AVALIAÇÃO DA VIABILIDADE DE UM SERVIÇO DE DETECÇÃO DE SUSPEITAS INCORRETAS

Trabalho de Conclusão de Curso

Engenharia da Computação

Francisco Miranda Soares da Silva Neto
Orientador: Prof. Sérgio Murilo Maciel
Co-orientador: Prof. Fernando Castor Filho

Universidade de Pernambuco
Escola Politécnica de Pernambuco
Graduação em Engenharia de Computação

FRANCISCO SOARES NETO

AVALIAÇÃO DA VIABILIDADE DE UM SERVIÇO DE DETECÇÃO DE SUSPEITAS INCORRETAS

Monografia apresentada como requisito parcial para obtenção do diploma de Bacharel em Engenharia de Computação pela Escola Politécnica de Pernambuco – Universidade de Pernambuco.

Recife, junho de 2011.

De acordo

Recife

____/_____/_____

_____
Prof. Sérgio Murilo Maciel

_____
Prof. Fernando Castor Filho

Aos meus pais.

Agradecimentos

Não foi fácil concluir este trabalho. Por diversas vezes pensei em desistir. Consegui, finalmente, terminar essa tarefa hercúlea. Mas não conseguiria sem o apoio dessas pessoas, a quem agradeço:

A meus colegas de trabalho, que me ajudaram a agüentar a barra desses meses de TCC. Valeu pela força, e pelo café.

Aos amigos da faculdade – obrigado por terem facilitado os momentos difíceis das disciplinas, e tornado o processo de adquirir conhecimento mais simples.

Aos colegas de Iniciação Científica – Anderson Murilo, Bruno, João Paulo, Rafael e Rodrigo, companheiros nos meus primeiros passos acadêmicos. Sofremos juntos, e crescemos juntos.

Aos meus professores que tanto me influenciaram no curso – em especial Márcio Cornélio, Sérgio Soares e Luís Meneses. Sei que o conhecimento que adquiri com vocês é e para sempre será extremamente útil.

Aos meus orientadores – Sérgio Murilo pela ajuda com o TCC; e Tiago Massoni e Fernando Castor, por serem meus pais na vida científica. Agradeço-os profundamente pelo voto de confiança que me depositaram. Muito mais do que professores, vocês são modelos do que almejo alcançar.

Aos meus amigos, dos quais os estudos tantas vezes me separam: obrigado por sempre estarem presentes quando precisei. Agora terei tempo de fazer barulho, de sair pra jogar conversa fora, e de aprender russo. Ao menos assim espero.

À minha namorada, Fernanda, que teve seu convívio comigo tão frustrado por conta deste trabalho. Juro que agora terei mais tempo. =)

Agradeço, por último, mas não menos importante, à minha família por todo apoio nos momentos da produção desse trabalho. Acima de todos, eu não conseguiria sem eles.

Resumo

Computação distribuída e suas variações nas quais aplicações são submetidas a um sistema para execução, tais como computação em grade ou em nuvem, dependem da habilidade de se ter certeza que um programa está em execução ou não. O não cumprimento desse requisito pode levar a perdas significativas para as pessoas ou instituições que dependem de tais serviços, desde simplesmente tempo desperdiçado, a perdas financeiras. Ainda, infraestruturas existentes não são confiáveis e, frequentemente, consideram erroneamente como falhos nós que estão em pleno funcionamento. Este trabalho lida com casos de mal-funcionamento do serviço que provê dada habilidade, e apresenta uma maneira simples e eficiente de evitá-los – o Serviço de Detecção de Suspeitas Incorretas.

Abstract

Distributed utility computing systems where applications are submitted to a system for execution, such as grid and cloud computing, rely on the ability to be sure if a program is executing or not. Not fulfilling this requirement may incur in significant losses for the people or institutions who depend on these services, ranging from simply waste of time to financial losses. At the same time, existing infrastructures are unreliable and often mistakenly consider that working nodes have failed. This work deals with malfunction cases of the service which provides such ability, and presents a simple and efficient way to avoid them – the Wrong Suspicion Detection Service.

Sumário

Índice de Figuras

Índice de Tabelas

Tabela de Símbolos e Siglas

IETF – *Internet Engineering Task Force* (Força-tarefa de engenharia da Internet)

JVM – *Java Virtual Machine* (máquina virtual Java)

UI – *User Interface* (interface com o usuário)

WAN – *Wide Area Network* (Rede de grande área)

XMPP – *Extensible Messaging and Presence Protocol* (protocolo extensível de troca de mensagens e presença)

Capítulo 1

Introdução

Computação distribuída e suas variações nas quais aplicações são submetidas a um sistema para execução, tais como computação em grade (KESSELMAN e FOSTER, 1998) ou em nuvem (ANTONOPOULOS e GILLAM, 2010), dependem da habilidade de se ter certeza que um programa está em execução ou não. Para isso, faz-se necessário o uso de um serviço detecção de defeitos (CHANDRA e TOUEG, 1996), o qual permite que o sistema detecte problemas no funcionamento de nós que o compõe.

Idealmente, um detector de defeitos deve ser capaz de detectar corretamente e em tempo finito as falhas de todos os nós que, de fato, falharam (CHANDRA e TOUEG, 1996). Isso ocasiona que processos que estão funcionando corretamente sejam detectados como falhos, geralmente pelo esgotamento de um limite de tempo de espera para comunicação. Apesar de isso poder ser considerado intuitivamente como uma boa medida para o sistema em geral, dado que são descartados processos que de um modo ou de outro estariam retardando a comunicação, o descarte de processos corretos já foi estudado como prejudicial ao funcionamento geral do sistema (SAMPAIO, BRASILEIRO, *et al.*, 2003).

No contexto de grades computacionais, ou mesmo de computação em nuvem, o tempo de processamento de dados pode ser contabilizado em uma transação comercial. Logo, a suspeita incorreta de um componente pode gerar problemas econômicos, já que cada perda de componente deve ser enxergada como uma perda de lucro para o administrador do sistema. Ao mesmo tempo, a manutenção e execução de cada nó da grade têm custos quando um nó não está fazendo trabalho útil. Tais custos aumentam devido à necessidade de retrabalho. Usuários que dependem da computação também podem ser prejudicados. Esses podem não ser atendidos quando requisitam uma execução, visto haver a necessidade de nós corretos compensarem outros erroneamente considerados falhos. Além disso, tempo de trabalho é perdido pela interrupção de uma

computação que requer um determinado período de tempo para execução (SAMPAIO, BRASILEIRO, *et al.*, 2003).

1.1 Breve descrição da solução

Este trabalho lida com casos de mau funcionamento do serviço de detecção, onde processos corretos são erroneamente suspeitos, e visa apresentar uma solução simples e eficiente para corrigir o problema em uma grade que utiliza simples limites de tempo para suspeitar de componentes como falhos.

A solução aqui apresentada foi implementada sobre uma plataforma de *middleware* de grade. Os passos na detecção de defeitos e as ações tomadas pela grade, dado o ocorrido, serviram como base para análise das mudanças necessárias. Assim, o novo serviço foi inserido na grade, para determinar quando e como reagir a possíveis suspeitas incorretas.

1.2 Organização do trabalho

Este trabalho apresenta em sua estrutura: no capítulo 2, os conceitos fundamentais para definição da solução. No Capítulo 3, é descrito o raciocínio no qual a solução proposta se baseia. No Capítulo 4 é feita uma avaliação do impacto da solução no comportamento da plataforma de *middleware* para grades computacionais, e sua eficácia. O Capítulo 5, por fim, apresenta considerações finais sobre o trabalho e uma análise sobre trabalhos futuros.

Capítulo 2

Fundamentos

Este capítulo resume os fundamentos sobre os quais se baseiam este trabalho. São descritas, a seguir, as bases de tolerância a falhas de sistemas e de grades computacionais, e brevemente a arquitetura da plataforma de *middleware* de grade computacional cuja implementação foi estendida neste trabalho.

2.1 Tolerância a falhas

Na construção de sistemas, espera-se satisfazer requisitos de funcionamento. Esses requisitos requerem uma especificação determinada de funcionamento correto, cuja satisfação determina a corretude do sistema.

