL/FEN - Universidade do Estado do Rio de Janeiro
[2]Instituto de Computação - Universidade Federal Fluminense

rblandre@ime.uerj.br, alexszt@uerj.br, loques@ic.uff.br

Abstract. *Fault tolerance is a constant concern in data centers, where servers have to run with a minimum quality level. Variations on the operating conditions, server demands and in system failure rate have to be handled in such a way that SLAs are honoured and services not interrupted. We present an approach to handle fault tolerance requirements guided by contracts, based on component replication, which describe adaptation policies for each application. At run-time a software infrastructure autonomically manages the deployment, monitoring and maintenance of the fault tolerance requirements described in the contract, reconfiguring the application, when necessary, to maintain compliance. An example with an Apache web server and replicated Tomcat servers is used to validate the approach.*

Resumo. *Tolerância a faltas é uma preocupação constante em "data centers", onde servidores devem executar com um mínimo de qualidade. Variações nas condições de operação, da demanda e na taxa de faltas devem ser mitigadas, de tal forma que SLAs sejam cumpridos e os serviços não sejam interrompidos. Apresentamos uma abordagem para tratar requisitos de tolerância a faltas, baseada em replicação de componentes, que utiliza contratos para descrever as políticas de adaptação requeridas em cada aplicação. Em tempo de execução uma infra-estrutura de software gerencia de forma autonômica a implantação, o monitoramento e a manutenção dos requisitos descritos no contrato, reconfigurando a aplicação quando necessário para manter a aderência. Um exemplo com um servidor Apache e servidores Tomcat replicados é discutido para validar a abordagem.*

1. Introdução

Sistemas distribuídos são sujeitos a faltas decorrentes de problemas na infra-estrutura de hardware e software sobre os quais são executados. Estas faltas podem provocar falhas nestes sistemas levando à indisponibilidade dos serviços providos. O uso de técnicas de tolerância a faltas permite a recuperação destes sistemas, ainda que parcialmente, levando à continuidade dos serviços [Jalote 1994]. Tolerância a faltas é obtida usualmente através da redundância de elementos de software e hardware, na maioria das vezes alocados de forma estática, que podem ficar grande parte do tempo sub-utilizados. Assim sendo, o dimensionamento adequado de elementos redundantes é importante para controlar os custos finais dos sistemas.

Algumas abordagens para introduzir tolerância a faltas utilizam soluções *ad hoc*, misturando o código responsável por atender os requisitos funcionais com o código responsável pela redundância, através de replicação de componentes, e a manutenção da

consistência das réplicas. O resultado é um código com alto grau de acoplamento, o que prejudica o reuso. Outras abordagens eliminam este problema utilizando mecanismos embutidos na infra-estrutura de suporte sobre a qual as mesmas são desenvolvidas, separando requisitos funcionais dos requisitos de tolerância a faltas. Ainda assim, a configuração destes mecanismos é definida de forma estática, forçando a pré-alocação de recursos. Exemplos que utilizam esta abordagem são o JEE e .NET.

Abordagens adaptativas para o suporte de requisitos de tolerância a faltas procuram alcançar um equilíbrio entre a robustez e a utilização eficiente dos recursos. Por um lado, técnicas de replicação e consistência diferenciadas podem ser empregadas em contextos de operação específicos, consumindo, obviamente, os recursos necessários (tempo de processamento, banda de rede, etc.). Por outro, a alocação de recursos também pode ser "verde" na medida em que os recursos redundantes passam a ser utilizados apenas quando os mesmos são realmente necessários para atender a uma dada política de tolerância a faltas de forma aceitável.

A aplicação de técnicas de tolerância a faltas adaptativas tem se tornado atraente em instalações do tipo *data centers*, onde servidores devem executar sem interrupção visível dos serviços providos e com determinado nível de qualidade. Falhas e variação de carga devem ser mitigadas, de tal forma que SLAs (*Service Level Agreement*) sejam cumpridos. Por exemplo, técnicas mais robustas, como a replicação *ativa* ou *ativa cíclica*, podem ser acionadas quando há um cenário de operação mais propenso a falhas e, no outro sentido, técnicas mais simples, como a replicação *passiva*, podem ser novamente acionadas quando o cenário de operação voltar a ser mais tranqüilo. Neste contexto, é desejável que os requisitos de tolerância a faltas e as políticas de adaptação possam ser descritas em nível alto de abstração. Também é desejável que estes requisitos possam ser implantados com separação de interesses em relação às aplicações, e gerenciados em tempo de execução de forma autonômica..

Neste artigo propomos o uso de contratos arquiteturais para especificar requisitos de tolerância a faltas, em que perfis quantificam propriedades como o *tipo de replicação*, o *número de réplicas* e o *intervalo de checkpointing* desejado; adicionalmente uma máquina de negociação especifica níveis de qualidade desejados e como estes são impostos. Em tempo de execução uma infra-estrutura de software permite a implantação, o monitoramento e a manutenção dos requisitos descritos no contrato de forma autonômica, reconfigurando dinamicamente a aplicação e os recursos responsáveis pela tolerância a faltas para manter os requisitos contratados.

Para validar a abordagem, empregamos os contratos descritos em CBabel e a infra-estrutura de suporte de CR-RIO para prover tolerância a faltas em um exemplo envolvendo o servidor HTTP Apache [Apache 2007a], integrado a um grupo de servidores Tomcat [Apache 2007b]. A idéia da aplicação-exemplo é acionar dinamicamente uma técnica de replicação adequada para cada contexto de operação específico, considerando o tempo de resposta do conjunto de réplicas e a taxa de faltas associada.

A Seção 2 apresenta os elementos de CR-RIO. Na Seção 3 apresentamos a integração dos elementos de suporte a tolerância a falhas em CR-RIO e a especificação de requisitos de replicação através de contratos. A Seção 4 apresenta, então, a aplicação Apache-Tomcat e discute alguns pontos de sua implementação. Na Seção 5 são citados trabalhos relacionados. Na Seção 6 são apresentados as conclusões e trabalhos futuros.

