UNIVERSIDADE FEDERAL DO MARANHÃO
CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA DE ELETRICIDADE

Antonio Eduardo Bernardes Viana

Uma Abordagem Autonômica para Tolerância a Falhas na Execução de Aplicações em Desktop Grids

São Luís
2011

Antonio Eduardo Bernardes Viana

Uma Abordagem Autonômica para Tolerância a Falhas na Execução de Aplicações em Desktop Grids

Dissertação apresentada ao Programa de Pós-Graduação em Engenharia de Eletricidade da Universidade Federal do Maranhão como requisito parcial para a obtenção do grau de MESTRE em Engenharia de Eletricidade.

Orientador: Francisco José da Silva e Silva

Doutor em Ciência da Computação – UFMA

São Luís

2011

Viana, Antonio Eduardo Bernardes.

Uma abordagem autonômica para tolerância a falhas na execução de aplicações em *desktop grids* /Antonio Eduardo Bernardes Viana. – São Luís, 2011.

133 f.

Impresso por computador (fotocópia).

Orientador: Francisco José da Silva e Silva.

Dissertação (Mestrado) – Universidade Federal do Maranhão, Programa de Pós-Graduação em Engenharia de Eletricidade, 2011.

1. Tolerância a falhas. 2. Computação autonômica. 3. Grades computacionais. 4. Moagem. I. Título.

CDU 004.052.3

UMA ABORDAGEM AUTONÔMICA PARA TOLERÂNCIA A FALHAS NA EXECUÇÃO DE APLICAÇÕES EM DESKTOP GRIDS

Antonio Eduardo Bernardes Viana

Dissertação aprovada em 05de setembro de 2011.

Prof. Francisco José da Silva e Silva, Dr.
(Orientador)

Prof. Alfredo Goldman Vel Lejbman, Dr.
(Membro da Banca Examinadora)

Prof. Samyr Beliche Vale, Dr.
(Membro da Banca Examinadora)

Aos meus pais, irmãos e minha família.

Resumo

Grades de computadores são caracterizadas pelo alto dinamiso de seu ambiente de execução, alta heteroneidade de recursos e tarefas e por requererem grande escalabilidade. Essas características tornam tarefas como configuração, manutenção e recuperação da execução de aplicações em caso de falhas bastante desafiadoras e cada vez mais difícil de serem realizadas exclusivamente por agentes humanos.

A computação autonômica denota sistemas computacionais capazes de mudar seu comportamento dinamicamente em resposta a variações do ambiente de execução. Para isso, o software é geralmente organizado seguindo-se o modelo MAPE-K (*Monitoring, Analysis, Planning, Execution and Knowledge*), no qual gerentes autonômicos realizam as atividades de sensoriamento do ambiente de execução, análise de contexto, planejamento e execução de ações de reconfiguração dinâmica, compartilhando algum conhecimento sobre o sistema controlado.

Nesse trabalho apresentamos um mecanismo autonômico baseado no modelo MAPE-K para prover tolerância a falhas na execução de aplicações em grades de computadores capaz de monitorar o ambiente de execução e, a partir da avaliação dos dados coletados, decidir quais ações de recofiguração devem eventualmente serem aplicadas ao mecanismo de tolerância falhas para manter o sistema em equlíbrio com os objetivos de minimizar o tempo médio de conclusão das aplicações e prover alta taxa de sucesso na conclusão de suas tarefas. Este trabalho descreve ainda a avaliação de desempenho do mecanismo autonômico proposto, realizada através do uso técnicas de simulação e que levou em consideração diversos cenários típicos de ambientes de *desktop grids* oportunistas.

Palavras-chaves: Grades de Computadores, Tolerância a Falhas, Computação Autonômica.

Abstract

Computers grids are characterized by the high dynamism of its execution environment, resources and applications heterogeneity, and the requirement for high scalability. These features turn tasks such as configuration, maintenance and recovery of failed applications quite challenging and is becoming increasingly difficult to perform them only by human agents.

The autonomic computing paradigm denotes computer systems capable of changing their behavior dynamically in response to changes in the execution environment. For achieving this, the software is generally organized following the MAPE-K (Monitoring, Analysis, Planning, Execution and Knowledge) model, in which managers perform the execution environment sensing activities, context analysis, planning and execution of dynamic reconfiguration actions, based on shared knowledge about the controlled system.

In this work we present an autonomic mechanism based on the MAPE-K model to provide fault tolerance for applications running on computer grids, which is capable of monitoring the execution environment and, based on the evaluation of the collected data, to decide which reconfiguration actions must eventually be applied to the fault tolerance mechanism in order to keep the system in balance with the goals of minimizing the applications average completion time and to provide a high success rate in completing their tasks. This paper also describes the performance evaluation of the proposed autonomic mechanism, accomplished through the use of simulation techniques that took into account several opportunistic desktop grids typical environmental scenarios.

keywords: Grid Computing, Fault Tolerance, Autonomic Computing.

Agradecimentos

Agradeço a Deus em primeiro lugar e à minha família.

Agradeço ao meu orientador, Prof. Dr. Francisco, que me conduziu na elaboração deste trabalho.

Agradeço à equipe do Laboratório de Sistemas Distribuídos da UFMA, com os quais compartilhamos conhecimentos na área de pesquisa, especialmente a equipe de grades autonômicas e a Berto que ralou vários fins de semanas junto conosco para alcançarmos este trabalho.

Agradeço à UFMA e ao Programa de Pós-Graduação de Engenharia de Eletricidade, especialmente ao Alcides que sempre nos motivou a continuar nossa pesquisa.

Agradeço ao Núcleo de Tecnologia da Informação que permitiu que começássemos e continuássemos até a conclusão essa empreitada, especialmente ao Nélio e ao Osvaldo, que pacientemente seguraram toda a barra quando eu tive que me dedicar mais do que poderia nesta obra.

Ainda no NTI, Agradeço a equipe do SIPAC: Marcondes, DJefferson, Jone e William que também fez parte dessa luta.

Agradeço a vários amigos que sempre me incentivaram continuar, e me serviram de inspiração, como Gilberto e Geraldo.

Agradeço ao pessoal da PROGF da UFMA que compreendeu e me apoiou mesmo na minha ausência.

"Não é o mais forte que sobrevive, nem o mais inteligente, mas o que melhor se adapta às mudanças."

Charles Darwin

Sumário

Lista de Figuras

Lista de Tabelas

1 Introdução

É crescente o uso de aplicações computacionais em diversas áreas do conhecimento, incluindo o de aplicações de alto desempenho, que exigem uma capacidade de processamento superior à oferecida pela maioria dos computadores utilizados atualmente na indústria, comércio e academia. Como exemplos desse tipo de aplicação, podemos citar o processamento de sequenciamento de genes (genoma), mineração de dados (para o mercado financeiro), enovelamento de proteínas (indústria farmacêutica), análise de sinais (SETI) e previsão de tempo (INPE/CPTEC), entre outros.

A abordagem tradicional para a execução de aplicações que demandam alto desempenho computacional é a utilização de supercomputadores, cujo custo de aquisição e manutenção especializada é muito elevado. Uma outra abordagem de menor custo é a utilização de *clusters* de computadores pessoais, cujas máquinas são interligadas por uma rede de alta velocidade. Um exemplo bastante popular desta abordagem é o *cluster Beowulf*, baseado no sistema operacional Linux. Em um *cluster Beowulf* [3], um nó é definido como mestre, cuja atribuição é distribuir as tarefas que compõem a aplicação para execução nos demais nós e disponibilizar uma interface para os usuários do sistema. Estas máquinas são usualmente dedicadas para a execução de aplicações de alto desempenho e costumeiramente são subutilizadas, permanecendo uma parcela significativa de tempo ociosas [23].

Ao longo desta última década, a abordagem da computação em grade tem sido explorada para a execução de aplicações de alto desempenho. Uma grade de computadores é um sistema que coordena recursos distribuídos, como processadores, software, dados e periféricos conectados através de uma rede, com a finalidade de formar um sistema distribuído de larga escala que pode ser utilizado para realizar computações diversificadas. Estes recursos computacionais podem estar geograficamente dispersos e pertencerem a diversas organizações. Desta forma, uma questão fundamental em um sistema de computação em grade é que recursos de diferentes organizações podem ser reunidos para permitir a colaboração de um grupo

de pessoas ou instituições. Tal colaboração é realizada sob a forma de uma organização virtual.