Entretanto, todo sistema tende a falhar. Com o passar do tempo, componentes se desgastam, estruturas se deterioram, e o comportamento do sistema tende a se desviar de seu padrão de corretude. O crescimento de complexidade de sistemas apenas serve para intensificar esse problema, o que gera a necessidade cada vez maior da preocupação com a Tolerância a Falhas desses.

2.1.1 Falha, erro e defeito

Para discutir-se tolerância a falhas, é necessário primeiro diferenciar-se os conceitos básicos desse campo. Essa definição de conceitos é em geral acordada pela comunidade da área, e derivada dos trabalhos de Laprie (1985), Anderson e Lee (1990).

O conceito de **falha** (do inglês *fault*) define a origem física ou algorítmica do problema. Por exemplo, um *chip* de memória com um problema de *stuck-at-zero* (um bit de memória fixado no valor zero, ou em um) provavelmente levará à corrupção dos dados armazenados nele. A corrupção desses dados, que levarão a um problema no funcionamento do sistema de acordo com sua especificação, é o que caracteriza o **erro** (do inglês *error*). Um sistema está em erro se o processamento do sistema a partir desse estado pode levar a um desvio da especificação do sistema.

Assim, finalmente, o estado de desvio da especificação do sistema é determinado um **defeito** (do inglês *failure*). No exemplo apresentado, a interpretação dos dados corrompidos da memória pode fazer com que um sistema de controle de acesso não autorize acesso a usuário algum.

Define-se que os conceitos de falha, erro e defeito estão respectivamente associados ao universo físico, ao universo da informação e ao universo do usuário. A Figura 1 ilustra o modelo de três universos, onde se encontram os conceitos.

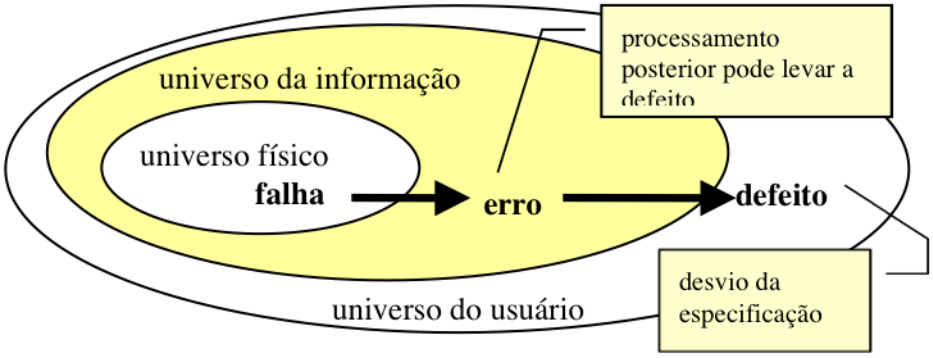

Figura 1. Modelo de três universos (WEBER, 2002)

Ainda assim, nem toda falha necessariamente leva a um erro, e nem todo erro necessariamente leva a um defeito. Por condições algorítmicas, a falha pode nunca ser acessada, e, portanto, nunca gerar um erro, assim como o defeito pode ser evitado ao se obter informação de outra fonte auxiliar, e ignorar-se o erro.

Falhas, entretanto, não podem ser evitadas. Componentes tendem a se deteriorar ou sofrer interferência externa. A complexidade afeta tanto os componentes de *hardware*, postos para trabalhar juntos, quanto componentes de *software* em execução paralelamente e com freqüente interferência humana. Das diferentes formas pelas quais podem ser geradas, surgem diferentes tipos de falhas, classificáveis quanto à natureza, duração, extensão ou valor.

Dado as diversas interações em um sistema distribuído geograficamente, como uma grade computacional, existem diversas maneiras pelas quais esses sistemas podem apresentar falhas. Em busca da categorização dos problemas ocorridos em sistemas distribuídos, podem ser categorizadas as falhas desses sistemas, como apresentado na Tabela 1.

Tabela 1. Tipos de falhas em sistemas distribuídos (TANENBAUM e STEEN, 2007)

Tipo de Falha	Descrição
Falhas por Queda	Quando um servidor ou uma máquina que estava executando um componente, e que antes da falha estava em funcionamento normal, para de responder.
Falhas por Omissão	Quando uma máquina deixa de responder a requisições que chegam ou de enviar mensagens.
Falha de Temporização	Quando a resposta se encontra fora do limite de tempo configurado para espera.
Falha de Resposta	Quando a resposta retornada está incorreta. Por exemplo, uma resposta que deveria ser um número retorna uma *String*.
Falhas Arbitrárias	Quando a resposta retornada não é a correta, mas os valores não podem ser detectados como falhos.

2.1.2 Dependabilidade

Tolerância a falhas existe como forma de alcançar dependabilidade (do inglês *dependability*), cujas principais medidas são descritas na Tabela 2.

Tabela 2. Principais medidas de dependabilidade (PRADHAN, 1996)

Dependabilidade	Qualidade do serviço fornecido por um dado sistema.
Confiabilidade (do inglês *reliability*)	Capacidade de atender a especificações, dentro de condições definidas, durante certo período de funcionamento e condicionado a estar operacional no início do período.
Disponibilidade (do inglês *availability*)	Probabilidade de o sistema estar operacional em um instante de tempo determinado; alternância de

	períodos de funcionamento e reparo.
Segurança (do inglês *safety*)	Probabilidade de o sistema estar operacional e executar sua função corretamente ou descontinuar seu funcionamento de forma a não causar danos a outros sistemas ou pessoas que dele dependam.
Segurança (do inglês *security*)	Proteção contra falhas maliciosas, visando privacidade, autenticidade, integridade e irrepudiabilidade dos dados.

2.1.3 Técnicas de tolerância a falhas

Para se atingir dependabilidade, um conjunto de técnicas pode ser utilizado. Aqui o foco é dado a técnicas de tolerância a falhas, nas linhas das quais este projeto se concentra.

Técnicas de tolerância a falhas são utilizadas para garantir que um sistema permaneça em correto funcionamento, mesmo no evento de falhas. A base de todas as técnicas está na redundância, pela utilização de algoritmos especiais ou componentes para evitar o erro no sistema.

Duas classes de técnicas existem: (i) mascaramento de falhas; e (ii) detecção, localização e reconfiguração. A primeira classe mascara a falha através do uso de redundância, impedindo a manifestação do erro. A segunda utiliza-se da detecção do erro para o tratamento da falha. Aqui será abordado o uso desta segunda classe.

2.1.4 Detecção de erros

O processo de detecção, localização e reconfiguração exige uma ordem de passos a serem tomados para manter o comportamento de acordo com a especificação do sistema. Vários autores apresentaram diversas classificações das fases a serem seguidas. A seguir, são consideradas as fases classificadas por Anderson e Lee (1981):

- Detecção de erros;

- Confinamento e avaliação;
- Recuperação de erros;
- Tratamento da falha.

A primeira fase detecta o erro – após a falha ter se manifestado como tal. Para isso, um detector de defeitos pode ser utilizado para, através do monitoramento constante do sistema, determinar a ocorrência de erros. Em um sistema distribuído isso é comumente implementado, de forma a detectar corretamente, e em tempo finito, as falhas de todos os nós que, de fato, falharam (CHANDRA e TOUEG, 1996).

Devido à latência da falha, o sistema pode ter espalhado dados inválidos até a detecção do erro. Para lidar com isso, processos de confinamento e avaliação são utilizados, o que precisa ser rigorosamente planejado de acordo com as decisões de projeto.

A seguir, a fase de recuperação de erros envolve a troca do estado atual por um estado livre de falhas. O tratamento de falha, por fim, envolve a localização do ponto da falha, e posterior reparação da falha e recuperação do sistema.

2.1.5 Recuperação de erros

O processo de recuperação de erros é de especial interesse para este trabalho. A abordagem utilizada no serviço de detecção de suspeitas incorretas em muito se assemelha a um processo de detecção/recuperação de erros, voltado ao próprio detector de erros da plataforma de *middleware* para grades estudada.

Recuperação de erros pode ser executada através de duas formas:

- Recuperação por retorno (do inglês *backward error recovery*);
- Recuperação por avanço (do inglês *forward error recovery*).