2. *Framework* para a Implantação de Contratos

O *framework* CR-RIO (*Contractual Reflective-Reconfigurable Interconnectable Objects*) é centrado em um modelo arquitetural e em uma linguagem de descrição, CBabel, para descrever a arquitetura funcional de aplicações e expressar seus requisitos não-funcionais por meio de contratos [Loques 2004]. Com base nestes elementos, desenvolveu-se uma infra-estrutura de suporte para: (i) interpretar a especificação dos contratos e armazená-la como meta-informação, associada à aplicação; (ii) prover mecanismos de reflexão e adaptação dinâmica, que permitem adaptar a configuração da aplicação (incluindo os seus elementos de suporte), visando atender as exigências de contratos; e (iii) prover elementos para impor, monitorar e manter os mesmos.

2.1 Contratos

A configuração funcional de uma aplicação é definida através da especificação de componentes arquiteturais, que realizam atividades essenciais à mesma, não permitindo negociação. Requisitos não-funcionais de uma aplicação são definidos através de restrições operacionais ou de qualidade e podem admitir alguma negociação envolvendo os recursos utilizados. Um contrato descreve em tempo de projeto os aspectos não-funcionais da aplicação, especificando o uso a ser feito de recursos compartilhados durante a operação, e variações aceitáveis na disponibilidade destes recursos:

Categorias. Descrevem propriedades e aspectos não-funcionais específicos de componentes, recursos ou serviços. Por exemplo, características do processador, memória e comunicação. Aspectos menos tangíveis como faixa de preço ("caro", "barato"), tolerância a faltas, ou qualidade de um componente ou serviço ("boa", "média", "ruim") também podem ser descritos. Cada Categoria é associada a componentes ou serviços de suporte, que irão alocar e monitorar os respectivos recursos, podendo utilizar para isso a infra-estrutura disponível.

Perfis. Quantificam ou valoram as propriedades de uma Categoria. A quantificação restringe cada propriedade, funcionando como uma instância de valores aceitáveis para determinada Categoria. Perfis podem ser definidos, com a granulosidade desejada, para indicar o nível de qualidade aceitável no contexto de operação de componentes individuais ou partes de arquitetura.

Serviços ou **configurações** arquiteturais. Especificam versões da arquitetura que vão definir os possíveis níveis de qualidade ou estados de operação da aplicação. Cada configuração contém uma descrição de componentes arquiteturais e suas interações, associados a um ou mais perfis, especializando a arquitetura básica. Assim, o nível de qualidade desejado/tolerado por um serviço ou configuração é diferenciado de outro pelo conjunto das propriedades declaradas nos perfis. Um serviço só pode ser implantado ou mantido se todos os perfis associados ao mesmo forem válidos.

Cláusula de negociação. Descreve uma política, definida por uma máquina de estados, que estabelece uma ordem arbitrária para a implantação das configurações. De acordo com o descrito na cláusula, quando uma configuração de maior preferência (com alta qualidade, por exemplo) não puder mais ser mantida, a gerência de contratos tentará implantar uma configuração de menor preferência (com menor qualidade ou que exija menos recursos). O retorno para uma configuração de maior preferência também pode ser descrito, permitindo que uma configuração de melhor qualidade seja (re)implantada, se os recursos necessários à mesma tornarem-se (novamente) disponíveis.

As arquiteturas descritas em CBabel são mapeadas em um modelo de objetos [Corradi 2005]. Este modelo é refletido em um repositório de meta-nível, que pode ser atualizado durante uma (re) configuração e consultado durante a vida da aplicação. Este repositório inclui também a representação dos contratos e mantém informações relacionadas aos recursos de interesse das aplicações. A semântica definida por um contrato é imposta em tempo de operação por uma infra-estrutura de suporte composta por um conjunto de componentes formando padrões reusáveis [Lisbôa 2004].

2.2 Infra-estrutura de Suporte

A infra-estrutura de suporte (Figura 1) é constituída de elementos com papéis bem definidos na implantação e reconfiguração da arquitetura das aplicações e na gerência autonômica dos contratos.

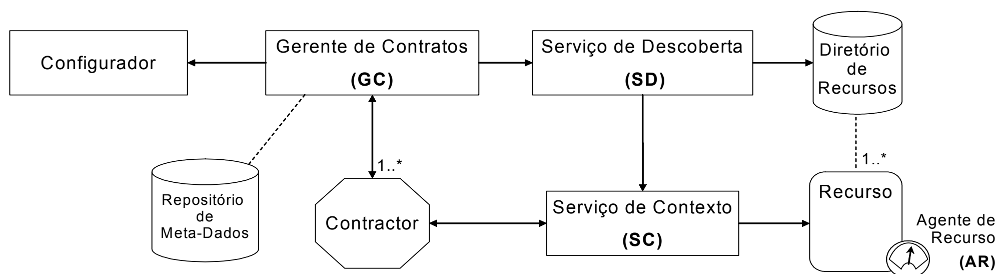

Figura 1. Infra-estrutura CR-RIO

Configurador. Elemento responsável por mapear as descrições arquiteturais (em CBabel) em ações que efetivam as configurações requeridas nos sistemas nativos. O Configurador provê duas APIs: configuração e reflexão arquitetural, através das quais as facilidades de configuração são acessadas. A API de configuração permite instanciar, ligar, parar e substituir componentes para implantar e reconfigurar a aplicação. Estas operações são atômicas e refletidas no repositório persistente de meta-nível, que pode ser consultado através da API de reflexão arquitetural.

Gerente de Contratos (GC). Responsável pela implantação dos serviços diferenciados e a pela gerência das políticas descritas no contrato. Para implantar uma das configurações do contrato, o GC (i) envia o conjunto de perfis de todas as configurações aos *Contractors*, (ii) solicita a verificação das restrições exigidas pelos perfis associados à configuração a ser implantada e (iii) aguarda a notificação dos mesmos. Se todos os *Contractors* responderem positivamente, a configuração pode ser implantada. Caso contrário, uma outra configuração deve ser selecionada, de acordo com a cláusula de negociação, e o procedimento de implantação é, então, reiniciado. Se nenhuma das configurações descritas no contrato puder ser implantada, a aplicação é terminada. O mesmo ocorre se durante a operação de uma dada configuração o CG receber uma notificação de perfil inválido. Para efetivar a implantação dos componentes arquiteturais da configuração selecionada (recém negociada com os *Contractors*) o CG utiliza o Configurador. O GC pode também iniciar uma nova negociação, quando os recursos para a implantação de uma configuração de maior preferência tornam-se disponíveis (os respectivos perfis são válidos), mesmo se a configuração corrente mantiver-se válida.