O componente central de uma arquitetura de grade é o seu *middleware*. Ele é utilizado para esconder a natureza heterogênea e a complexidade decorrente da distribuição dos recursos que compõem a grade. O *middleware* disponibiliza aos usuários e aplicações uma visão homogênea do ambiente, provendo interfaces padronizadas para diversos serviços. O projeto InteGrade [1] [14] é uma iniciativa multi-institucional (USP, PUC-Rio, UFMS, UFG, UFMA) para o desenvolvimento de um *middleware* de grade capaz de usufruir do poder computacional de estações de trabalho que estejam ociosas, aproveitando seus ciclos para a execução de aplicações paralelas de alto desempenho. O InteGrade disponibiliza suporte para a execução de aplicações regulares, *bag-of-tasks* (BoT) e paralelas (MPI e BSP), gerenciamento de recursos distribuídos em larga escala, mecanismos de tolerância a falhas e segurança, além de um ambiente integrado de desenvolvimento.

Um aspecto importante no desenvolvimento de um *middleware* de grade é o tratamento adequado da dinamicidade do ambiente. A computação em grade tornou possível a execução de aplicações cuja composição e interação é altamente dinâmica [58]. Além disso, a própria infraestrutura da grade é heterogênea e dinâmica. Esse dinamismo de aplicações e infraestrutura da grade, sua alta escalabilidade e heterogeneidade tornam tarefas como configuração, manutenção e recuperação em caso de falhas altamente complexas para serem realizadas por agentes humanos. A computação autonômica denota sistemas computacionais capazes de mudar seu comportamento dinamicamente em resposta a variações do ambiente de execução de acordo com políticas e objetivos estabelecidos, de forma análoga ao comportamento auto-regulatório de sistemas biológicos. Para isso, o software é usualmente organizado seguindo-se o modelo MAPE-K (*Monitoring, Analysis, Planning, Execution and Knowledge loop*) , no qual gerentes autonômicos realizam as atividades de sensoriamento do ambiente de execução, análise de contexto, planejamento e execução de ações de reconfiguração dinâmica, compartilhando algum conhecimento sobre o sistema controlado [51][30].

[1] `http://www.integrade.org.br/`

Além de ser capaz de lidar com a dinamicidade do ambiente, um outro requisito fundamental para *middleware* de grades é o provimento de mecanismos de tolerância a falhas (TF), uma vez que os usuários que submetem suas aplicações esperam que elas concluam com sucesso. *Desktop grids* estão sujeitos a diversos tipos de falhas, podendo apresentar problemas físicos (no *hardware* do equipamento e do enlace da rede), erros lógicos (na aplicação e no protocolo de comunicação) e sofrer invasão (por usuário ou *software* maliciosos). Um outro tipo de falha é causado pela dinamicidade dos recursos da grade, pois estes são compartilhados e podem se tornar indisponíveis a qualquer momento, mesmo quando ainda estão realizando alguma computação para a grade. Além disso, as aplicações submetidas podem executar longas tarefas e levar vários dias, o que também aumenta a probabilidade de falhas [62][14].

A dinamicidade e heterogeneidade do ambiente podem elevar ou diminuir significativamente o número de recursos compartilhados. O ambiente também pode variar de acordo com a utilização ou a disponibilidade de seus recursos, bem como a quantidade, o tipo e o tamanho das aplicações a serem executadas, e isso pode refletir em diferentes escolhas em relação ao mecanismo mais apropriado para recuperação de falhas na execução de aplicações. Como é impossível prever antecipadamente o comportamento de ambientes de grades reais, argumentamos que a tomada de decisão sobre o mecanismo de tolerância a falhas mais adequando deve ser realizada dinamicamente em tempo de execução.

Nesse trabalho apresentamos um mecanismo autonômico de tolerância a falhas para execução de aplicações em grades de computadores capaz de monitorar o ambiente de uma grade computacional e selecionar a estratégia de tolerância a falhas para a execução de aplicações mais apropriada. Este trabalho tomou como base as heurísticas descritas por Chtepen et al. em [5], que propõe tolerância a falhas adaptativa para execução de aplicações em grades de computadores. A esse trabalho fizemos diversas modificações: incluímos o suporte a aplicações do tipo *bag-of-tasks*, que compreende a classe de aplicações mais utilizada em *desktop grids*; alteramos parâmetros utilizados pelas técnicas replicação e *checkpointing* a fim de proporcionar um melhor desempenho em ambientes de grades oportunistas; tratamos um efeito indesejável que causa repetidas reconfigurações na técnica de tolerância a falhas, o qual chamamos de efeito *ping-pong*; projetamos e implementamos uma arquitetura

baseada no modelo MAPE-K, um padrão arquitetural muito utilizado em sistemas autonômicos; e avaliamos o seu desempenho através de simulações em vários cenários.

Para a avaliação do mecanismo autonômico proposto, foi utilizado o simulador AGST (*Autonomic Grid Simulator Tool*), uma ferramenta de simulação que permite avaliar comportamento autonômico em *desktop grids* [17]. O AGST é extensão do OGST (*Opportunistic Grid Simulator Tool*), ambas desenvolvidas por pesquisadores da Universidade Federal do Maranhão [22]. O AGST oferece suporte a modelagem de arquiteturas de software baseadas no modelo para sistemas autonômicos MAPE-K. O mecanismo de tolerância a falhas autonômico proposto neste trabalho foi avaliado através de simulações utilizando-se vários cenários, ajustando seus parâmetros para uso eficiente em ambientes de *desktop grids* oportunistas. Nas avaliações de desempenho realizadas, utilizamos como métricas o número de aplicações finalizadas com sucesso e o tempo médio de conclusão dessas aplicações.

1.1 Objetivos

Esta dissertação de mestrado tem como objetivo geral o desenvolvimento de um mecanismo, baseado em técnicas de computação autonômica, capaz de fornecer tolerância a falhas à execução de aplicações em *desktop grids* de modo eficiente. Os objetivos específicos deste trabalho são:

- Elaboração de um mecanismo de tolerância a falhas autonômico para a execução de aplicações em ambientes de grades oportunistas;
- Definição de heurísticas para formar a base de conhecimento do mecanismo autonômico, a qual oferecerá as diretrizes para o processo de tomada de decisão;
- Projeto e implementação de um modelo para o mecanismo autonômico na ferramenta de simulação AGST, com o objetivo de avaliar o desempenho da abordagem proposta neste trabalho;
- Avaliação do mecanismo autonômico proposto, através da realização de simulações considerando diversos cenários típicos de ambientes de grades oportunistas.

1.2 Estrutura da Dissertação

Esta dissertação está organizada como segue: o Capítulo 2 apresenta os principais conceitos sobre a computação autonômica, uma das bases fundamentais para elaboração do nosso trabalho. No estudo da computação autonômica, apresentaremos o funcionamento do ciclo de gerenciamento autonômico e o modelo MAPE-K, que representa a arquitetura básica dos sistemas autonômicos desenvolvidos atualmente. Através desse modelo arquitetural é possível criar e organizar, de acordo com as fases do ciclo de gerenciamento autonômico, componentes responsáveis por realizar as atividades de monitoramento, análise, planejamento e execução das ações de reconfiguração, utilizando uma base de conhecimento que está presente em todas as fases do ciclo. O Capítulo 3 descreve uma visão geral sobre a tolerância a falhas em grades de computadores, na qual apresentamos conceitos gerais de tolerância a falhas, abordagens para tratamento de falhas em sistemas distribuídos, os tipos de defeitos mais frequentes em sistemas computacionais, o comportamento comum de sistemas que falham e as principais técnicas de tolerância a falhas utilizadas em grades computacionais. O Capítulo 4 descreve uma estratégia autonômica de tolerância a falhas para execução de aplicações em *desktop grids* oportunistas que visa melhorar o desempenho das técnicas de tolerância a falhas através da adaptação dinâmica. O Capítulo 5 descreve as avaliações realizadas através de simulações da estratégia autonômica proposta neste trabalho, levando em consideração diferentes cenários do ambiente de execução. O Capítulo 6 discute relevantes trabalhos relacionados, enquanto o Capítulo 7 apresenta as conclusões obtidas a partir deste trabalho e apresenta os trabalhos futuros que podem ser desenvolvidos a partir deste esforço inicial.

2 Introdução a Computação Autonômica

As grades de computadores são sistemas que possuem grande escalabilidade, alta heterogeneidade e ambiente de execução dinâmico. Essas características tornam o gerenciamento desses sistemas altamente complexo. A computação autonômica agrega vários campos da computação com o objetivo de dotar sistemas complexos com a capacidade de autogerenciamento.

Neste capítulo serão apresentados alguns conceitos de computação autonômica, sua arquitetura básica e as etapas do ciclo de gerenciamento MAPE-K: monitoramento, análise, planejamento e execução.