O método de recuperação por retorno consiste em conduzir o sistema a um estado anterior antes da falha. A implementação disso é feita mais comumente através do uso de pontos de verificação (do inglês, *checkpoints*), pistas de auditoria (do inglês, *audit trails*), entre outras técnicas. É uma técnica computacionalmente cara para implementação, porém bastante genérica.

Recuperação por avanço, por outro lado, consiste em conduzir o sistema a um estado novo ainda não ocorrido desde a última manifestação do erro. Essa

técnica possui como desvantagem, em relação à por retorno, ser menos genérica, e requerer maior planejamento para aplicação.

Em geral, essas formas de recuperação são de simples implementação para sistemas de um único processo. Porém, ao se lidar com sistemas de processamento distribuído, com múltiplos processos em comunicação, a implementação torna-se complexa (JANSCH-PORTO e WEBER, 1997). Com uso de pontos de recuperação, por exemplo, temos que o ponto de um processo pode não coincidir com o de outro processo com o qual se comunica. Assim, para garantir a consistência do sistema levando em consideração as mensagens trocadas durante a falha, todos os nós podem ter que retornar seus pontos de recuperação. O pior caso disso pode ser alcançado quando o único estado consistente do sistema é o estado inicial. Isso também é denominado “efeito dominó”. Restrições de comunicação entre os processos podem ser necessárias para evitar este problema, o que torna o a técnica mais complexa.

2.2 Grades Computacionais

Uma grade computacional é composta por um grupo de recursos computacionais geograficamente distribuídos, conectados por uma rede de longa distância e uma plataforma de *middleware* capaz de gerenciar recursos heterogêneos e distribuídos. Usuários de uma grade podem submeter aplicações que serão executadas em um dos recursos distribuídos. Uma aplicação pode ser um processo simples ou trabalhos paralelos e computacionalmente complexos.

Grades podem ser divididas em dois tipos: grades dedicadas e grades oportunistas. Nas dedicadas, recursos são providos por um grupo pré-determinado de máquinas. Para as oportunistas, qualquer máquina pode executar tarefas, desde que tenham em execução os componentes necessários para provisão de recursos.

Os componentes principais de uma plataforma de *middleware* de grade, são apresentados na Figura 2. O componente **agente de acesso** (do inglês, *access agent*) permite aos usuários submeter aplicações para a grade, com possibilidade de vários agentes em interação com a grade concomitantemente. Ele permite a configuração de restrições específicas quanto à execução das tarefas, e deve ser

executado em todas máquinas nas quais tarefas sejam submetidas. O **serviço de escalonamento** (do inglês, *scheduling service*) recebe requisições, verifica usuários e usa um serviço de monitoramento para verificar que provedores podem executar a aplicação. Vários nós **provedores de recursos**, quando ociosos, podem receber tarefas para execução pelo serviço de escalonamento. Em grades oportunistas, falhas em provedores de recursos são freqüentes, dado que o serviço de provisão de recursos é interrompido quando um usuário requer uso dos recursos da máquina. Quando um provedor de recursos termina a execução de uma tarefa, ele retorna seus resultados para o agente de acesso. O **serviço de segurança** protege provedores de recursos através da limitação das permissões do sistema às aplicações da grade. Isso é implementado através de formas de *sandboxing*, que podem exigir interceptação de chamadas do sistema ou virtualização.

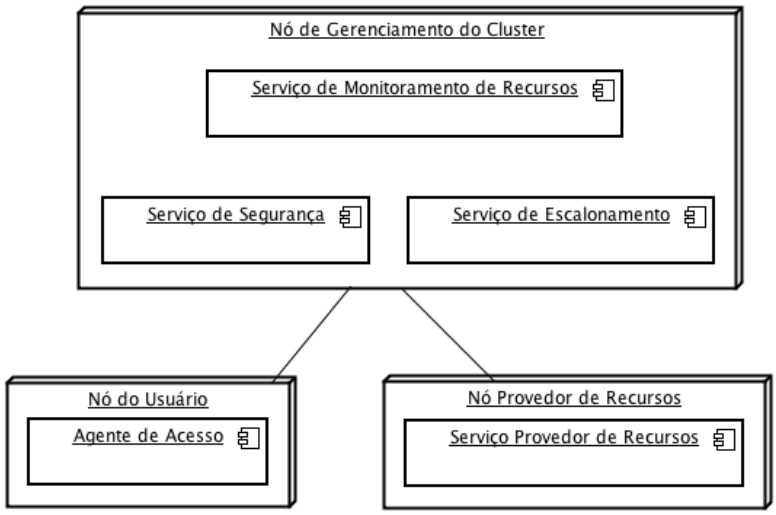

Figura 2. Uma implantação típica de *middleware* de grades computacionais (adaptado de (CAMARGO, GOLDCHLEGER, *et al.*, 2004))

Várias plataformas de *middleware* de grade são disponíveis para uso público, tais como a OurGrid (CIRNE, BRASILEIRO, *et al.*, 2006), a InteGrade (GOLDCHLEGER, KON, *et al.*, 2004) e a Globus (FOSTER e KESSELMAN, 1996).

2.3 OurGrid

A implementação do serviço de detecção de suspeitas incorretas utilizada neste trabalho se baseia no OurGrid, uma conhecida plataforma de *middleware* para

computação em grade. Nesta seção são detalhados alguns aspectos da plataforma importantes para o entendimento da solução.

2.3.1 Arquitetura de comunicação dos componentes

A arquitetura do OurGrid contém quatro componentes principais: o *Peer* (parceiro, no inglês) que gerencia os recursos da grade em geral, distribuindo provedores de recursos entre os que requisitam ter uma tarefa executada; o *Broker* (agente, no inglês) que é responsável por submeter tarefas de usuários e apresentar os resultados de volta aos usuários que os requisitaram; o *Worker* (trabalhador, no inglês) que é responsável pela execução de aplicações dos usuários que foram submetidas por um *Broker*; ou seja, cada máquina provedora de recursos executa um componente *Worker*; e um substrato de comunicação implementado na forma de um servidor do protocolo *XMPP* (Protocolo extensível de troca de mensagens e presença, do inglês *eXtensible Messaging and Presence Protocol*) (IETF, 2004), que também é responsável pela gerência de usuários da grade. Todos componentes são identificados no servidor de comunicação com nome de usuário e senha, e internamente com identificadores únicos para cada componente. Cada requisição de trabalho enviada por um *Broker* também possui um identificador único. A estrutura da grade está, em grande parte, dentro de padrões descritos na literatura (CAMARGO, GOLDCHLEGER, *et al.*, 2004), o que também oferece maior possibilidade de generalização da solução implementada nele.

Quanto à utilização por usuários, a arquitetura do OurGrid é organizada, como ilustrado na Figura 3, em quatro componentes principais - *Broker*, *Peer*, *Worker* e *Discovery Service* (Serviço de Descoberta, no inglês). Pode-se notar também que vários *Brokers* podem interagir com a grade ao mesmo tempo. O componente *Discovery Service* provê funcionalidade de comunicação entre grupos gerenciados por *Peers* diferentes. O escopo deste trabalho atém-se apenas a grupos isolados de componentes, portanto ele não será abordado.