Contractor. Coordena o processo de alocação e monitoração das propriedades de componentes básicos (mecanismos, recursos ou serviços). Uma aplicação distribuída necessita de um *Contractor* em cada nó do domínio, que recebem do GC o conjunto de

todos os perfis. Periodicamente cada *Contractor* solicita ao S. de Contexto os valores monitorados de propriedades de interesse e compara os mesmos com as restrições descritas nos perfis. O *Contractor* notifica o GC que a configuração atual não é mais válida se, pelo menos, um dos perfis foi violado. Notifica também a lista de todos os perfis que estão válidos, incluindo os relacionados à configuração atual.

Agente de Recursos (AR). Encapsula o acesso específico aos elementos básicos de suporte (recursos, serviços, componentes, etc.), provendo interfaces para gerência e monitoração de valores de propriedades requeridas. Os ARs são especializados e podem acessar serviços primitivos. Os valores monitorados são enviados para o S. de Contexto.

Serviço de Contexto (SC). O SC é responsável por disponibilizar informações de contexto e esconder os detalhes de baixo nível usados na comunicação com os (vários) ARs. Desta forma, a aplicação preocupa-se somente com os dados necessários e não como eles são obtidos. Ao receber uma consulta, o SC verifica quais as propriedades de contexto requisitadas de cada um dos recursos. Em seguida, o SC (e não a aplicação) comunica-se com cada AR envolvido para obter as informações de contexto individuais. Após coletar as informações, o serviço consolida as mesmas e repassa o resultado ao solicitante.

Serviço de Descoberta (SD). O SD é consultado quando uma aplicação não conhece de antemão o recurso ou componente a ser usado. Para isso a classe de recurso desejado é informada, bem como as restrições de contexto que este deve satisfazer. Por exemplo, a aplicação precisa acessar um servidor Web que tenha um tempo de resposta médio inferior a 1seg. Neste caso, o SD deve consultar o SC obtendo os valores das propriedades de contexto de elementos pré-selecionados. A resposta do SD é uma lista de referências de todas as instâncias de recursos da classe requerida pela aplicação (dentro do domínio de atuação do SD) que atendam as restrições de contexto impostas.

Observamos que a semântica dos contratos aciona de forma consistente os elementos da infra-estrutura. Por exemplo, se a arquitetura da aplicação especifica um determinado módulo *TomCat*, estaticamente (cuja referência é previamente conhecida), o GC simplesmente aciona o Configurador, para implantar e alocar o componente específico. Por outro lado, se o módulo é especificado através de uma atribuição dinâmica (cuja referência ainda não é conhecida), o GC aciona o SD para descobrir e selecionar um componente *TomCat* (possivelmente o melhor) antes de solicitar a alocação do mesmo ao Configurador.

3. Arquitetura, Categoria e Perfis do Serviço de Replicação

Consideramos a replicação, em nossa abordagem, um serviço de suporte que pode ser utilizado e referenciado em um contrato. Assim, este serviço deve seguir a arquitetura padrão descrita na Seção 2. Os elementos de suporte à replicação de componentes, comuns em várias propostas, como [Gorender 2005], [Lung 2006] foram integrados à infra-estrutura de suporte de CR-RIO: (a) um Gerente de Replicação (GR), (b) o grupo de réplicas e (c) Controladores de Réplica (CTL-R) para cada réplica individual.

As propriedades de replicação e faltas no nível arquitetural são descritas por Categorias. A categoria *Replication* (Listagem 1) define as propriedades do serviço de replicação, inspiradas em [Lung 2006], indicando o que pode ser requerido deste serviço, e também o que pode ser monitorado, independentemente da técnica utilizada. Propriedades: (i) o número de réplicas; (ii) o intervalo de *checkpoint* e acionamento do

protocolo de consistência (l: 3); (iii) o intervalo de monitoramento de cada réplica (l: 4), e (iv) o tempo limite para que cada réplica responda quando monitorada (l: 5). Com estas propriedades o GR pode identificar que o número de réplicas está fora da especificação. Por exemplo, o perfil *ActiveCP* indica que cada réplica deverá responder ao processo de monitoramento, a cada 20s, e em até 200ms (l: 8-9). Uma réplica será considerada indisponível se não responder dentro deste intervalo. O perfil *ActCNRep*, separado por questões de modularidade, indica que o grupo deve ter 4 réplicas.

```
1 category Replication {
2   numberOfReplicas: numeric;
3   checkPointInterval: numeric s;
4   monitoringInterval: numeric s;
5   timeoutInterval: numeric ms;
6 };

 7 profile{
 8   Replication.monitoringInterval = 20;
 9   Replication.timeoutInterval = 200;
10 } ActiveCP;

11 profile{
12   Replication.numberOfReplicas = 4;
13 } ActCNRepP;
```

Listagem 1. Categoria para replicação, perfil para replicação ativa cíclica

A categoria *Faults* (Listagem 2) é proposta para especificar propriedades relacionadas a faltas: (i) o número de faltas tolerado antes da configuração se tornar inválida (l: 2); o intervalo em que as faltas podem ocorrer (l: 3); e o período de tempo mínimo para que o grupo de réplicas seja considerado estável. A propriedade *stableInterval*, adequadamente utilizada em um contrato, permite que uma técnica menos robusta seja usada, no lugar de uma mais robusta, dado que o número de faltas está, por algum tempo, abaixo do especificado. Além disso, ela aplica certo retardo à decisão de controle, evitando instabilidades devido a condições transitórias. Por exemplo, o perfil *ActiveCFaults* especifica que são toleradas 2 faltas a cada 15seg, e o grupo é considerado estável se não apresentar faltas durante 60seg.

```
1 category Faults{
2    numberOfFaults: decreasing numeric;
3    faultInterval: decreasing numeric s;
4    stableInterval: increasing numeric s;
5 };

 6  profile{
 7    Faults.numberOfFaults=2;
 8    Faults.faultInterval=15;
 9    Faults.stableInterval=60;
10  } ActiveCFaults;
```

Listagem 2. Categoria para faltas e perfil de faltas para replicação ativa cíclica

Os perfis baseados nas categorias *Replication* e *Faults* serão usados em conjunto para especificar, em um contrato, o nível de tolerância a faltas requerido e, em tempo de execução, para avaliar se este nível está sendo respeitado. É necessário, então, incluir os componentes de software responsáveis efetivamente por gerenciar estas propriedades.