2.1 Conceitos Básicos de Computação Autonômica

O termo computação autonômica surgiu em 2001, com um manifesto publicado por Paul Horn, pesquisador da IBM, que lançou um desafio sobre o problema da crescente complexidade de gerenciamento do software [34]. O termo autonômico vem da biologia e está relacionado às reações fisiológicas involuntárias do sistema nervoso [51]. No corpo humano, o sistema nervoso autonômico cuida de reflexos inconscientes, isto é, de funções corporais que não requerem nossa atenção como a contração e expansão da pupila, funções digestivas do estômago e intestino, a frequência e profundidade da respiração, a dilatação e constrição de vasos sanguíneos, etc. Esse sistema reage às mudanças, ou perturbações, causadas pelo ambiente através de uma série de modificações, a fim de conter as perturbações causadas ao seu equilíbrio interno.

Em analogia ao comportamento humano, dizemos que um sistema computacional está em equilíbrio quando o seu ambiente interno (formado pelos seus subsistemas e pelo próprio sistema) está em equilíbrio com o ambiente externo. Conforme observaram Parashar e Hariri em [57], se o ambiente interno ou externo perturba a estabilidade do sistema, ele sempre atuará de modo a recuperar o equilíbrio original. Dessa forma, os sistemas de computação autonômica são sistemas capazes

de se autogerenciarem e se adaptarem dinamicamente às mudanças a fim restabelecer seu equilíbrio de acordo com as políticas e os objetivos do negócio do sistema [34]. Para isso, devem dispor de mecanismos efetivos que os permita monitorar, controlar e regular a si próprios, bem como recuperarem-se de problemas sem a necessidade de intervenções externas.

2.1.1 Propriedades de Sistemas Autonômicos

A essência da computação autonômica é o autogerenciamento. Para implementá-lo, o sistema deve ao mesmo tempo estar atento a si próprio e ao seu ambiente. Desta forma, o sistema deve conhecer com precisão a sua própria situação e ter consciência do ambiente operacional em que atua. Do ponto de vista prático, conforme Hariri [**?**], o termo computação autonômica tem sido utilizado para denotar sistemas que possuem as seguintes propriedades:

- **Autoconsciência (*self-awareness*):** o sistema conhece a si próprio: seus componentes e inter-relações, seu estado e comportamento;
- **Consciência do contexto (*context-aware*):** o sistema deve ser ciente do contexto de seu ambiente de execução e ser capaz de reagir a mudanças em seu ambiente;
- **Autoconfiguração (*self-configuring*):** o sistema deve ajustar dinamicamente seus recursos baseado em seu estado e no estado do ambiente de execução;
- **Auto-otimização (*self-optimizing*):** o sistema é capaz de detectar degradações de desempenho e de realizar funções para auto-otimização;
- **Autoproteção (*self-protecting*):** o sistema é capaz de detectar e proteger seus recursos de atacantes internos e externos, mantendo sua segurança e integridade geral;
- **Autocura (*self-healing*):** o sistema deve possuir a habilidade de identificar potenciais problemas e de se reconfigurar de forma a continuar operando normalmente;
- **Aberto (*open*):** o sistema deve ser portável para diversas arquiteturas de hardware e software e, consequentemente, deve ser construído a partir de protocolos e interfaces abertos e padronizados;

- **Capacidade de Antecipação (*anticipatory*):** o sistema deve ser capaz de antecipar, na medida do possível, suas necessidades e comportamentos considerando seu contexto e de se autogerenciar de forma pró-ativa.

As propriedades de autoconfiguração, auto-otimização, autocura e autoproteção são suficientes para realizar a visão original do termo [51]. Um requisito inerente aos sistemas autonômicos é que eles sejam adaptativos, ou seja, que sejam capazes de se reconfigurar dinamicamente de acordo com alterações percebidas em seu ambiente de execução. Desta forma, mecanismos de autoconsciência, consciência de contexto e autoconfiguração são usualmente considerados requisitos para a incorporação de mecanismos de auto-otimização e autocura.

2.2 Arquitetura de um Sistema Autonômico

Arquiteturas de sistemas autonômicos visam formalizar um quadro de referência que identifica as funções comuns e estabelece os alicerces necessários para alcançar a autonomia. Em geral, essas arquiteturas apresentam soluções para automatizar o ciclo de gerenciamento de sistemas, conforme mostrado na Figura 2.1, o qual envolve as seguintes atividades:

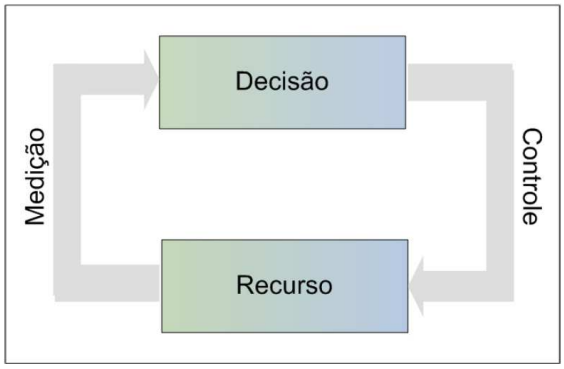

Figura 2.1: Ciclo de gerenciamento de sistemas [19]

- **Monitoramento ou Medição:** coleta, agrega, correlaciona e filtra dados sobre os recursos gerenciados;
- **Análise e Planejamento:** analisa os dados coletados e determina se devem ser realizadas mudanças nas estratégias utilizadas pelo recurso gerenciado;

- **Controle e Execução:** escalona e executa as mudanças identificadas como necessárias pela função de análise e decisão.

2.2.1 MAPE-K

Em 2003 a IBM propôs uma versão automatizada do ciclo de gerenciamento de sistemas chamado de MAPE-K (*Monitoring, Analysis, Planning, Execution and Knowledge*) [11], representado na Figura 2.2. Este modelo está sendo cada vez mais utilizado para inter-relacionar os componentes arquiteturais dos sistemas autonômicos. De acordo com essa arquitetura, um sistema autonômico é formado por um conjunto de elementos autonômicos.

Um elemento autonômico contém um único gerente autonômico que representa e monitora um ou mais elementos gerenciados (componente de hardware ou de software) [38]. Cada elemento autonômico atua como um gerente responsável por promover a produtividade dos recursos e a qualidade dos serviços providos pelo componente do sistema no qual está instalado.

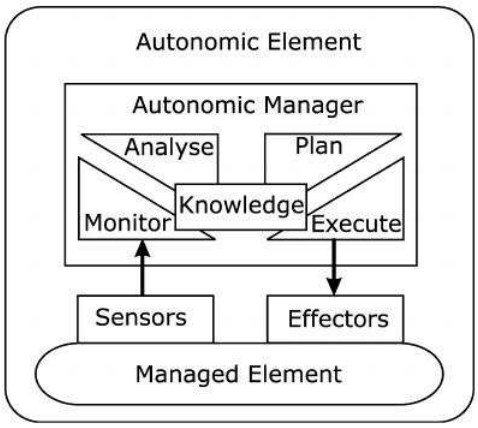

Figura 2.2: MAPE-K: Ciclo de gerenciamento autonômico

No ciclo MAPE-K autonômico (Figura 2.2), o elemento gerenciado representa qualquer recurso de software ou hardware, ao qual é dado o comportamento autonômico através do acoplamento de um gerenciador autonômico. O elemento gerenciado pode ser, por exemplo, um servidor de aplicações *Web* ou banco de dados, um componente de software específico em um aplicativo (por exemplo, o otimizador de consulta em um banco de dados), o sistema operacional, um conjunto de máquinas em um ambiente de rede, etc.

Os sensores (*sensors*) são responsáveis por coletar informações do elemento gerenciado. Estes dados podem ser os mais diversos possíveis, por exemplo, o tempo de resposta das requisições dos clientes, caso o elemento gerenciado seja um servidor de aplicações *Web*. As informações coletadas pelos sensores são enviadas aos monitores (*monitors*), os quais realizam a interpretação e pré-processamento desses dados, colocando-os em um nível mais alto de abstração. Para após isso, serem enviadas para a etapa seguinte do ciclo: a fase de análise e planejamento (*analyse and planning*). Nesta próxima fase, temos como produto uma espécie de plano de trabalho, que consiste de um conjunto de ações a serem executadas pelo executor (*execute*). O componente responsável por fazer as alterações no ambiente é chamado de atuador (*effectors*). Somente os sensores e atuadores possuem acesso direto ao elemento gerenciado. Durante todo ciclo de gerenciamento autonômico, pode haver a necessidade de uma tomada de decisão, dessa forma, faz-se necessário também a presença de uma base de conhecimento (*knowledge*), sendo que esta é comumente mais explorada na fase de análise e planejamento.