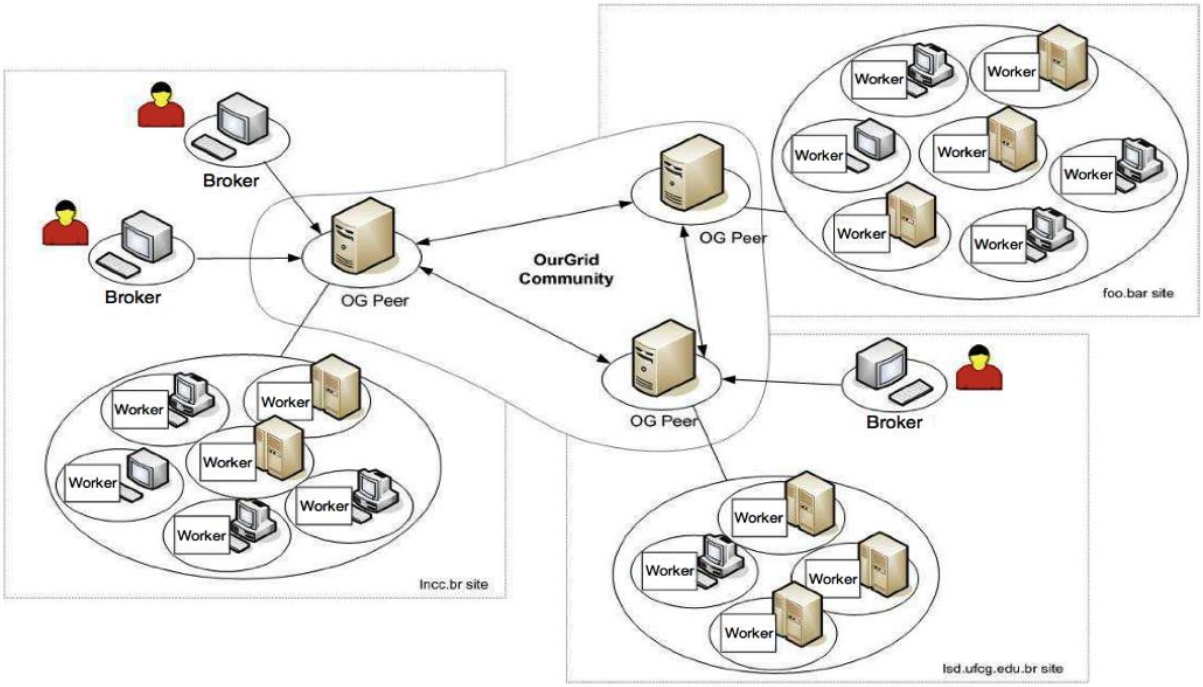

Figura 3. Componentes do OurGrid (adaptado de (CIRNE, BRASILEIRO, *et al.*, 2006))

Cada componente – *Broker*/*Peer*/*Worker* –, possui por sua vez uma complexa arquitetura interna, onde cada responsabilidade é atribuída a um elemento diferente. Cada componente é dividido em três elementos principais: Negócios (do inglês, *Business*), Comunicação (do inglês, *Communication*) e UI (*User Interface*, sigla comum inglesa para interface com o usuário). Como representado na Figura 4, esses elementos correspondem a pacotes no sistema, que englobam interfaces com chamadas genéricas para instâncias dos elementos. Para cada elemento, definem-se interfaces, como descrito a seguir:

- Negócios:
  - *Requester* (Requisitante, no inglês): a interface *business.requester.RequesterIF* serve de protocolo para as classes que tomam as decisões lógicas do componente. Assim, para cada requisição de alteração de configurações em um componente, classes que implementam o protocolo serão chamadas; essas processarão a requisição, e agregarão a uma lista de respostas objetos que implementam o protocolo *IResponseTO*. A interpretação dessas respostas define o

comportamento do sistema a partir de uma determinada requisição.

- DAO (Objeto de Acesso a Dados, do inglês: *Data Access Object*): este pacote engloba os objetos de acesso (SUN MICROSYSTEMS, INC., 2002) aos dados importantes do componente em execução. Cada componente possui um objeto *Factory* (GAMMA, HELM, *et al.*, 2005) que mantém as instâncias dos DAOs.
- *Controller* (Controlador, no inglês): englobam-se aqui classes utilitárias de controle dos componentes. Em geral suas funcionalidades são chamadas durante o processamento das requisições ou de seus objetos de resposta.

- Comunicação:
  - *Receiver* (Receptador, no inglês): engloba as classes que implementam o protocolo *ReceiverIF*. Essas classes recebem as requisições pela camada de comunicação e as traduzem em objetos que implementam o protocolo *IRequestTO*. Esses objetos definem parâmetros de requisições, logo cada implementação de *IRequestTO* é interpretada por uma implementação diferente de *RequesterIF*.
  - *Sender* (Remetente, no inglês): interpretam as respostas recebidas da camada de negócios. Cada implementação diferente de *IResponseTO* é interpretada por uma implementação especializada do protocolo *SenderIF*.
- UI:
  - *Async* (forma curta para assíncrono, no inglês): define a UI de forma assíncrona, na forma de interfaces gráficas com o usuário.
  - *Sync* (forma curta para síncrono, no inglês): define a UI de forma síncrona, através de comandos em uma interface de linha de comandos.

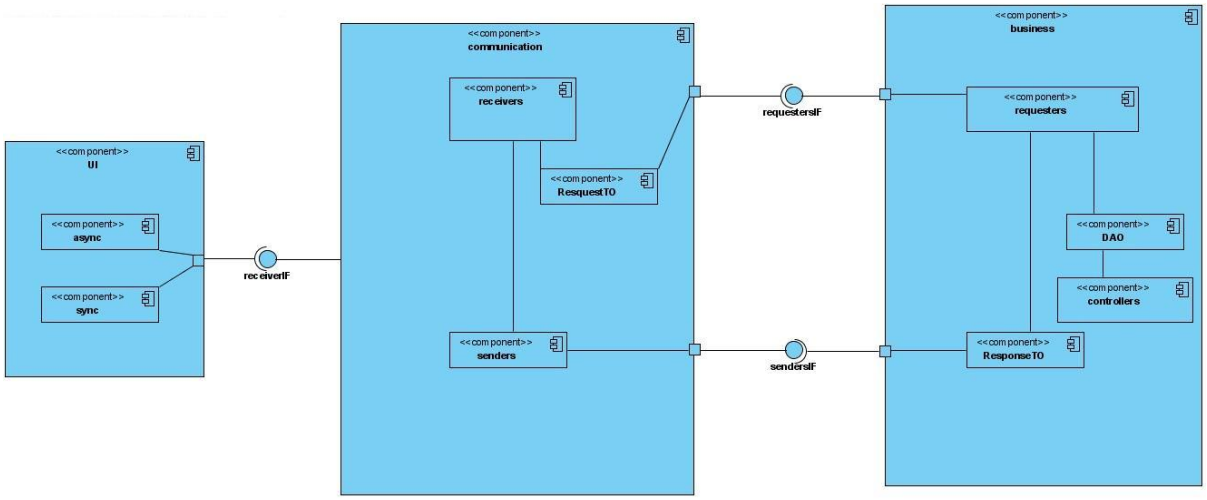

Figura 4. Diagrama de componentes internos de cada componente OurGrid (LABORATÓRIO DE SISTEMAS DISTRIBUÍDOS - UFCG, 2009)

A Figura 5 ilustra a seqüência de comunicação entre os componentes. Pode-se notar na ilustração o uso de classes *RequestControl* e *ResponseControl*; essas classes abstratas e seus subtipos (para cada componente) são as responsáveis pela definição de qual requisição e resposta vai para cada *Requester* e *Sender*, respectivamente. Toda comunicação é iniciada por um componente remoto que solicita execução de um método remoto (1). Essa solicitação é capturada por um *Receiver* em um *RequestTO* (2), executada e interpretada pelo *RequestControl* (3, 4 e 5). Várias respostas podem ser geradas nesse processamento (6 e 7), as quais são executadas pelo *ResponseControl* (8) e respondidas através de um *Sender.*

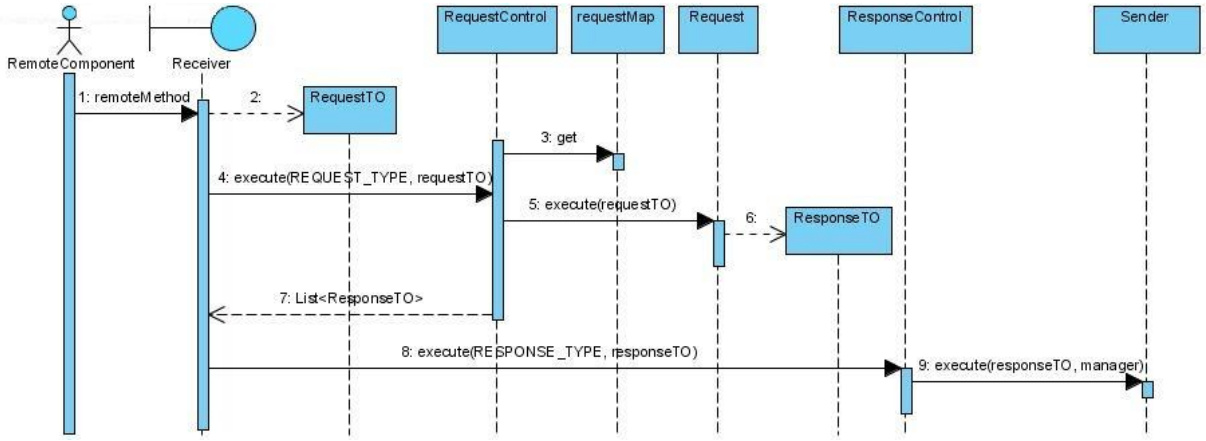

Figura 5. Seqüência de comunicação entre componentes para resposta a uma chamada de método remoto e sua requisição. (LABORATÓRIO DE SISTEMAS DISTRIBUÍDOS - UFCG, 2009)