Em nossa solução (Figura 2) o papel de Gerente de Replicação (GR) é encapsulado em um *Contractor*. Baseado nos perfis de replicação e faltas, ele controla a qualidade do serviço, avaliando se as réplicas estão “vivas”, se o número de réplicas está dentro do especificado e se o número de faltas do grupo está dentro dos parâmetros. O GR realiza suas atividades independentemente da técnica de replicação. As interações entre um módulo cliente e os módulos replicados são realizadas por comunicação de grupo. Como este é um papel de interação, um conector[1] de grupo realiza a difusão

---

[1] Módulos representam componentes funcionais da aplicação e conectores a mediação ou interconexões entre os módulos (considerados de meta-nível). Cadeias de conectores podem ser interpostas na rota de interação entre módulos, permitindo filtrar, restringir ou mesmo distribuir as interações.

seletiva de mensagens para os membros, e um AR associado a este conector monitora a comunicação e a qualidade da difusão.

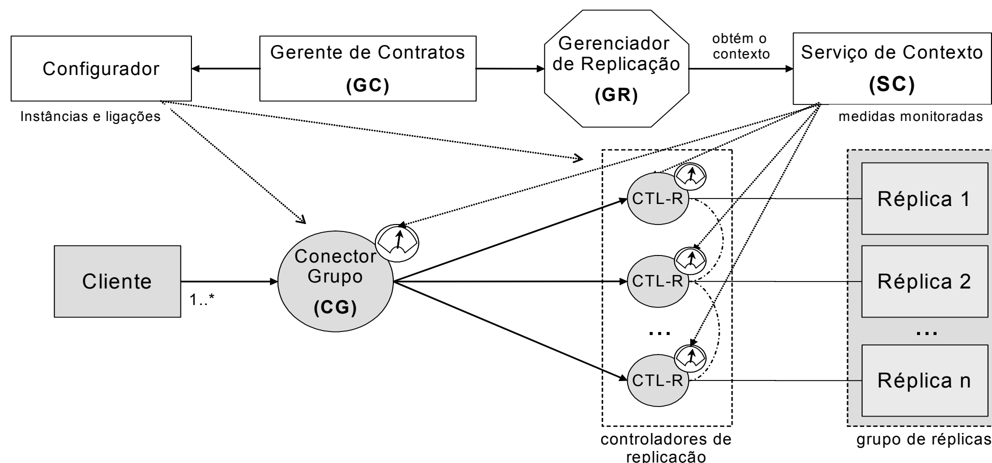

Figura 2. Estrutura do Serviço de Replicação

Cada módulo do conjunto de réplicas tem a sua interação com outros módulos da aplicação interceptada por um conector Controlador de Réplica (CTL-R). Este elemento, essencial em nossa abordagem, dá suporte às várias estratégias de manutenção da consistência das réplicas sem interferir diretamente nos módulos replicados. Cada técnica de replicação é associada a um CTL-R especializado. A seleção de um CTL-R específico determina a técnica de replicação a ser utilizada. O CTL-R recebe um perfil contendo as propriedades de monitoração (*interval* e *timeout*). Um AR associado realiza os testes pró-ativamente e registra os resultados do SC. Desta forma é possível ao GR verificar se cada réplica atende aos perfis de replicação. O CTL-R é autônomo para realizar procedimentos de eleição, quando necessário, e se comporta adequadamente quando o mesmo é eleito primário (respondendo as requisições, persistindo o estado e mantendo a consistência do grupo).

Uma vez selecionada uma técnica de replicação e implantada a configuração correspondente, o conjunto de CTL-R realiza os procedimentos para atingir a consistência pós-reconfiguração e os ARs passam a monitorar as propriedades de interesse, declaradas nos perfis. Periodicamente o GR consulta o SC, que apresenta as informações de contexto consolidadas a partir dos vários ARs. O GR verifica, então, se os intervalos de tempo e o número de réplicas se mantém dentro dos perfis da configuração atual e se os mesmos atendem aos perfis de outras configurações. Ao receber uma notificação do GR o GC decide o que deve ser feito. No caso de uma invalidação da configuração de replicação atual, uma outra versão da configuração da aplicação é selecionada segundo a política descrita na cláusula de negociação do contrato, por exemplo, com uma técnica de replicação mais robusta, e a rotina para sua implantação é iniciada.

No caso da notificação de configuração válida o GC pode decidir mudar a configuração atual. Por exemplo, se não ocorrerem faltas durante o intervalo especificado pela propriedade *stableInterval,* isso indica que o número de faltas baixou para patamares mais adequados para um serviço que possui menores exigências. Neste caso, o contrato pode especificar, por exemplo, que uma configuração com replicação passiva pode substituir a configuração atual, digamos, ativa cíclica. Caso contrário, nenhuma ação é disparada e a técnica de replicação atual é mantida.

4. Exemplo de Aplicação

Para validar a abordagem apresentada, vamos tomar como exemplo um cenário usualmente encontrado em *data centers*: um servidor HTTP Apache e um conjunto de servidores de aplicações Tomcat. O uso de um conjunto replicado de servidores pode ter o objetivo de melhorar a escalabilidade do serviço (ou diminuir o tempo de resposta) e tornar o sistema mais robusto, mitigando falhas através de redundância. Em nosso exemplo, a preocupação está relacionada aos requisitos não-funcionais de tempo de resposta médio do grupo de módulos *TomCat* e a tolerância a faltas:

(a) Sob carga de acesso normal, com tempo de resposta abaixo de 200ms, apenas 2 servidores Tomcat serão utilizados com a técnica de replicação passiva. O objetivo é diminuir o consumo de recursos enquanto as requisições forem tratadas a tempo pela réplica primária;

(b) Se a carga de acesso aumentar e o tempo de resposta aumentar para mais de 200ms, até 4seg, 4 servidores redundantes serão alocados e a técnica de replicação ativa cíclica será utilizada. A política é aumentar o número de réplicas para aumentar a vazão de requisições tratadas e utilizar uma técnica de replicação mais robusta, mesmo com consumo maior de recursos.