Na prática isso é implementado utilizando dois ou mais ciclos de gerenciamento autonômico, um ou mais ciclos de controle local e um global. Os ciclos de controle local tratam apenas de estados conhecidos do ambiente local, sendo baseado no conhecimento encontrado no próprio elemento gerenciado. Por esta razão, o ciclo local é incapaz de controlar o comportamento global do sistema. O ciclo global, por sua vez, a partir de dados provenientes dos gerenciadores locais ou através de um monitoramento a nível global, pode tomar decisões e atuar em todo o sistema. No entanto, a implementação das interações entres os diversos níveis existentes vai depender das necessidades e das limitações da aplicação.

Na Figura 2.3 derivada do "Ashby's Ultra-stable System" [28], a arquitetura MAPE-K está voltada para a execução de aplicações de alto desempenho, onde o ambiente interno é caracterizado pelo estado interno das aplicações em execução e o ambiente externo caracteriza o estado do ambiente de execução. O controle do sistema envolve as etapas do ciclo MAPE-K: *monitoring and analysis* (M&A); *planning engine* (PE); e *knowledge engine* (KE). O laço de controle local (L) gerencia o comportamento de elementos individuais do sistema e o laço de controle global (G) gerencia o comportamento do sistema como um todo.

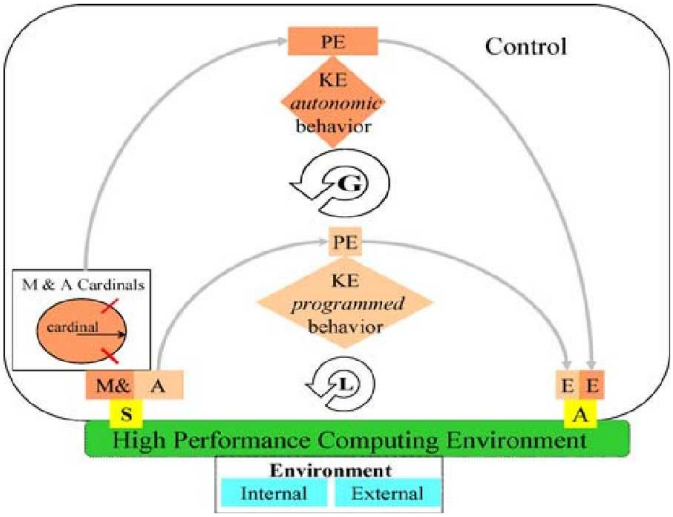

Figura 2.3: Arquitetura com ciclo autonômico local e global.

2.3 Monitoramento

O monitoramento corresponde à primeira fase do ciclo autonômico MAPE-K. Nesta etapa, sensores são utilizados com a finalidade de se obter dados que reflitam mudanças de comportamento do elemento gerenciado ou informações do ambiente de execução que sejam relevantes ao processo de autogerenciamento. Os sensores são dispositivos de hardware ou software que estão diretamente ligados ao componente que se deseja monitorar.

O tipo de informação coletada, bem como o modelo dos monitores empregados, é específico para cada tipo de aplicação. Contudo, alguns requisitos são comuns entre eles, tais como a filtragem, a escalabilidade, a dinamicidade e a tolerância a falhas [49].

Um monitor deve prover filtragem, pois a quantidade de dados coletados pode crescer muito rapidamente e consumir recursos de transmissão, processamento e armazenamento. Grande parte desses dados pode ser de pouca relevância e, portanto, descartá-los não implicaria em prejuízo. Considerando-se que um sistema pode crescer indefinidamente, contendo inúmeros objetos sob monitoramento, a tarefa dos monitores não pode consumir recursos e reduzir o desempenho do sistema. Em contrapartida, as mudanças no ambiente ou no comportamento do sistema não podem afetar o serviço de monitoramento [24].

Além disso, falhas podem ocorrer e o monitor pode apresentar algum comportamento atípico ou mesmo parar de funcionar completamente. Dessa forma, devem ser levadas em consideração provisões como redundância e mecanismos de recuperação de falhas dos monitores.

Uma das atividades que pode está envolvida na função de monitoramento é a detecção de eventos. Como observamos em alguns trabalhos [1][65][73] [60], ela é implementada como um componente que integra o serviço de monitoramento. Nessa tarefa, esse componente é responsável por fazer uma pré-análise dos "dados crus"(isto é, os dados do mesmo modo que são recebidos dos sensores) a fim de gerar eventos de interesse para o sistema. Os eventos gerados devem ter alguma relevância para o processo de tomada de decisão. Dependendo do que está sendo monitorado e dos detalhes que o sistema necessita, os eventos podem ser, por exemplo: elevado consumo de CPU ou uso de CPU em 90%, ocorrência ou suspeita de falhas, altas taxas de chegadas de requisições de usuário, etc.. Para que um sistema autonômico consiga tomar as decisões corretas é necessário haver precisão na detecção de eventos, uma vez que para se obter um diagnóstico correto sobre o estado do sistema deve-se distinguir com exatidão os eventos decorridos.

As técnicas de detecção de eventos variam desde estabelecer limiares (por exemplo, quando o consumo de CPU alcançar 80% disparar um evento de alto consumo de CPU) até a utilização de seleção de características e técnicas de sistemas especialistas (como função de utilidade, lógica *fuzzy*, etc.) que ajustam o monitor de acordo com os registros de monitoramento a fim de obter maior precisão. Estudaremos algumas dessas técnicas na Seção 2.4 seguinte cujo foco será o processo de decisão do sistema autonômico.

2.3.1 Classificação de Sistemas de Monitoramento

Aplicações autonômicas possuem requisitos de monitoramento diferentes, variando os elementos que se deseja monitorar. De acordo com as características destes elementos, podem ser implementadas formas de monitoramento específicas. Em [31], Huebscher e McCann identificaram dois tipos de monitoramento em sistemas autonômicos, classificados de acordo com o tipo de sensor utilizado:

- **Monitoramento Passivo:** no monitoramento passivo os dados obtidos de monitoramento são fornecidos pelo sistema sem necessidade de nenhuma alteração na aplicação. Por exemplo, no Linux, informações sobre uso de CPU e memória podem ser coletados do diretório `/proc`;

- **Monitoramento Ativo:** no monitoramento ativo para que se possa fazer a captura de dados é necessário alterar a implementação da aplicação ou sistema operacional. São exemplos desse tipo de monitoramento, sensores que são inseridos na forma de *bytecodes* em aplicações Java. ProbeMeister[1] é uma ferramenta que permite esse tipo de operação.

Conforme observamos os trabalhos [49] e [31], pudemos constatar que o monitor também pode ser classificado quanto à estratégia utilizada:

- **Monitoramento orientado a eventos**: Eventos são ações que acontecem no sistema e podem alterar o seu estado, por exemplo, o ”processo ocioso”, ”processo em execução”, ”início do processo”, ”chegada de uma requisição”, etc. Os eventos acontecem instantaneamente e, portanto, nesse tipo de monitoramento, cada evento corresponde a um dado que é transmitido para o monitor. Essa abordagem pode ser aplicada nos casos em que a taxa de ocorrência de eventos é muito baixa. A desvantagem é que o numero de eventos gerados pode ser muito grande e informações redundantes e desnecessárias estariam sobrecarregando os recursos do sistema;

- **Monitoramento orientado ao tempo**: Nessa estratégia, o monitor obtém informações de forma periódica, ou seja, a coleta de dados do ambiente é feita de tempo em tempos, segundo um intervalo definido. Se a interpretação dessas informações representarem alguma informação significativa, eventos são gerados para notificarem os demais elementos interessados por essa informação. A grande dificuldade encontrada nessa estratégia está em definir o melhor intervalo para a realização da coleta. A desvantagem nesse caso seria que informações importantes seriam perdidas durante esse intervalo.

[1] O projeto ProbeMeister é financiado pela DARPA (uma agência do Departamento de Defesa dos EUA) no âmbito do projeto DASADA, o qual visa fornecer suporte abrangente para avaliação de aplicações e componentes, adaptação e mudança em tempo de execução.

- **Monitoramento autonômico**: As duas estratégias acima podem ser intercaladas conforme ocorrem mudanças no ambiente. Por exemplo, se a taxa de ocorrência de eventos está elevada a melhor estratégia seria a orientada ao tempo. O tamanho do intervalo de tempo na segunda estratégia também pode ser refinado segundo uma abordagem autonômica de acordo com as características do ambiente. Como é o caso em [1] que a frequência de monitoramento é modificada em função da qualidade de serviço.

2.4 Análise e Planejamento

A análise é a fase que ocorre depois do monitoramento no ciclo de gerenciamento MAPE-K, tendo como fase seguinte o planejamento. Na prática estas duas fases são geralmente implementadas em um único componente. O processo de análise e planejamento é essencial para autonomia do sistema, pois é nele que são geradas as decisões sobre quais modificações serão realizadas no sistema, que corresponde ao elemento gerenciado.