2.3.2 Detecção de defeitos

A detecção de defeitos no OurGrid utiliza uma estratégia tradicional de detecção baseada em *heartbeat* em um modelo *Pull*. Um componente monitor *p* requisita uma mensagem de *heartbeat* periodicamente a cada componente monitorado *q*, de forma a certificar-se de que este permanece em funcionamento de forma correta. Se, durante um período de tempo T_{to}, *p* não recebe uma mensagem de resposta de *q*, *q* é considerado falho.

O encargo do monitoramento e detecção é atribuído à plataforma de comunicação utilizada pelo OurGrid. Quando um componente é informado que outro que ele monitora falhou, uma série de ações específica por função dos componentes é disparada. Essas reações incluem a troca de mensagens com outros componentes e em geral a reorganização da grade.

Capítulo 3

Reação a Suspeitas Incorretas

Como descrito na Seção 2.1, detectores de defeitos têm o propósito de prover uma visão consistente sobre a situação dos componentes do sistema. Também é tomado como certo que suspeitas incorretas sempre ocorrerão com a utilização de detectores de defeitos baseados em limites de tempo de resposta (SAMPAIO, BRASILEIRO, *et al.*, 2003). A partir disso, a razão deste trabalho foi motivado pela diminuição do retrabalho causado por suspeitas incorretas, sem sobrecarregar os recursos do sistema para fazê-lo. Para isso, um novo serviço foi projetado e implementado, o Serviço de Detecção de Suspeitas Incorretas (SDSI).

O trabalho se baseia no comportamento do OurGrid frente a suspeitas incorretas. A estabilidade da grade garante que, mesmo sob o efeito de suspeitas incorretas, o recebimento e posterior processamento de novos trabalhos permanecem intactos. As suspeitas incorretas abordadas nessa avaliação limitam-se a suspeitas quanto ao funcionamento do componente *Worker* que executa um trabalho para um componente *Broker*. Nesses casos, no fluxo de funcionamento normal do OurGrid, o *Broker* encerra a execução do trabalho, libera o *Worker*, e todo o trabalho executado até então é ignorado. A requisição feita ao *Peer* permanecerá como pendente, e logo este repassará novamente algum *Worker* para execução, o que acarreta em retrabalho. A abordagem utilizada pelo serviço proposto objetiva eliminar o encargo do retrabalho.

Para poder elaborar o novo serviço, foi necessário estudo aprofundado da plataforma de *middleware* de grade OurGrid, e a partir disso isolar decisões de projeto quanto às alterações necessárias ao tratamento das suspeitas incorretas. Essas alterações foram, então, implementadas na forma do SDSI, como um serviço de detecção de defeitos na própria detecção de defeitos do OurGrid e recuperação por retrocesso nos estados do componente *Worker*. As decisões originais e mudanças planejadas são descritas a seguir.

3.1 Decisões de projeto originais

Originalmente, no OurGrid, algumas decisões de projeto foram tomadas quanto ao comportamento dos componentes da grade ante a suspeita de uma falha. Devido à falta de planejamento sobre falso-positivos do detector de defeitos, o comportamento da grade ante as suspeitas incorretas não se diferencia das corretas. Segue-se um padrão de desconexão, através das decisões:

- O *Broker*, percebendo incorretamente a falha, envia requisição do tipo *DISPOSE_WORKER* para seu *Peer*, apaga configurações da execução, e aguarda novo *Worker* para o trabalho.
- O *Peer*, ao receber uma requisição de tipo *DISPOSE_*WORKER, libera o *Worker* que havia alocado para o *Broker*, envia a requisição de tipo *STOP_WORKING* a esse, e seleciona nova requisição de trabalho para alocar seu *Worker*. O processo de seleção utiliza uma heurística que pesa a compatibilidade da especificação da requisição com as especificações do *Worker*, além do tempo de espera das requisições.
- O *Worker*, ao receber uma requisição de tipo *STOP_WORKING*, cancela o processo do trabalho em execução e apaga toda configuração de variáveis da execução, incluindo identificadores do *Broker*, da requisição de trabalho e do processo no sistema. Seu estado volta a *IDLE*.

As decisões tomadas colaboram para diminuir a carga de processamento na máquina utilizada para o *Worker*, para a distribuição justa de requisições de trabalho, e para rápida reconfiguração do sistema em caso de nova execução. Entretanto, o estado da execução penalizada pela suspeita incorreta é perdido completamente, assim como todo o trabalho executado até então.

A partir das decisões observadas acima, novas decisões para a plataforma com suporte a suspeitas incorretas foram formuladas.

3.2 Processo de detecção do SDSI

Como mencionado, durante o evento de uma suspeita, uma série de passos específicos identificam que ocorreu uma suspeita de falha por parte do *Broker*

quanto a um *Worker*. Inicialmente, o componente *Broker* envia uma requisição *DISPOSE_WORKER* ao componente *Peer*. Este, por sua vez, envia uma requisição *STOP_WORKING* ao componente *Worker*, através da qual este é comunicado de que deve cancelar o processamento por completo. A partir daí, o *Worker* torna-se novamente propício a executar quaisquer novas requisições com as quais tenha compatibilidade de especificação.

Uma seqüência de passos seguintes caracteriza a suspeita incorreta. Normalmente, em casos onde não houve uma falha e a requisição *DISPOSE_WORKER* foi chamada, a requisição do *Broker* não continua na fila de execuções a serem tratadas pelo *Peer*. Entretanto, no evento de uma suspeita incorreta, o pedido do *Broker* continua a esperar resolução. Caso o *Peer* escolha por indicar para a execução o mesmo *Worker* que sofreu a suspeita, este acabaria por receber e executar a mesma requisição. Esta poderia ser identificada como igual à cancelada por possuir identificador de requisição, identificador de *Broker* e comando a ser executado, iguais à anterior.

A partir disso, tratamento pode ser concentrado em impedir o retrabalho. Ou seja, o *Worker* deve executar o trabalho submetido a ele diretamente, desde a primeira vez em que é solicitado, sem parar, até a segunda vez em que é solicitado. Para facilitar isso, o *Peer* também deve priorizar na seleção de *Workers* para encaminhar ao *Broker* aquele do qual ele originalmente suspeitou. Finalmente, após recebimento e execução da nova requisição, idêntica à anterior, o resultado será gerado sem necessidade de retrabalho.