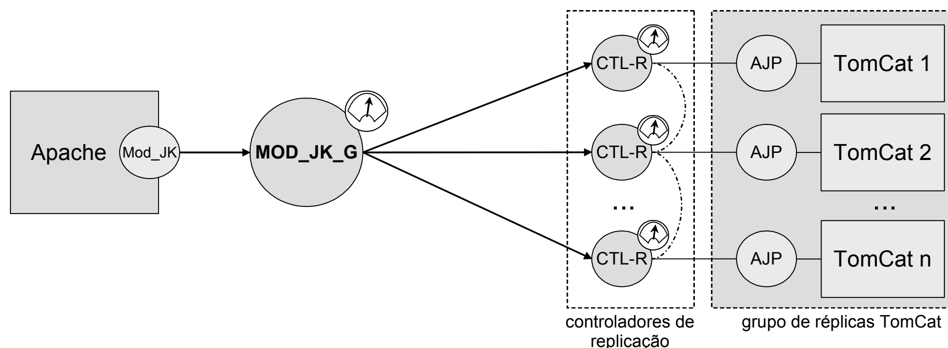

Figura 3 Arquitetura do exemplo

A Figura 3 apresenta o diagrama geral da arquitetura da aplicação. Uma requisição vinda de um cliente Web é processada pelo módulo *Apache* que identifica através da URL que se trata de conteúdo dinâmico, que deve ser processado por um módulo *TomCat*. Em um cenário sem replicação a comunicação Apache-Tomcat é realizada por dois elementos: o Mod_JK e o AJP, conectores disponibilizados pelos respectivos produtos. Em nossa arquitetura este fluxo é interceptado pelo conector *MOD_JK_G*, que implementa a comunicação de grupo.

Ao receber uma requisição do Apache o conector *MOD_JK_G* difunde a requisição para os conectores CTL-R de cada réplica *TomCat.* Estes realizam os procedimentos de consistência adequados à técnica de replicação estabelecida, e encaminham a requisição se for apropriado. Por exemplo, no caso de replicação ativa cíclica o CTL-R com a posse da ficha encaminha a requisição para processamento no módulo *TomCat* correspondente, via conector AJP e as demais descartam a mesma. Após processar a requisição, o módulo *TomCat* devolve a resposta ao seu respectivo CTL-R que coloca a mesma no seu caminho de volta. No caso da replicação passiva, o CTL-R da réplica primária repassaria a requisição para o módulo *TomCat*, e enviaria o estado para as réplicas secundárias de acordo com o a propriedade *checkPointInterval*.

4.1. Arquitetura e contrato

Uma vez traçada a política de tolerância a faltas, a arquitetura da aplicação é descrita (Listagem 3). As classes de módulo são listadas: *Apache*, que atuará como um cliente, e *TomCat*, que será replicado (l: 1). O mesmo se dá com os conectores específicos utilizados na arquitetura da aplicação (l: 2).

```
1   module Apache, TomCat;
2   connector Mod_JK_G, AJP, CTLRp, CTLRs, CTRLa;
3   module{
4      group TCGroup; // referência ao grupo de réplicas TomCat
5      instantiate Apache as ap, Tomcat as tc; // alocação de módulos
6      join tc to TCGroup;
7      link ap to TCGroup by Mod_JK;
8   }  webApp;
9   start webApp under webContract;
```

Listagem 3. Descrição da arquitetura do exemplo

Uma referência para o grupo de réplicas *TomCat* é criada (l: 4), e as referências para as instâncias dos módulos são declaradas. Instâncias de conectores são criadas automaticamente. O módulo *tc* é incluído no grupo *TCGroup* (inicialmente o grupo tem apenas um elemento). Por fim, a topologia é descrita, ligando-se o módulo *ap* aos elementos do grupo *TCGroup* através do conector *Mod_JK_G*. A linha 9 declara que o módulo *webApp* deve ser iniciado sob o contrato *webContract* (descrito adiante). Devido ao espaço reduzido, não detalhamos a semântica das descrições em CBabel. Vale observar que a descrição é declarativa. Ações de configuração são necessárias, em tempo de implantação, para carregar a aplicação de acordo com esta descrição.

O próximo passo é a descrição do contrato que especifica no nível arquitetural os requisitos descritos coloquialmente. No exemplo, cada regra descrita anteriormente dá origem a um serviço da arquitetura contendo uma configuração diferenciada:

passServ, para a replicação passiva com "reserva quente" (*hot standby*) onde uma réplica é eleita primária e apenas esta processa efetivamente as requisições. As réplicas secundárias são atualizadas a cada *checkpoint;*

actCServ, para replicação ativa cíclica, estilo *round-robin*, onde as várias réplicas se revesam como primária, circulando uma ficha. Neste caso não existe atualização de estado baseado em *checkpointing*.

```
1  profile {                                      7  profile {
2    Replication.checkPointInterval = 10;         8   Faults.numberOfFaults=2;
3    Replication.monitoringInterval = 20;         9   Faults.faultInterval=15;
4    Replication.timeoutInterval = 200;          10  } PassiveFaults;
5  } PassiveP;
6  profile{ Replication.numberOfReplicas = 2; } PassNRepP;
```

Listagem 4. Perfis para a replicação passiva

Cada configuração é associada a perfis relacionados às categorias *Replication* e *Faults*. Os perfis para a replicação ativa cíclica foram descritos na Seção 3. Para a configuração com replicação passiva (Listagem 4) o perfil *PassiveP* descreve que o intervalo de *checkpointing* da réplica primária é de 10s (l: 2), os conectores CTL-R serão monitorados a cada 20 s (l: 3) e a resposta deve ser dada em até 200ms (l: 4). O perfil *PassNRepP* indica que 2 réplicas são necessárias. O perfil *PassiveFaults* indica

um máximo de 2 faltas a cada 15s (l: 8-9). Como *stableInterval* não foi especificada, se as outras propriedades estiverem válidas, a configuração será considerada estável.