A fase de análise e planejamento recebe como entrada os eventos e seus dados gerados pelo sistema de monitoramento e gera como saída um conjunto de ações, também chamado de plano de ações. Estas ações correspondem às operações de reconfiguração que, de acordo com o mecanismo de tomada de decisão, devem ser executadas no sistema a fim de manter o equilíbrio do sistema considerando seus objetivos. Muitas técnicas podem ser empregadas na fase de análise e planejamento, entre as quais se destacam o uso de regras Evento-Condição-Ação, funções de utilidade e aprendizado por reforço.

2.4.1 Análise e Planejamento Baseados em Regras ECA

Regras do tipo Evento-Condição-Ação (*ECA Rules*) determinam as ações a serem realizadas quando um evento ocorre desde que certas condições sejam satisfeitas. De acordo com [19], *ECA Rules* são especificações declarativas de regras ou regulamentações que determinam o comportamento de componentes de aplicações. Para cada evento é definido um conjunto de regras que podem gerar uma ou mais ações. Esses eventos também podem satisfazer as condições de diferentes regras

e algumas dessas regras podem gerar ações que conflitam entre si. Nem sempre esses conflitos podem ser detectados durante a escrita das regras, sendo muitas vezes descobertos em tempo de execução.

ECA Rules são regras definidas da seguinte forma:

on *event* **if** *condition* **do** *action*;

Figura 2.4: Definição de uma regra do tipo ECA.

O evento especifica quando as regras deverão ser acionadas; a condição é a parte a ser verificada, podendo esta ser satisfeita ou não. E por fim, a "ação" que é a operação a ser realizada caso a condição seja satisfeita.

2.4.2 Análise e Planejamento Baseados em Funções de Utilidade

A teoria da utilidade foi inicialmente aplicada na economia para estudar as decisões de consumo quando se coloca em alternativa vários bens e serviços ou a posse da riqueza. Por outro lado, ela tem sido frequentemente utilizada pela ciência da computação em problemas de tomada de decisão, quando existem objetivos contraditórios ou vários objetivos a alcançar e nenhum deles pode ser atingido com certeza [61]. A utilidade fornece um meio pelo qual a probabilidade de sucesso pode ser ponderada em relação à importância dos objetivos.

Funções Utilidade são funções que fazem o mapeamento de estados (ou uma sequência de estados) em números reais [61]. Ou seja, as preferências de um agente entre estados do mundo são captadas por uma função, que atribui um único número para expressar a *"desejabilidade"* de um estado.

Análise e Planejamento baseados em funções de utilidade permitem determinar, com precisão, o "melhor"estado viável. Essa abordagem atribui um valor utilitário para cada variante da aplicação em função do contexto de aplicação, propriedades e objetivos. Muitos trabalhos utilizam funções de utilidade para qualificar e quantificar a conveniência de diferentes alternativas de adaptação [32][68][37].

Podemos dividir o domínio da tomada de decisão em dois grupos:

- **Determinístico:** é aquele onde uma ação tem um efeito conhecido, direto e garantido. Ou seja, não existe incerteza sobre o estado final do ambiente após a execução da ação.

- **Não-Determinístico:** é aquele onde uma ação A terá vários estados resultantes possíveis $Resultado_i(A)$, onde i varia sobre os diferentes resultados. O agente atribui a probabilidade $P(Resultado_i(A) \mid Fazer(A),\ E)$ a cada resultado, onde E resume a evidência sobre o ambiente, ou seja, alguma informação do ambiente que possa influenciar nas probabilidades. Dessa forma, as utilidades são combinadas com probabilidades de resultados de ações para fornecerem uma utilidade esperada referente a cada ação.

A tarefa do agente é trazer possíveis estados que maximizem a utilidade. Assim, podemos calcular a utilidade esperada da ação, dada a evidência, usando a seguinte formula:

$$\boxed{UE(A|E) = \sum_i P(Resultado_i(A)|Fazer(A), E)\ x\ U(Resultado_i(A))}$$

Figura 2.5: Segunda Fórmula.

Segundo o "Principio da Utilidade Máxima Esperada (UME)" um agente racional deve escolher uma ação que maximize a utilidade esperada do agente.

2.4.3 Análise e Planejamento Baseados em Aprendizado por Reforço

Aprendizado por reforço é uma técnica no qual se baseia na experiência das interações de forma a mapear estados a ações por tentativa e erro, maximizando o desempenho geral, também conhecido como valor recompensa. Por exemplo, imagine uma criança que tenta andar de patins pela primeira vez. Como a criança sabe andar sem patins, ela tenta aplicar o conhecimento prévio em busca de obter o mesmo êxito para se locomover com esse equipamento. Assim, os movimentos que ela conseguir aplicar para andar terão mais credibilidade para uso futuro em outras situações análogas a essa. No entanto, as ações que não resultam em benefícios, como uma queda, por exemplo, são armazenadas de maneira não tão positiva.

O aprendizado por reforço (*reinforcement learning*) [67] consiste, basicamente, em fazer um agente escolher suas ações se baseando apenas na

interação com o ambiente. A aprendizagem por reforço é geralmente utilizada quando não se consegue obter exemplos de comportamento correto para as situações que o agente enfrenta ou quando o agente atuará em um ambiente desconhecido.

Duas importantes propriedades da aprendizagem por reforço são a não utilização de conhecimento específico, e o aprendizado incremental. O processo de engenharia de conhecimento envolvido na construção de um agente aprendiz consiste apenas em codificar, de forma otimizada, as percepções e ações do agente e o reforço do ambiente, ou seja, definir que características do ambiente são relevantes para a tomada de decisão. Por outro lado, não é preciso codificar como o agente deve agir, nem mesmo que objetivo deve se alcançar, pois todo o aprendizado será automaticamente conduzido pelo reforço. O constante aprendizado incremental, por sua vez, permite naturalmente a adaptação a oponentes variados.

A vantagem do aprendizado por reforço é que ele não requer um modelo explícito do sistema a ser gerido [21, 18]. No entanto, ele perde na escalabilidade em tentar representar grandes espaços de estados, o qual também tem impacto no seu tempo para treinar. Para resolver esse problema, um número de modelos híbridos tem sido proposto, a fim de acelerar o aprendizado ou reduzir o espaço de domínio, como encontramos, por exemplo, em [**?**][70].

2.5 Execução

Na etapa de execução do ciclo MAPE-K são realizadas reconfigurações no sistema de forma a restabelecer seu equilíbrio. A finalidade da reconfiguração é permitir que um sistema evolua (ou simplesmente mude) incrementalmente de uma configuração para outra em tempo de execução, introduzindo pouco (ou, idealmente, nenhum) impacto sobre a execução do sistema. Desta forma, os sistemas (ou aplicações) não necessitam ser finalizados, ou reiniciados, para que haja as mudanças.

A autoconfiguração (*self-configuring*) é a característica do sistema autônomo que o permite ajustar-se automaticamente às novas circunstâncias percebidas em virtude do seu próprio funcionamento, de forma a atender a objetivos especificados pelos processos de autocura, auto-otimização ou autoproteção [19].

O processo de reconfiguração é realizado pelos executores através dos atuadores. Executores recebem como entrada um plano de ações gerado na etapa de análise e planejamento e utiliza os atuadores pertinentes para implementar as ações de reconfiguração descritas no plano. As reconfigurações devem ser realizadas dinamicamente, sem impor a necessidade de parar e/ou reiniciar o sistema.

2.5.1 Classificação das Abordagens para Reconfiguração

Segundo McKinley et al., duas abordagens gerais têm sido utilizadas para atingir adaptação: adaptação paramétrica e adaptação composicional [50]. A adaptação paramétrica consiste na alteração de variáveis que determinam o comportamento do sistema. Por exemplo, o ajuste no valor do tempo de periodicidade de *checkpointing* em uma técnica de tolerância a falhas. Já a adaptação composicional (ou estrutural) consiste na substituição de algoritmos ou partes estruturais de um software, permitindo que este adote novas estratégias (algoritmos) para tratar situações que não foram inicialmente previstas na sua construção. Podemos citar como exemplo de adaptação composicional, a troca do algoritmo de escalonamento em um ambiente de grades de computadores para melhorar o desempenho ou a eficiência com que são executas as aplicações.