3.3 Decisões de projeto para o tratamento das suspeitas

As seguintes decisões, baseadas nas decisões originais listadas anteriormente, visam possibilitar o tratamento de suspeitas incorretas pelo SDSI. Para isso, o estado da execução do *Worker* deve ser armazenado e processado adequadamente. Assim, para tratar da suspeita errônea como explicado na Seção 3.2, foram tomadas algumas decisões:

- No momento de recebimento da mensagem *DISPOSE_WORKER*, o serviço registra em uma estrutura de dados uma relação entre identificador da requisição de trabalho do *Broker*, identificador público do *Broker* no subsistema de comunicação (sua chave pública), e identificador público do *Worker* no sistema de comunicação. Assim, com apenas três variáveis por suspeita, a um custo pequeno de memória, pode-se rastrear se uma requisição ao *Peer* já foi suspeita, funcionalidade inexistente originalmente no OurGrid;
- Durante a execução do algoritmo de busca de requisições para redistribuir o *Worker*, agora ocioso, o *Peer* verifica as requisições de trabalho compatíveis. Caso elas contenham a requisição anterior (verificada no serviço), que foi ordenada a parar, assim como igualdade de *Broker* solicitador e *Worker* atendente, esta será a selecionada para o envio do *Worker.*
- Ao receber a requisição de *STOP_WORKING*, o serviço armazena as informações relativas ao processo que estava em execução: o identificador público do *Broker*; o identificador da requisição que estava em execução; e o identificador do processo em execução no sistema operacional – atualmente implementado no OurGrid como um subtipo de *Integer*, mas podendo caracterizar-se como qualquer implementação de um protocolo *ExecutionHandler*. Além disso, o componente foi alterado para não mais solicitar o cancelamento do processo em execução no sistema operacional.
- Quando o *Worker* é alocado para um novo *Broker*, e tem a requisição *REMOTE_EXECUTE_PROCESSOR* recebida e em processamento para execução do comando, o serviço verifica a originalidade da requisição. Caso o identificador da requisição de processamento, o identificador do *Broker* e o comando coincidam com os registrados pelo sistema no momento da suspeita e interrupção, nada é iniciado. O serviço se encarrega de, através do uso do identificador do processo no sistema operacional, recuperar informações do estado do processo, que serão informadas ao *Broker* quando do término de processamento. Caso contrário, é enviado comando para cancelar o

processo em execução no sistema operacional, e os passos comuns de iniciação de processo do OurGrid são seguidos. Desse modo, por não reiniciar o processamento, evita-se o retrabalho, com custos pequenos de processamento. O processo original ainda é cancelado caso não seja o requerido, e nenhuma mensagem extra é enviada para permitir o tratamento. Ademais, os *Workers* modificados mantém total compatibilidade com a arquitetura atual do OurGrid, podendo ser implantados em meio a uma comunidade existente.

As decisões implementadas deram forma ao SDSI. Percebe-se um padrão de detecção e recuperação de erros no comportamento do serviço. O SDSI pode ser comparado, então, a um detector de defeitos do próprio detector de defeitos da grade. Seu diagnóstico é feito a partir da análise das mensagens e estado do sistema, e sua recuperação assemelha-se a uma recuperação por retrocesso. Felizmente, dado ao isolamento do controle de início e término de processos no *Worker*, a recuperação pode ser feita sem tanta complexidade, visto que não é necessário notificar nenhum outro componente na grade do tratamento.

Capítulo 4

Custos da Abordagem Proposta

De acordo com as metas da implementação do projeto, o novo serviço proposto e implementado deve ter baixo impacto computacional nos componentes onde está inserido. Desse modo, a seguir é feita uma breve análise discursiva do custo do SDSI em termos de recursos computacionais.

4.1 Configuração de ambiente

Como proposto, o foco principal do serviço adicionado ao OurGrid é a diminuição de retrabalho. Assim, testes foram executados com a simulação da suspeita incorreta de um *Broker* quanto ao *Worker* que executava um *job* para o mesmo.

A configuração para teste utilizada está ilustrada na Figura 6. Todos os testes foram executados através da simulação de uma rede interna em um ambiente virtualizado. A execução dos experimentos ocorreu entre máquinas virtuais da plataforma *Oracle VirtualBox* (ORACLE CORPORATION, 2010), versão *3.2.10 rc66523*. Quatro máquinas virtuais desta plataforma foram utilizadas, das quais três executavam os componentes da grade. A máquina virtual restante ficou encarregada de executar um emulador de WAN (rede de larga escala, do inglês *Wide Area Network*), denominado *WANem* (TATA CONSULTANCY SERVICES, 2011), versão 2.3, utilizado no processo de produção do falso-positivo para teste. Cada máquina de componentes possuía instalado o sistema operacional *Xubuntu*, versão 10.10, e kernel *Linux 2.6.35-22-generic*. A máquina virtual do emulador de rede, por sua vez, executa o próprio emulador, que é embutido em uma versão modificada da distribuição Linux *Knoppix*, versão *5.3*, com versão de kernel *Linux 2.6.24-4*. Devido aos componentes da grade serem executados na forma de aplicações Java, as máquinas virtuais tiveram nelas instalada a versão *1.6.0_22-b04* da máquina virtual Java produzida pela Oracle. Além dessas configurações, só foram inseridos nas máquinas virtuais os próprios componentes da grade.

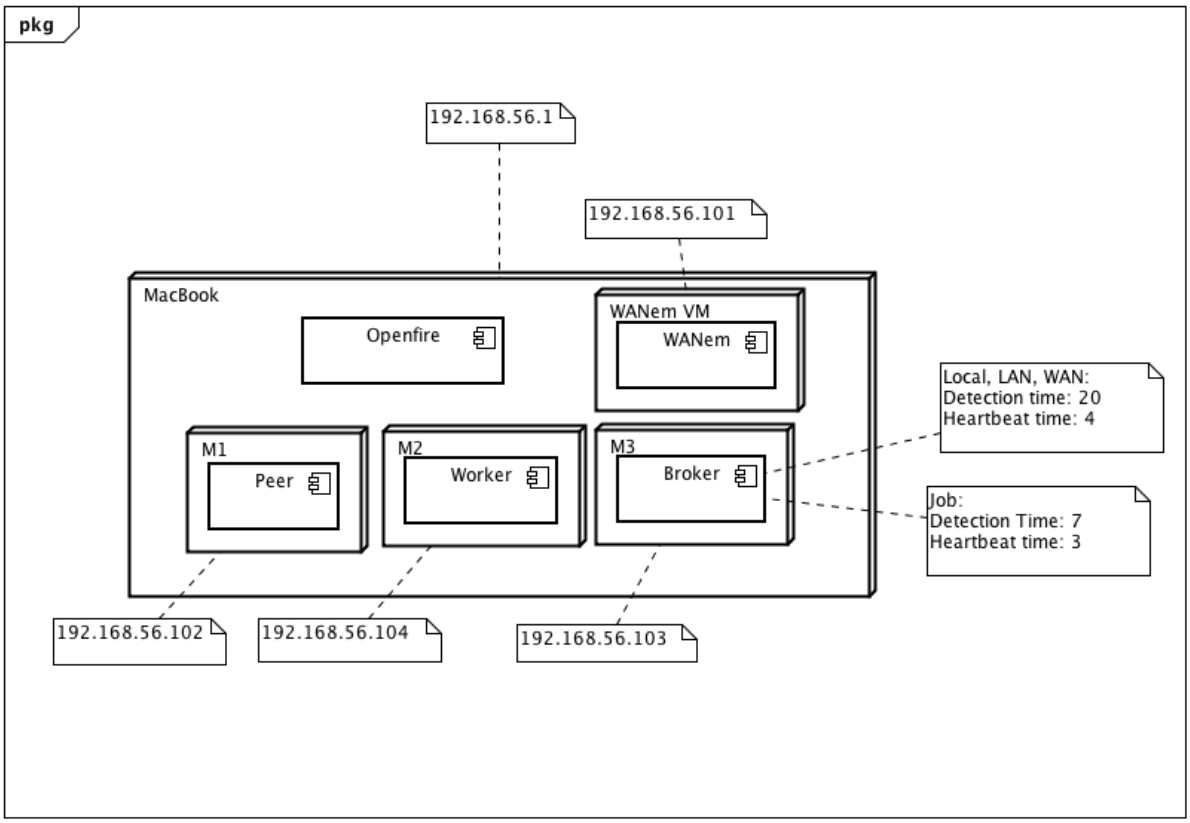

Figura 6. Ambiente de teste da implementação do OurGrid com o SDSI.

Os componentes da grade foram distribuídos entre as máquinas virtuais de componentes, com um *Broker* em uma máquina, em outra um *Peer*, e um *Worker* na restante. As interfaces de redes de todas as máquinas virtuais foram configuradas como interfaces virtuais ligadas à placa de rede da máquina física, cada interface virtual com um IP da rede local simulada.