Para contemplar o aspecto de tempo de resposta do grupo, uma categoria relacionada à comunicação é utilizada. Os perfis representam os requisitos de tempo de resposta para cada técnica de replicação (Listagem 5). O tempo de resposta é monitorado, em nosso exemplo, pelo AR associado ao conector de grupo *MOD_JK_G*.

```
1   category Communication { ... resposeTime: decreasing numeric ms; ...}
3   profile { Communication.resposeTime >= 200 and <= 4000; } comActCP;
4   profile { Communication.resposeTime < 200; } comPassP;
```

Listagem 5. Categoria de comunicação e perfis de tempo de resposta

Neste ponto descreve-se o contrato *webContract* (Listagem 6). Em cada serviço, *passServ* (l: 2-8) e *actCServ* (l: 9-15), estruturas da arquitetura são especializadas para incorporar os elementos arquiteturais da replicação e para associar as mesmas aos perfis adequados. No serviço *passServ* o grupo *TCGroup* passa a ser restrito pelo perfil *PassNRepP e PassiveFaults* (l: 3) e somente será válido se todas as propriedades destes estiverem válidas. Um vetor de módulos *TomCat* é estruturado com o número de réplicas desejado, sendo que cada instância selecionada é restrita pelo perfil *PassiveP* (l: 4-5). Em seguida, o módulo é incorporado ao grupo (l: 6). Por último, o módulo Apache, *ap*, é ligado ao grupo *TCGroup* por uma composição dos conectores Mod_JK_G, *CTLRp*, um CTL-R especializado para replicação passiva, e o conector *AJP* para adaptar a interface dos módulos *TomCat.* Esta composição é associada ao perfil *comPassP*, que restringe o tempo de resposta (l: 7).

```
 1  contract{
 2    service{
 3      group TCGroup with PassNRepP, PassiveFaults;
 4      for (i=0; i < PassiveP.Replication.numberOfReplicas; i++) {
 5        instantiate tc[i]=select*(TomCat) with PassiveP;
 6        join tc[i] to TCGroup;                               }
 7        link ap to TCGroup by Mod_JK_G > CTLRp > AJP with comPassP;
 8    } passServ;
 9    service{
10      group TCGroup with ActNRepP, ActiveCFaults;
11      for (i=0; i < ActiveCP.Replication.numberOfReplicas; i++ {
12        instantiate tc[i]=select*(TomCat) with ActiveCP,;
13        join tc to TCGroup;                                 }
14      link ap to TCGroup by Mod_JK_G > CTLRa > AJP with comActCP;
15    } actServ;
16    negotiation {
17       not passServ -> (actCServ || out-of-service);
18           actServ  ->  passServ;                   };
19  } webContract;
```

Listagem 6. Contrato da aplicação de exemplo

A construção *select* indica que o módulo específico vai ser selecionado dinamicamente através do SD. É parametrizada pela classe do módulo (*TomCat*, no caso) e pelos perfis associados à declaração *instantiate*. Em nosso caso, o perfil especifica que a réplica selecionada deve ter o parâmetro *timeoutInterval* validado (l: 5 e 14). A construção *select** indica que o SD será continuamente monitorado e se uma outra instância, mais apta, da classe requerida estiver disponível, o módulo atual será substituído. Com isso é possível realizar reparos, localizados e atômicos [Cardoso 2007], na configuração sem a necessidade da intervenção do GR. Por exemplo, no caso

iminente de uma falha em um nó, monitorado pelo *timeoutInterval*, a referência de um novo módulo, mais apto pode ser descoberta, substituindo o módulo atual. Com isso evita-se que a seqüência de notificações de perfil e serviço inválidos, e que uma nova negociação, e implantação de uma outra configuração sejam disparadas.

A cláusula de negociação efetivamente mapeia os requisitos de tolerância a faltas em uma máquina de estados, que determina a política de implantação, a prioridade e as possíveis transições entre as configurações previamente descritas. Uma vez a aplicação em operação, esta política é gerenciada autonomicamente. O sistema só voltará a sofrer intervenção manual se nenhuma das configurações previstas no contrato puder ser implantada ou mantida. A ordem das regras de negociação (l: 16-18) determina sua prioridade e o serviço da parte esquerda é aquele a ser implantado e monitorado. A configuração prioritária (l: 17) é a descrita no serviço *passServ*. Se esta não puder ser implantada ou mantida (daí o *not*) porque um dos perfis foi violado, por exemplo, o GC tentará implantar o serviço *actCServ*. Isto é, se o número de faltas aumentar, uma configuração de replicação mais robusta será utilizada. Caso nenhuma das configurações possa ser implantada, um serviço especial, *out-of-service*, é implantado, indicando que a aplicação não pode executar com a qualidade requerida. Já na regra da linha 18, não existe a condição "not". Isso quer dizer que se a configuração atual é a do serviço *actCServ* (ou seja os perfis deste serviço estão válidos), e o serviço *passServ* puder ser implantado (ou seja se os perfis deste serviço também são válidos), então a transição para este é incondicional. Com isso podemos expressar o requisito de voltar ao sistema uma técnica de replicação que exigem menos recursos, com menos réplicas.

4.2 Implantação e aspectos de implementação

Para validar a abordagem proposta e o contrato *webContract*, desenvolvemos alguns componentes específicos da aplicação e integramos os mesmos à infra-estrutura previamente desenvolvida [Corradi 2005], [Santos 2006]. Desenvolvemos em Java as classes do conector de grupo *MOD_JK_G,* os conectores CTL-R e customizamos as classes abstratas de ARs para as categorias *Replication e Faults.*

A comunicação original entre o Apache e o Tomcat se dá através do conector Mod_JK [Apache 2007c], que encaminha as requisições para o conector AJP, componente padrão do Tomcat. Em nossa solução, o conector *MOD_JK_G* é interposto neste caminho para receber do Mod_JK as requisições do Apache e realizar a comunicação de grupo, encaminhando estas requisições para o grupo de CTL_Rs. Foi necessário também contemplar os protocolos específicos do Mod_JK e do AJP dentro do código. A Figura 4 apresenta o diagrama de interações simplificado desta composição de elementos. A interação *1.2.1.1.1* é justamente a comunicação de *1:n* e o retorno correspondente é realizado de *n:1*.