2.5.2 Questões de Projeto de Mecanismos de Reconfiguração

O desenvolvimento de um mecanismo de reconfiguração dinâmica não é algo simples. Diversos requisitos devem ser considerados a fim de manter as características e funcionalidades do sistema [54], entre as quais destacamos:

- **Separação de responsabilidades:** No desenvolvimento de sistemas autonômicos, a separação de responsabilidades permite que o código funcional da aplicação (responsável pelas regras de negócio) seja separado do código responsável pela adaptação. Isto simplifica o desenvolvimento, a manutenção e o reuso do código adaptativo;
- **Confiabilidade:** Um problema quando se modifica um sistema em tempo de execução é a sincronização entre reconfigurações e execução funcional do

sistema. A reconfiguração não deve prejudicar a funcionalidade do sistema. Ledoux et al., em [44], definiram um conjunto de propriedades a fim de garantir a confiabilidade no contexto das reconfigurações dinâmicas em sistemas baseados em componentes. Esse conjunto de propriedades foi baseado em transações e denominado ACID (Atomicidade, Consistência, Isolamento e Durabilidade);

- **Preservação de consistência:** Quando a reconfiguração do sistema ocorre em tempo de execução, é importante que a reconfiguração preserve a consistência do sistema. O estado do sistema interno deve ser mantido e as informações trocadas entre os componentes não devem ser perdidas;

- **Custo da reconfiguração:** Em [27] o custo de reconfiguração é definido como uma medida dos efeitos negativos introduzidos pela reconfiguração. Esses efeitos negativos incluem, por exemplo, a indisponibilidade temporária do serviço, ou a perturbação induzida em outros serviços após o possível aumento do consumo de recursos de rede durante a reconfiguração. A relação custo/benefício deve ser avaliada.

A reconfiguração dinâmica tem sido tradicionalmente usada em sistemas distribuídos para permitir que aplicativos se adaptem às novas condições do ambiente de execução e requisitos e permitir a evolução do software. Recentemente, seu domínio foi ampliado para abranger situações em ambientes móveis, em configurações de sistemas ubíquos [27].

Uma das principais complexidades da reconfiguração dinâmica de software é a transferência de estado. Em linhas gerais, a transferência de estado refere-se ao processo que garante que o software recém-configurado inicie suas operações com informações apropriadas sobre o seu estado, formado pelas configurações anteriores, garantindo assim a continuação coerente do fluxo da aplicação.

2.6 Abordagens para o Desenvolvimento de Software Autoconfigurável

No que se refere desenvolvimento de sistemas autonômicos, alguns princípios e tecnologias vêm sendo utilizados para facilitar esse processo do ponto

de vista de programação como: reflexão computacional e programação orientada a aspectos.

2.6.1 Reflexão Computacional

Reflexão computacional é a habilidade de um software observar ou até mesmo modificar a sua estrutura ou comportamento [48]. Essa abordagem expõe detalhes de implementação do software em um nível de abstração que permite mudanças em seu comportamento sem comprometer a sua portabilidade. Dessa maneira, software reflexivo deve incorporar estruturas de dados que representam os diversos aspectos do software, em uma autorrepresentação. Esses aspectos são causalmente conectados com os aspectos de implementação do sistema. Portanto, modificações em qualquer um desses aspectos levam às mudanças no outro aspecto. Isto assegura que o software sempre tem uma autorrepresentação precisa de si mesmo e que o estado e a computação do software estarão em conformidade com essa representação.

A reflexão computacional compreende duas atividades: introspecção e intercessão [64]. A introspecção denota a capacidade que um software tem de examinar sua própria estrutura, estado e representação. A esses elementos dá-se o nome de metainformação, que representa qualquer informação contida e manipulável por um software computacional que seja referente a si próprio. Por exemplo: um software pode observar quais os métodos que compõem uma interface, os parâmetros e seus tipos de dados.

A outra atividade da reflexão é a intercessão que permite que um software aja sobre as observações obtidas e modifique seu estado e comportamento. Isto é realizado a partir da interceptação de operações realizadas pelo software e posterior modificação dessas. Por exemplo: quando o software invocar uma chamada a um escalonador de processos, esta será interceptada e modificada, introduzindo um código mais adequado a situação do ambiente computacional.

Em geral, software reflexivo é desenvolvido usando-se o padrão de desenvolvimento *reflection*[48]. Esse padrão divide um software em um metanível e em um nível base. O metanível representa a estrutura e o comportamento do software em elementos chamados metaobjetos; enquanto o nível base define a lógica da aplicação e

a implementação das regras de negócio. Esses dois níveis são causalmente conectados através de um protocolo de metaobjetos [40], de forma que modificações em qualquer um dos dois serão refletidas no outro.

A principal vantagem da organização do software reflexivo em duas camadas é a nítida separação do código em dois níveis de funcionalidade, um nível base que provê funcionalidade do domínio e um metanível que permite que o comportamento, a forma ou a implementação do nível base sejam manipulados, regulados ou influenciados. Em decorrência disto, software reflexivo apresenta outras vantagens como maior reutilização do código, mais facilidade na manutenção e depuração do código e maior transparência na incorporação de conteúdo adaptativo no software.

2.6.2 Programação Orientada a Aspectos (POA)

Kiczales et al. em [39] observaram que em alguns sistemas de software complexos, desenvolvidos segundo o paradigma da orientação a objeto, existem vários requisitos funcionais e não-funcionais entrelaçadas ao longo do código computacional, tais como: padrões de acesso à memória, tolerância a falhas, segurança e persistência de dados. Isto dificulta o desenvolvimento e manutenção do software, pois a alteração em um desses requisitos implica na modificação de todas as classes que utilizam o mesmo.

A programação orientada a aspectos (POA) surgiu como consequência do princípio de separação de responsabilidades e permite expressar responsabilidades entrelaçadas em termos de elementos chamados aspectos, desenvolvidos separadamente de outras partes do sistema e armazenados em uma unidade de código visível a todos os componentes do sistema [69]. Desta maneira, o software é organizado em classes ou componentes que representam as regras de negócio da aplicação (regras de negócio), e aspectos relacionam os requisitos funcionais e não-funcionais que afetam o comportamento do sistema e que estariam entrelaçados no código das classes e componentes.

O uso de POA auxilia o desenvolvimento de software autoadaptativo, pois muitas mudanças no ambiente computacional estão relacionadas a requisitos que estão entrelaçados no código da aplicação. Dessa maneira, o encapsulamento desses

requisitos em aspectos permite a modificação do software em um único ponto, ao invés de modificar todos os locais em que esse requisito está presente. Além disso, é possível substituir dinamicamente a implementação de um aspecto por outra, sem que haja modificação no código da aplicação. Por exemplo, a modificação do mecanismo de persistência de dados utilizado pela aplicação em face da diminuição da largura de banda disponível.

2.7 Conclusão

Nesse capítulo vimos que para um sistema ser autonômico ele deve ser capaz de se autogerenciar. Isso implica que ele deve possuir propriedades como autoconfiguração, auto-otimização, autoproteção, autocura, autoconsciência e ciência de contexto. Também apresentamos o modelo arquitetural MAPE-K *loop*, proposto originalmente pela IBM, que constitui o modelo mais utilizado atualmente no desenvolvimento de sistemas autonômicos. Vimos também que na prática, um sistema pode ter mais de um ciclo autonômico, o que torna necessário a utilização gerentes autonômicos locais e globais. Em sistemas distribuídos, os ciclos locais são usualmente utilizados para gerenciar o comportamento autonômico de componentes locais, enquanto que os ciclos globais controlam os ciclos locais e o comportamento do sistema como um todo.

Também descrevemos alguns aspectos que envolvem as fases do ciclo o autonômico. No monitoramento, os sensores enviam os dados puros do ambiente para os monitores, que são responsáveis por processá-los e disponibilizá-los em um nível mais alto de abstração. Diversas técnicas como *ECA Rules* e funções de utilidade podem ser utilizadas na fase de análise e planejamento para realizar a tomada de decisão e gerar o plano de ações de reconfiguração do sistema. Na fase de execução ocorrem reconfigurações que podem ser paramétricas ou estruturais, sendo observados os critérios de separação de responsabilidades, confiabilidade, preservação de consistência e custo.

Por fim, entre as abordagens usualmente empregadas na implementação de mecanismos de reconfiguração dinâmica podemos citar o uso de Reflexão Computacional e Programação Orientada a Aspectos.

3 Tolerância a Falhas em Grades de Computadores

O aumento da confiabilidade de software em geral tem sido alcançado empregando técnicas básicas para evitar erros de programação ou a realização de testes de software. Esses cuidados podem e devem ser utilizados, mas não são suficientes para garantir uma operação completamente livre de falhas em sistemas computacionais.

Em grades de computadores oportunistas, a tolerância a falhas representa uma característica muito importante, uma vez que não se pode garantir a confiabilidade dos recursos que a compõem, nem que eles não estejam sendo manipulados por algum agente malicioso. Características como escalabilidade, heterogeneidade e a dinamicidade estão presentes no ambiente de grade e aumenta muito a probabilidade de erros e o grau de complexidade na realização do tratamento de falhas [62].