Outro componente crucial para a grade é o servidor do protocolo XMPP, que, como mencionado na Seção 2.3, representa o substrato de comunicação. Foi utilizado para esse propósito o servidor *Openfire* versão 3.6.4 (JIVE SOFTWARE, 2011).

A máquina física utilizada, encarregada de executar tanto a plataforma de virtualização quanto o servidor de comunicação, foi um *MacBook* com processador *Intel Core 2 Duo* de 2 GHz, memória principal de 2.5 GB de *SDRAM DDR2* 667 MHz, e cujo sistema operacional era *Mac OS X 10.6.7*. Sua configuração na rede

tornou-se a de um *gateway* entre as máquinas virtuais, através do qual elas também se comunicavam por utilizarem o servidor XMPP para troca de mensagens.

Para provocar a suspeita incorreta, era necessário forçar a rede a produzir um atraso que induzisse o OurGrid a detectar uma Falha de Temporização. Para isso, o componente *Broker* teve configurado, relativo à comunicação com os *Workers* que o servem, 7 segundos como limite de espera de *heartbeats*, e intervalo de requisição de *heartbeats* de 3 segundos. Todos os componentes, em sua comunicação normal na grade, foram configurados com limite de espera de *heartbeats* em 20 segundos, e intervalo entre requisição de *heartbeats* de 4 segundos.

Foi possível simular a falha através de um ligeiro atraso em relação ao limite de espera configurado para o *job*, porém sem gerar reconhecimento de falha quanto a outros componentes. O emulador de WAN permite causar um atraso configurado em pacotes roteados através dele. Assim, o WANem foi configurado para gerar atraso de 8 segundos[1], continuamente. Um falso-positivo era gerado, quando desejado no teste, a partir do redirecionamento da comunicação do *gateway* para a máquina virtual do *Broker* por um período de apenas 8 segundos. Os pacotes transmitidos do substrato de comunicação XMPP para o *Broker* eram, então, atrasados em 8 segundos, o que esgotava o limite de espera de uma resposta do *Worker*. A configuração do roteamento de pacotes entre as máquinas foi feita com a utilização do *script* de linha de comando *Bash* mostrado na Figura 7.

```
#!/bin/bash

sudo route add -net $1 $2 255.255.255.255
sudo route add -host $1 $2
sleep $3
sudo route delete -host $1
sudo route delete -net $1
```

Figura 7. *Script* para configuração do roteamento.

[1] O tamanho pequeno dos intervalos utilizados foi determinado por uma característica do emulador: o WANem só permite emulação de atrasos de, no máximo, 10 segundos. Em geral esses intervalos no OurGrid são configurados para valores maiores, de 30 segundos à alguns minutos. Isso não impede, entretanto, de falso-positivos ocorrerem também nesses casos.

Os trabalhos executados para testes eram de curta duração, porém o suficiente para comprovar a eficácia do SDSI. Ambos possuem duração aproximada de 10 minutos para execução. O primeiro consistia em simplesmente chamar o comando *sleep*, de sistemas *Unix*, para fazer com que o sistema operacional aguardasse 10 minutos. O segundo envolvia o cálculo de todos os números primos até determinado número, pela aplicação do Crivo de Eratóstenes.

4.2 Redução de retrabalho

Como proposto, o foco principal do serviço adicionado ao OurGrid é a diminuição de retrabalho. O comportamento tradicional do OurGrid é insatisfatório quando uma execução que possa durar 10 dura 20. Com isso em consideração, o serviço foi testado.

Os testes obtiveram resultados satisfatórios. Pela versão original do OurGrid, a duração das requisições, tendo esperado 5 minutos para forçar a suspeita incorreta, era em torno de 15 minutos – a soma do tempo da suposta falha com a nova execução posterior. Ao executar os testes com a versão adaptada com o SDSI, as requisições de trabalho submetidas pelo *Broker* permaneceram com a duração original dos trabalhos, em torno de 10 minutos.

Dado tratar-se de uma plataforma de *middleware* de grade com grande utilização científica (LABORATÓRIO DE SISTEMAS DISTRIBUÍDOS - UFCG, 2011), podem-se considerar aplicações que executem em grade durante horas, mesmo dias. A margem de ganho com a redução de retrabalho é, portanto, indefinida, dependente apenas da aplicação, do momento da suspeita incorreta, e do número de suspeitas.

4.3 Consumo de memória

Um dos requisitos deste trabalho é a necessidade de tratar suspeitas incorretas sem um grande custo adicional de recursos computacionais. O consumo de memória, em especial, tende a aumentar com a inserção de um novo serviço, mesmo que apenas pelo carregamento de um conjunto de classes novo pela JVM (máquina virtual Java – do inglês *Java Virtual Machine*).

O consumo de memória com o novo serviço teve algum acréscimo, visto que, além de classes carregadas pela JVM, algumas variáveis da execução tiveram de ser mantidas. Porém, a modificação em termos de componentes no sistema foi pequena. Constituiu-se apenas do armazenamento de três variáveis do tipo *String*, uma variável do tipo *long*, e um objeto *ExecutionHandler*, o qual atualmente é implementado como um *Integer*. Ao considerar-se o tamanho total do sistema, a alteração em consumo de memória foi desprezível.

4.4 Uso de CPU

Quanto ao requisito não-funcional de baixo consumo de recursos computacionais, o SDSI buscou não alterar significativamente o uso de CPU dos componentes. Através do estudo cuidadoso do funcionamento da grade, as alterações necessárias puderam ser feitas *in loco*, sem necessidade de uma grande estrutura externa ou de outras técnicas que viessem a aumentar a complexidade computacional da grade. A alteração limitou-se a pequenas mudanças no controle de fluxo do programa, em partes isoladas – graças à própria modularização do funcionamento do OurGrid, em *Requests* e *Responses*. Logo, não houve motivo para aumento de uso de CPU.

Entretanto, um ponto do tratamento de suspeitas incorretas teve um impacto controverso no uso de CPU. A mudança necessária para se recuperar um processo envolve não cancelar o processo do trabalho no sistema operacional. Durante o tempo em que o *Worker* é desalocado do *Broker*, até ser alocado novamente, o processo continua em execução, o que constitui um aumento de custo computacional, embora não diretamente do OurGrid. O custo é maior do que do *Worker* ocioso originalmente no OurGrid, porém não maior do que o da alocação normalmente. Entretanto, quando se trata de uma suspeita incorreta, o processo não cancelado se torna um ganho em uso de recursos. Primeiro, a redução do retrabalho apresenta ganho indireto no consumo pela máquina em geral – em oposição à perda de manter o processo ativo após o cancelamento. Finalmente, evitar criar um novo processo também elimina o custo inerente à criação de processos, especialmente no caso de um novo processo desnecessário. Em casos nos quais o *Worker* realmente

deveria ter cancelado o processo, ele ainda o fará antes da próxima alocação, o que retorna ao custo normal da grade.

Com isso em consideração, é possível considerar que o tratamento de suspeitas incorretas tem um impacto pequeno no uso de CPU, com inclusive uma possível melhoria quanto ao comportamento padrão.

4.5 Envio de mensagens

A forma utilizada para detecção e tratamento de suspeitas incorretas leva em consideração o número já bastante grande de mensagens enviadas pelo OurGrid. A todo o momento, componentes requisitam mensagens de *heartbeat* e recebem informação de volta; enviam requisições para iniciar execuções em outros componentes; respondem sobre o estado de execução; entre outras diversas operações que são executadas de forma distribuída.

Outras abordagens consideradas para a solução envolveriam a construção de módulos cliente-servidor para troca de informações entre componentes. A escolha dessa abordagem facilitaria a notificação geral do ocorrido de uma suspeita incorreta. Entretanto, para isso, incorrer-se-ia em algum custo adicional de trocas de mensagem, principalmente se a decisão sobre o tratamento também envolvesse mais trocas de mensagem.

A abordagem utilizada, com análise das requisições utilizadas pelos componentes do OurGrid, permitiu detecção e tratamento das suspeitas incorretas através da própria alteração da máquina de estados da plataforma. Assim, nenhuma mensagem adicional precisou ser enviada para o tratamento. Todas as mensagens enviadas com o tratamento de suspeitas incorretas são as mesmas que seriam enviadas normalmente pelo OurGrid sem o tratamento. Adicionalmente, os componentes alterados com o SDSI mantêm total compatibilidade com a plataforma original, ainda com capacidade de tratar suspeitas incorretas.