A comunicação de grupo foi implementada com o pacote JGoups [Ban 2007], através da classe *RPCDispatcher*, que provê um mecanismo de invocação dinâmica no cliente e a chamada de procedimentos nos servidores remotos (um pouco mais complexo que um RPC para grupo). Observe na chamada a seguir que o segundo parâmetro, "*sendBytes",* é o nome do método remoto a ser invocado em cada réplica, *buffer* contém os dados encaminhados pelo Apache, *class* é um vetor com os tipos dos parâmetros em *buffer* (usado para remontar a invocação por reflexão no lado remoto).

```
RspList rsp_list = disp.callRemoteMethods((Vector) null, "sendBytes",
      new Object[] { buffer }, new Class[] { byte[].class },
      GroupRequest.GET_ALL, 0L);
```

O lado "servidor" (*ServerConnector*, na Figura 4) encapsula a funcionalidade dos conectores CTL-Rs, implementando as características específicas de cada técnica de replicação, e faz interface com o conector AJP. Foi necessário adaptar esta interface ao esquema de reuso de conexões abertas entre o Apache e o Tomcat. Para isso fizemos uso de *threads*, através do pacote JNIO [Sun 2007]. O suporte provido pelo JNIO para o uso de *threads* e para chamadas de E/S não bloqueantes é mais escalável. Para implementar as especializações de CTL-R para cada técnica de replicação optou-se por usar do padrão *strategy* [Gamma 1995] ao invés de classes separadas.

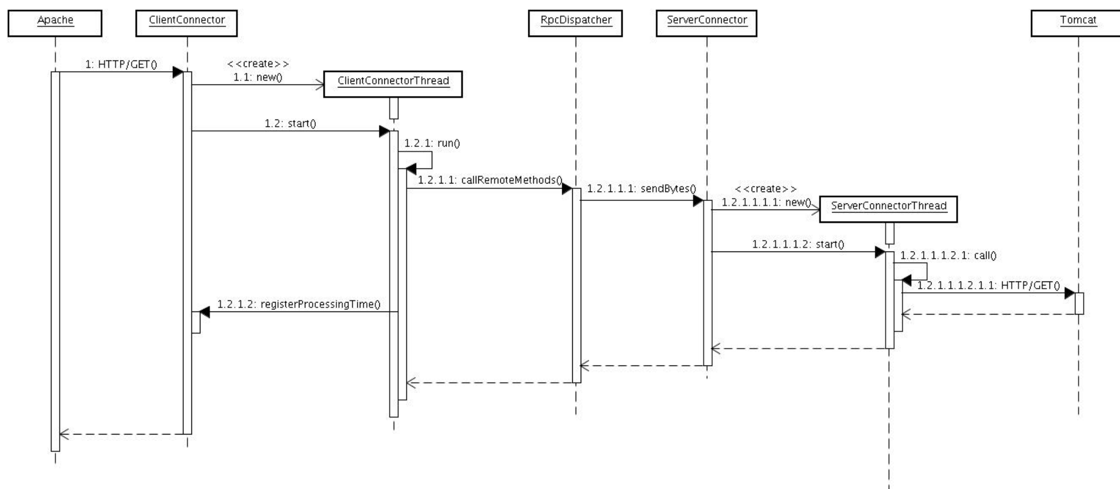

Figura 4. Diagrama de interação dos conectores MOD_JK_G > CTL-R > AJP

A implementação das várias estratégias de replicação está sendo concluída. Atualmente temos a replicação ativa (não utilizada no exemplo, pois tem o efeito de aumentar muito o tempo de resposta) e ativa cíclica. Alguns testes de desempenho estão sendo realizados com a ferramenta JMeter [Apache 2007d] que permite "estressar" o servidor Apache. Num teste preliminar, configuramos o JMeter para simular 10 usuários simultâneos solicitando documentos de 8Kbytes a duas réplicas Tomcat executando na mesma máquina, com replicação ativa. O tempo medido no teste foi de ~4s, contra ~400ms do teste realizado com apenas uma instância do Tomcat e sem a infra-estrutura de replicação (ordem de grandeza similar a encontrada em [Favarim 2004]). Além de concluir e refinar a implementação, serão realizados testes em cenários distribuídos, como pede o exemplo, também considerando o número de adaptações realizadas. Com isso será também possível detectar limitações da abordagem.

5. Trabalhos Relacionados

Servidores de aplicação como o JEE e .NET e oferecem mecanismos de replicação, mas não permitem configurar a técnica utilizada. No servidor de aplicação JBoss é possível utilizar a infra-estrutura JBossCache [JBoss 2007] e AOP (*Aspect Oriented Programming*) para replicar objetos do *cache* de diferentes formas. No entanto o uso destes mecanismos é *ad hoc*. Considerando o Apache-Tomcat, é possível configurar o balanceamento de carga de forma nativa através do conector Mod_JK [Apache 2007c]. No entanto, técnicas de replicação adaptativas também não são suportadas.

Uma abordagem adaptativa para tolerância a faltas em serviços replicados também é explorada em [Kalbarczyk 2005] numa linha semelhante a deste trabalho. A preocupação principal dos autores é a arquitetura e o desempenho do suporte. As especificações são, no entanto, *ad hoc* e embutidas no código dos serviços.

A organização dos elementos de tolerância a faltas em nossa abordagem é baseada em [Lung 2006], uma extensão do padrão FT-CORBA da OMG. No entanto, este padrão também não é adaptativo. Uma vez definidos os requisitos de tolerância a faltas os mesmos não podem ser alterados de acordo com as necessidades da aplicação. Em [Lung 2007] é apresentada uma infra-estrutura para tolerância a faltas adaptativas chamada GroupPac, uma implementação livre do padrão FT-CORBA. Através desta infra-estrutura é possível criar programas baseados em CORBA que mudam suas propriedades de tolerância a faltas de acordo com regras codificadas no programa. No entanto, estas regras são programadas de forma *ad hoc*. Em nossa abordagem a tolerância a faltas é especificada em nível alto e integrada a uma infra-estrutura de suporte recorrente para vários domínios de aplicação.