Neste capítulo veremos alguns conceitos de tolerância a falhas, abordagens contra falhas em sistemas distribuídos, os tipos de defeitos mais frequentes em sistemas computacionais, o comportamento comum de sistemas que falham e as principais técnicas de tolerância a falhas utilizadas em grades computacionais.

3.1 Conceitos de Tolerância a Falhas

Para compreensão do que é tolerância a falhas é necessário entender o significado de alguns termos que podem ser confundidos por apresentar sentidos muito próximos. Os termos da língua inglesa *"fault"*, *"error"* e *"failure"* têm sido traduzidos para o português com diferentes significado em trabalhos nacionais, o que tem causado muita confusão. Para evitar esse problema, decidimos adotar neste trabalho o sentido que tem sido mais utilizado na teoria da tolerância a falhas. Dessa

forma, transcrevemos as definições dadas por Jalote, em [36] utilizando os seguintes termos:

- **Falha (*Fault*):** É um evento caracterizado por um comportamento não esperado de um componente ou uma parte do sistema, o qual tem um potencial para a geração de erros, ou seja, podem ocasionar erros, mas nem sempre isso acontece.
- **Erro (*Error*):** É uma propriedade do estado do sistema que foi ocasionado por uma falha (*fault*).
- **Defeito ou Avaria (*Failure*):** É um desvio de comportamento do sistema que o leva a atuar de forma avessa aos requisitos de suas especificações.

Dessa forma, podemos compreender Tolerância a Falhas como sendo a utilização de mecanismos e técnicas para que um sistema possa manter seu comportamento consistente com suas especificações, ainda que na presença de falhas (faults). Podemos, portanto, traduzir a expressão *"fault tolerance"* como *"tolerância a falhas"*, o qual tem sido amplamente utilizado na literatura.

3.2 Técnicas de Abordagens Contra Falhas

Existem duas principais abordagens utilizadas contra falhas em sistemas distribuídos: a prevenção e a tolerância [36]. A abordagem baseada na prevenção de falhas tenta evitar o acontecimento ou a introdução de falhas no sistema e são geralmente empregadas ações e procedimentos durante o desenvolvimento do projeto do sistema. Embora sejam aplicados grandes esforços nesse sentido, a prevenção de falhas, hoje, não é capaz de eliminar totalmente as falhas. Pois ainda não é possível testar exaustivamente os sistemas, a não ser que estes sejam muito simples [55], e nem sempre é viável proceder-se uma rigorosa especificação formal do software, existindo sempre a possibilidade de um sistema estar vulnerável e sujeito a falhas.

A abordagem baseada na tolerância a falhas tenta prover a continuidade do funcionamento do serviço a despeito da ocorrência de falhas. Um sistema é dito tolerante a falhas se na presença de falhas ele consegue mascará-las de modo que se mantenha consistente com as suas especificações.

São consideradas as seguintes fases na abordagem de tolerância a falhas:

- **Detecção de Erro:** é o ponto inicial para a tolerância a falhas. O ideal é que o mecanismo de detecção seja capaz de descobrir todos os possíveis erros causados pelas falhas que se deseja tratar.
- **Contenção de Danos:** o atraso entre a ocorrência do erro e a detecção pode permitir que o erro de propague e se espalhe para outras partes do sistema. O objetivo dessa fase é determinar o alcance da corrupção do sistema e de seus componentes.
- **Recuperação do Erro:** após a detecção e a identificação da extensão do erro, devem ser feitos esforços para a recuperação do erro, ou seja, para que o sistema volte ao seu estado operacional.
- **Tratamento de Falhas:** deve-se garantir que o componente defeituoso não continue gerando erros após a recuperação do erro. Deve-se nesta fase localizar o componente defeituoso e efetuar a reparação substituindo ou reconfigurando o mesmo.

3.2.1 Tipos de Falhas

Sistemas computacionais podem apresentar falhas por uma série de fatores. Sathya e Babu em [62], apresentam vários desses motivos e os classificam da seguinte forma:

- **Falhas de Rede:** são geralmente causados por falhas na partição da rede, por perda de pacotes, por corrupção dos dados, etc.;
- **Falhas físicas:** são causados por falhas nos equipamentos ou em seus componentes como, por exemplo, falhas de CPUs, falhas em memórias, falhas de discos (locais de armazenamento), entre outros;
- **Falhas de ciclo de vida:** são falhas geradas por erros cometidos na atualização ou manutenção de um sistema, inclusive nas mudanças de suas versões;
- **Falhas de mídia:** são falhas causadas às mídias, tais como defeitos na cabeça de leitura e gravação do disco rígido, danificação de setores do disco, etc.;

- **Falhas de processador:** são falhas que causam a pane da máquina ou do sistema operacional;
- **Falhas de processo:** são falhas causadas pela escassez de recursos, problemas de software, etc.;
- **Falhas de Interrupções de usuário:** são aquelas falhas geradas pela intervenção do usuário na execução normal do sistema. Por exemplo, quando o usuário decide forçar o reinício do sistema operacional e pressiona as teclas "`Ctrl + Alt + Del`". Um outro exemplo é quando ele decide forçar o término de um processo e pressiona as teclas "`Ctrl + C`";
- **Falhas por expiração do serviço:** o tempo de serviço de um recurso pode expirar durante a execução de uma aplicação que o esteja usando;
- **Falhas de interação:** são falhas ocasionadas por problemas na interação de dois ou mais sistemas, podendo ser causados por incompatibilidades de protocolos, incompatibilidade de segurança, ou problemas de políticas, problemas de atrasos no processamento, entre outros.

Com relação ao tempo de duração, as falhas em sistemas computacionais podem ser classificadas como:

- **Transientes:** são falhas de duração limitada, causadas por mal-funcionamento temporário ou interferência externa;
- **Intermitentes:** falhas que acontecem repetidamente por curto intervalo de tempo;
- **Permanentes:** são falhas que quando acontecem, não permitem que o sistema ou o componente volte ao seu funcionamento normal.

3.2.2 Comportamento de Sistemas que Falham

Em sistemas distribuídos, quando um sistema falha, ele pode se comportar de diferentes modos. Os mais comuns comportamentos de falhas são [62]:

- **Falha de parada (*Fail-stop*):** O sistema simplesmente para. Ele não produz qualquer saída de dados após a falha. Também para imediatamente de enviar ou de responder a quaisquer eventos ou mensagens;

- **Falha bizantina (*Byzantine*):** O sistema não para após uma falha, contudo se comporta de modo inconsistente, enviando informações errôneas ou atrasando o envio de mensagens. Pode ser causado por mal-funcionamento ou por ação maliciosa;

- **Falha rápida (*Fail-fast*):** O sistema se comporta inicialmente com falhas bizantinas e após um curto período de tempo o sistema para (fail-stop).

3.3 Técnicas de Tolerância a Falhas em Grades Computacionais

A tolerância a falhas pode ser aplicada em várias situações no decorrer do funcionamento da grade, tais como durante a transferência de grandes volumes de dados, enquanto as tarefas de uma aplicação estão em execução ou mesmo durante o funcionamento dos componentes que formam o *middleware* da grade. Entretanto, como este não é o foco deste trabalho, trataremos apenas da fundamentação relativa à falhas na execução de aplicações.

Hwang e Kesselman dividiram as principais técnicas utilizadas para prover de tolerância a falhas na execução de aplicações em grades computacionais em dois níveis: nível de tarefa e nível de *workflow* [33][72]. Essa taxonomia está esquematizada conforme pode ser observado na Figura 3.1.

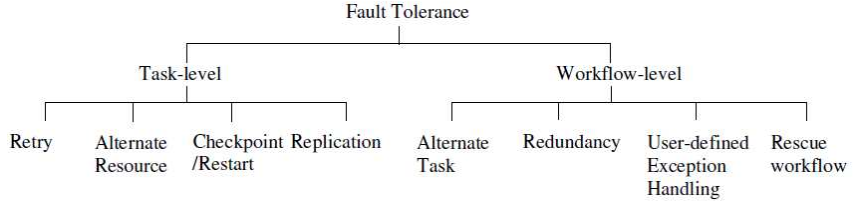

Figura 3.1: Taxonomia das técnicas de tolerância a falhas utilizadas em grades.

- Nível de tarefa: refere-se às técnicas de tolerância a falhas que atuam diretamente no nível da tarefa, aplicando técnicas de recuperação para mascarar o efeito das falhas;

- Nível de fluxo de trabalho (*workflow*): refere-se às técnicas de tolerância a falhas que atuam no controle fluxo de execução das tarefas determinado pelo *workflow*. Essas técnicas permitem que sejam criados no *workflow* procedimentos de recuperação para mascarar o efeito das falhas;

3.3.1 Técnicas do Nível de Tarefa

As técnicas que atuam no nível de tarefa são comumente usadas em sistemas de grades computacionais e independem se o *middleware* da grade oferece do suporte à execução de *workflows*. As principais técnicas que atuam nesse nível são: reinício, replicação e *checkpointing*.