Capítulo 5

Conclusão e Trabalhos Futuros

O trabalho aqui apresentado descreveu o projeto e implementação de um serviço de detecção de suspeitas incorretas, denominado SDSI, na plataforma de *middleware* de grades computacionais OurGrid. A resolução do problema de suspeitas incorretas tem impacto em empreendimentos que utilizam sistemas de computação distribuída, como grades computacionais ou mesmo sistemas de computação em nuvem. A perda de trabalhos por conta de suspeitas incorretas nesses sistemas pode representar grande prejuízo para o fornecedor de recursos, que precisa processar mais de uma vez a mesma tarefa.

A solução foi avaliada em relação à interferência desta no uso de recursos computacionais da grade, e na quantidade de retrabalho que pode ser evitada ao se tratar das suspeitas incorretas. Em geral, não houve alteração significativa no uso de recursos computacionais, exceto pelo fato de ser evitado todo processamento do retrabalho – o qual pode até duplicar um trabalho a ser executado, para cada suspeita incorreta.

5.1 Trabalhos futuros

O escopo da solução aqui proposta e implementada teve várias restrições devido ao prazo de conclusão. Com isso, diversas melhorias estão planejadas como passos futuros deste mesmo projeto.

A forma como a detecção e a recuperação atuam é completamente alheia ao conhecimento do componente *Broker*, que recebe o resultado mais eficazmente, porém sem uma interpretação mais profunda da falha que acreditou ocorrer. Uma das possíveis alterações futuras inclui a comunicação ao *Broker* de que ocorreu uma suspeita incorreta. A partir disso, pode ser vantajoso adaptar o limite de espera configurado nele para trabalhos em execução. Talvez também haja situações em que o tratamento da suspeita não seja desejado; assim, espera-se adicionar ao formato de especificação de trabalhos uma cláusula que deixe explícito a opção.

Desse modo, o tratamento ficaria condicionado à escolha do usuário que escreve o trabalho.

Extensões desse trabalho também incluem o tratamento de suspeitas incorretas de outros componentes. Acredita-se ser necessário o tratamento no serviço para o total alcance de suas propostas, mas não foi possível atendê-lo no prazo deste trabalho, logo não puderam fazer parte de seu escopo.

Outro ponto de melhoria do trabalho inclui a integração deste projeto com o projeto Hamster (FARIAS, SOARES-NETO e CASTOR, 2010) para detecção e tratamento de defeitos em grades computacionais. O trabalho do SDSI será complementar ao do Hamster, e espera-se que ambos juntos possam ser anexados à base de dados do OurGrid. Assim como o Hamster, o SDSI não é um serviço que pretende ser apenas usado no OurGrid. Portanto, parte do esforço dessa integração incluirá a integração do SDSI a outras plataformas de *middleware* para grades computacionais, como o InteGrade e o Globus. Dado o tamanho e abrangência de uso da plataforma Globus, certamente o uso do SDSI nele traria grandes recompensas em termos de visibilidade e disseminação das idéias deste trabalho.

Ademais, é esperado mais avaliações, em trabalhos futuros, do impacto do SDSI no consumo de recursos da grade e do sistema operacional que hospeda seus componentes. As avaliações feitas aqui devem ser validadas com mais execuções empíricas do serviço, inclusive no contexto do serviço do OurGrid implantado em escala global.

Bibliografia

ANTONOPOULOS, N.; GILLAM, L. **Cloud Computing:** Principles, Systems and Applications. [S.l.]: Springer, 2010.

CAMARGO, R. Y. D. et al. **Grid:** An Architectural Pattern. The 11th Conference on Pattern Languages of Programs (PLoP'2004). Monticello, Illinois, EUA: [s.n.]. 2004.

CHANDRA, T. D.; TOUEG, S. Unreliable failure detectors for reliable distributed systems. **Journal of the ACM**, Março 1996. 225-267.

CIRNE, W. et al. Labs of the World, Unite! **Journal of Grid Computing**, 2006. 225-246.

FARIAS, R.; SOARES-NETO, F.; CASTOR, F. **Hamster:** making grid middleware fault-tolerant. Proceedings of the ACM international conference companion on Systems, Programming Languages and Applications - Software for Humanity. Reno/Tahoe, Nevada, USA: ACM. 2010. p. 251-252.

FOSTER, I.; KESSELMAN, C. Globus: A Metacomputing Infrastructure Toolkit. **International Journal of Supercomputer Applications**, 1996. 115-128.

GAMMA, E. et al. **Padrões de Projeto:** soluções reutilizáveis de software orientado a objetos. 1ª Edição. ed. [S.l.]: Bookman, 2005.

GOLDCHLEGER, A. et al. InteGrade object-oriented Grid middleware leveraging the idle computing power of desktop machines. **Concurrency and Computation : Practice & Experience**, April 2004. 449-459.

IETF. RFC 3920 - Extensible Messaging and Presence Protocol (XMPP): Core, 2004. Disponivel em: <http://tools.ietf.org/html/rfc3920>. Acesso em: 06 Junho 2011.

JANSCH-PORTO, I. E. S.; WEBER, T. S. **Recuperação em Sistemas Distribuídos.** XVI Jornada de Atualização em Informática, XVII Congresso da SBC. Brasília: [s.n.]. 1997. p. 263-310.

JIVE SOFTWARE. OpenFire. **Ignite Realtime**, 2011. Disponivel em: <http://www.igniterealtime.org/projects/openfire/>. Acesso em: 06 Junho 2011.

KESSELMAN, C.; FOSTER, I. **The Grid:** Blueprint for a New Computing Infrastructure. [S.l.]: Morgan Kaufmann Publishers, 1998.

LABORATÓRIO DE SISTEMAS DISTRIBUÍDOS - UFCG. Component Internal Architecture. **OurGrid Middleware**, 2009. Disponivel em: <http://redmine.lsd.ufcg.edu.br/wiki/ourgrid/Development-architecture>. Acesso em: 06 Junho 2011.

LABORATÓRIO DE SISTEMAS DISTRIBUÍDOS - UFCG. Success Stories. **OurGrid**, 2011. Disponivel em: <http://www.ourgrid.org/index.php?option=com_content&view=article&id=19&Itemid=27&lang=en>. Acesso em: 06 Junho 2011.

LAPRIE, J. C. **Dependable computing and fault-tolerance:** concepts and terminology. Proceedings of the Annual International Symposium on Fault Tolerant Computing. Ann Arbor: IEEE. 1985. p. 2-11.

LEE, P. A.; ANDERSON, T. **Fault Tolerance:** Principles and Practice. 2. ed. New York: Springer-Verlag, 1990.

ORACLE CORPORATION. VirtualBox. **VirtualBox**, 2010. Disponivel em: <http://www.virtualbox.org>. Acesso em: 19 Dezembro 2010.

PRADHAN, D. K. **Fault-tolerant computer system design**. Upper Saddle River, NJ, USA: Prentice Hall Inc., 1996.

SAMPAIO, L. M. R. et al. **How Bad Are Wrong Suspicions? Towards Adaptive Distributed Protocols.** International Conference on Dependable Systems and Networks. Los Alamitos, CA, USA: IEEE Computer Society. 2003.

SUN MICROSYSTEMS, INC. Core J2EE Patterns - Data Access Object. **Sun Developer Network**, 2002. Disponivel em: <http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html>. Acesso em: 06 Junho 2011.

TANENBAUM, A. S.; STEEN, M. V. **Sistemas Distribuídos – princípios e paradigmas.** São Paulo: Prentice Hall, 2007.

TATA CONSULTANCY SERVICES. WANem - The Wide Area Network emulator. **WANem**, 2011. Disponivel em: <http://wanem.sourceforge.net/>. Acesso em: 06 Junho 2011.

WEBER, T. S. **Um roteiro para exploração dos Conceitos Básicos de Tolerância a Falhas - Material Didático do Curso de Especialização em Redes e Sistemas Distribuídos.** Instituto de Informática - UFRGS. Porto Alegre - RS. 2002.