6. Conclusões e Trabalhos Futuros

Neste trabalho apresentamos uma abordagem que provê uma forma para especificar políticas de tolerância a faltas, em nível alto de abstração, através de contratos, e uma infra-estrutura de software reusável para implantar e manter a política especificada. Uma contribuição de nossa abordagem é a ligação entre a semântica dos contratos e as ações correspondentes no sistema de suporte. Além disso, esta semântica permite a realização de verificações formais das especificações de tolerância a faltas, antes de implantar a aplicação [Braga 2007]. Observa-se, entretanto, que nem todas as técnicas de replicação podem ser aplicadas a qualquer aplicação. Por exemplo, na configuração de teste utilizando a replicação ativa observa-se que o tempo de resposta aumenta em muitas vezes se comparado com a replicação ativa cíclica.

Um outro aspecto importante é a auto-configuração e auto-otimização da aplicação de acordo com a política especificada no contrato, e em resposta ao contexto de operação, requisitos importantes da computação autonômica [Sztajnberg 2006]. A configuração utilizando a técnica de replicação mais adequada ao contexto vigente é implantada dinamicamente, mantendo-se a aplicação dentro da qualidade requerida.

Outros requisitos não-funcionais podem ser considerados no contrato, além dos relacionados à tolerância a faltas, tais como a capacidade da CPU e a memória, para aprimorar a política de decisão dos contratos. Por exemplo, o número de máquinas atendendo ao grupo de réplicas, a disponibilidade de CPU e memória de cada uma delas pode compor as características dos perfis. A política neste caso seria prover um serviço robusto, onde a carga das réplicas seria balanceada, a um custo de energia aceitável (tornando o *data center* mais "verde") [Freitas 2005]. Neste contexto, onde múltiplas variáveis (perfis), potencialmente conflitantes, são utilizadas, o contrato baseado em uma maquina de estados pode tornar-se complexo. Para estes casos estamos investigando a aplicação de funções de utilidade [Petrucci 2007], em que pesos são atribuídos a cada propriedade para expressar sua importância e, de acordo com o contexto a seleção de uma dada configuração será resultado desta função. Alguns resultados obtidos podem ser consultados em [Petrucci 2007] e [Leal 2007].

Agradecimento. Os autores gostariam de agradecer o apoio da FAPERJ e CNPq.

Referências

Apache.org (2007a) “Apache HTTP Server Project”. http://httpd.apache.org/

Apache.org (2007b) “Apache Tomcat”. http://tomcat.apache.org/

Apache.org (2007c) “The Apache Tomcat Connector - Reference Guide”. http://tomcat.apache.org/connectors-doc/reference/apache.html

Apache.org (2007d) “Apache JMeter”. http://jakarta.apache.org/jmeter

Ban, B. et al. (2007) “Jgroups - A Toolkit for Reliable Multicast Communication”, outubro. http://www.jgroups.org/javagroupsnew/docs/index.html,

Braga, C., Chalub, F. R., Sztajnberg, A. (2007) “A Formal Semantics for a Quality of Service Contract Language”, FESCA@ETAPS 2007, Braga, Portugal.

Cardoso, L. X. T., Sztajnberg, A., Loques, O. (2007) “Proposta de uma infra-estrutura de suporte para serviços de contexto e descoberta de recursos”, IV WSO / CSBC’07, pp. 875-886, Rio de Janeiro.

Corradi, A. (2005) “Um framework de suporte a requisitos não-funcionais de nível alto”, Dissertação de Mestrado, Instituto de Computação, UFF.

Favarim, F., Fraga, J., Lung, L. C e Siqueira, F. (2004) “Suporte de Tolerância a Faltas Adaptativa para Aplicações Desenvolvidas em CCM”, 22° SBRC, Gramado.

Freitas, G., Cardoso, L., et al. (2005) “Otimizando a Utilização de Servidores através de Contratos Arquiteturais”, VII WTR / 23º SBRC, pp. 35-43, Fortaleza.

Gamma, Erich, et al (1995) “Design Patterns – Elements of Reusable Object-Oriented Software”, Addison Wesley, 1ª edição.

Gorender, S., Cunha, P. R., Macedo, R. J. (2005) ”The Implementation of a Distributed System Model for Fault Tolerance with QoS”, 23o, pp. 827-840, Fortaleza.

Jalote, P. (1994) “Fault Tolerance in Distributed System”, Prentice-Hall.

JBoss.org (2007) “JBoss Cache”. http://labs.jboss.com/jbosscache/

Kalbarczyk, Z., Iyer, R. K., Wang, L. (2005) “Application Fault Tolerance with Armor Middleware”, IEEE Internet Computing, Vol. 9, No. 2, pp. 28-37.

Leal, D. A. S. (2007) “Suportando a adaptação de aplicações pervasivas pelo uso de funções utilidade”, Dissertação de Mestrado, Instituto de Computação, UFF.

Lisbôa, J. C. e Loques, O. G. (2004) “Um Padrão Arquitetural para Gerenciamento de Qualidade de Serviço em Sistemas Distribuídos”, SugarLoafPLoP 2004, Fortaleza.

Loques, O., et al. (2004) “A contract-based approach to describe and deploy non-functional adaptations in software architectures”, JBCS, Vol. 10, No. 1, pp. 5-18.

Lung, L. C., Favarim, F., et al. (2006) “An Infrastructure for Adaptive Fault Tolerance on FT-CORBA”, 9th IEEE ISORC, pp. 504-511, Coréia do Sul.

Lung, L. C., Padilha, R. (2007) “GroupPac”. http://sourceforge.net/projects/grouppac.

Petrucci, V., Sztajnberg, A.; Loques, O. G. (2007) “Seleção de Recursos em Grades Computacionais usando Funções de Utilidade”, IV WCGA / 25º SBRC, Belém.

Santos, A. L. G.; Leal, D. A. e Loques, O. G. (2006) “Um suporte para adaptação dinâmica de arquiteturas ubíquas” XXXII CLEI, Santiago, Chile.

Sun Microsystems, Inc (2007) “New I/O APIs”. http://java.sun.com/j2se/1.5.0/docs/guide/nio

Sztajnberg, A.; Loques, O. G. (2006) “Describing and Deploying Self-Adaptive Applications”, 1st LAACS / CSBC 2006, Campo Grande.