Reinício (*Retrying*)

A mais simples técnica usada para prover tolerância a falhas em grades computacionais, o reinício consiste em tentar reiniciar a execução da mesma tarefa, podendo ser feito no mesmo recurso ou ser re-escalonada para um recurso diferente daquele em que ocorreu a falha. Esse re-escalonamento pode adotar diferentes critérios para a seleção dos recursos como, por exemplo, escolher o primeiro recurso a se tornar disponível (esquema FIFO), o recurso de maior poder computacional, o mais ocioso, o de maior credibilidade (em relação à segurança e à proteção contra agentes maliciosos), ou ainda se basear num histórico de monitoramento da utilização dos recursos por parte dos usuários (em grades de nós não dedicados – *desktop grids*).

Replicação (*Replication*)

É a técnica de tolerância a falhas que executa diferentes réplicas da mesma tarefa em diferentes recursos da grade simultaneamente tendo a expectativa de que ao menos uma delas terminará sua execução com sucesso.

A execução dessas réplicas precisa ser coordenada, de modo que seja garantida a alocação e liberação adequada dos recursos da grade utilizados pelas réplicas. As réplicas podem ser escalonadas para recursos no mesmo ou em diferentes domínios. Uma vez que falhas de redes podem acontecer tornando todo um conjunto de nós inacessível, os demais domínios podem funcionar como um *backup* garantindo o término das tarefas e entrega dos resultados aos seus clientes.

Outro fator importante quando se utiliza replicação é a integridade do resultado da computação, uma vez que nós defeituosos ou que tenham sofrido a ação de algum agente malicioso podem produzir resultados errôneos. Para prevenir esta situação, pode-se adotar um algoritmo de concordância bizantina, devendo-se aguardar que todas as réplicas cheguem ao final de sua computação para comparação dos seus resultados.

Checkpointing

Esta técnica consiste em salvar periodicamente o estado de uma aplicação a fim de que, na ocorrência de falhas, ela seja reiniciada e continue sua execução a partir do seu último estado salvo. Nesse processo é importante que as informações a cerca do estado da aplicação sejam consistentes, tanto durante sua gravação quanto em sua recuperação, e espelhem o mesmo estado de execução de quando foram persistidas, antes da falha. Com aplicações paralelas acopladas, isto é, cujas tarefas realizam trocas de mensagens ao longo de sua execução, essa preocupação se torna mais incisiva, pois o estado consistente da aplicação é completamente dependente dos estados salvos de cada processo que a compõe.

Uma situação, exemplificada na Figura 3.2, demonstra como são geradas inconsistências de estados no *checkpointing*. As duas linhas correspondem cronologicamente à execução dos processos a e b. O processo a tem seu estado salvo e envia a mensagem m'. A tarefa b recebe a mensagem m' e só depois tem seu estado salvo. Na recuperação do estado dessa tarefa, teremos uma inconsistência, pois a tarefa a tornará a enviar a mensagem m' e b que já teria recebido m' receberá novamente m'.

Com o intuito de garantir que o estado global de aplicações paralelas acopladas seja consistente, são utilizados certos protocolos. Esses protocolos podem

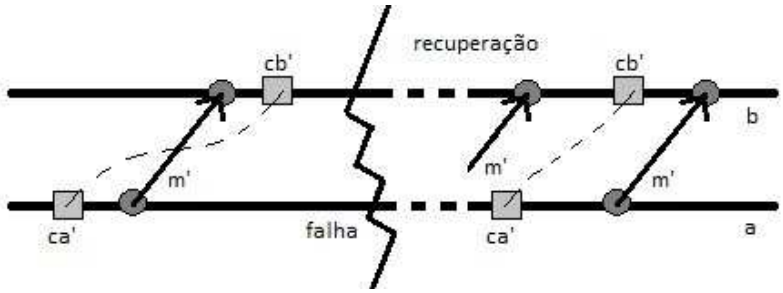

Figura 3.2: Problema de inconsistência de *checkpointing* na troca de mensagens.

ser divididos basicamente em três categorias: não-coordenados, coordenados e induzidos por comunicação [20, 15].

- **Não-coordenado:** a tomada do *checkpoint* é feita individualmente por cada processo, sem que haja coordenação com os demais e por isso é mais flexível, podendo ser realizado a qualquer instante. Entretanto, a construção de um estado global consistente durante o processo de recuperação é uma tarefa difícil, pois nem sempre o último *checkpoint* pode ser utilizado na construção desse estado global. Nesse caso, é traçada uma linha de recuperação entre os *checkpoints* dos processos, de modo que os *checkpoints* mais recentes que irão formar o estado consistente global são separados dos *checkpoints* que devem ser descartados. Algumas vezes vários *checkpoints* devem ser descartados e pode inclusive não se formar um estado global consistente, sendo necessário desprezar todos os *checkpoints* e re-executar a aplicação desde o início.

- **Coordenado:** o *checkpoint* de cada tarefa individual é orquestrado em conjunto com os demais processos, de forma que os últimos estados salvos de todos os processos da aplicação sempre formem um estado consistente global. O *checkpointing* coordenado pode ser feito de forma bloqueante ou não-bloqueante. Ele é dito bloqueante quando um processo coordena o momento em que todos os processos devem parar sua execução e salvar seus estados, permanecendo bloqueados até que todos terminem de fazer o *checkpoint*. No *checkpointing* coordenado não-bloqueante, um processo salva seu *checkpoint* e envia uma mensagem aos demais para que estes também salvem seus novos estados. Durante essa operação, nenhuma outra mensagem é enviada entre os processos, exceto a confirmação da operação que deve ser trocada entre todos os processos.

- **Induzido por comunicação:** Neste protocolo, os processos não são coordenados e as salvas dos estados dos processos são feitas individualmente. Para evitar que *checkpoints* sejam descartados, como acontece no protocolo não-coordenado, são realizados *checkpoints* adicionais durante a comunicação, ou seja, além do salvamento periódico dos estados dos processos, eles também devem salvar seu estado antes de enviarem ou receberem mensagens a fim de manter a linha de recuperação sempre à frente e garantir a formação de um *checkpoint* global consistente.

O *checkpointing* ainda pode ser classificado em relação à forma de obtenção das informações sobre o estado do processo. Nessa classificação, existem duas principais abordagens [15]:

- **no nível sistema:** consiste em obter e armazenar o estado da aplicação diretamente a partir dos dados contidos no espaço de memória da aplicação, junto com informações de registradores e do estado do sistema operacional. Para que o mecanismo de *checkpointing* possa atuar nesse nível, o sistema operacional deve disponibilizar funções que permitam realizar essas operações ou alterações no núcleo do sistema devem ser feitas nesse sentido. A vantagem desta abordagem é que o processo é totalmente transparente à aplicação, podendo ser utilizado sem modificações para aplicações escritas em diferentes linguagens de programação. A maior desvantagem é que os *checkpoints* gerados não são portáveis, o que torna essa abordagem não muito atrativa em *desktop grids* oportunistas.

- **no nível da aplicação:** a aplicação é totalmente responsável por fornecer e recuperar os dados sobre o estado que devem ser persistidos no *checkpoint*. Nesse caso, o programador pode definir os momentos da computação, a quantidade e os tipos de dados que devem incluídos no *checkpoint*, mas para isso deve instrumentar o código da aplicação, ou seja, reescrever a aplicação inserindo instruções específicas para realização do *checkpointing*. Uma outra abordagem comumente usada em grades computacionais inclui a recompilação da aplicação utilizando bibliotecas fornecidas junto com os serviços da grade. Esse processo, além de prover *checkpoints* portáveis, permite que o *middleware* da grade tenha maior controle sobre quando o *checkpointing* da aplicação deve ser realizado.

3.3.2 Técnicas do Nível de *Workflow*

As técnicas desse nível de atuação são aplicadas somente às grades que permitem a execução de *workflows*. Estas técnicas são baseadas no conhecimento do contexto de execução das tarefas e no controle do fluxo que foi definido para execução das suas computações. As técnicas mais conhecidas são [33]:

Tarefa Alternativa (*Alternative Task*)

A ideia básica desta técnica consiste em utilizar uma tarefa alternativa para executar no lugar da tarefa que falhou. Para isso o usuário deve dispor de implementações diferentes para realizar a mesma tarefa computacional. O emprego dessa técnica pode ser útil quando cada implementação possui diferentes características. Por exemplo, a primeira implementação executa mais rápida, porém é menos confiável. Já a segunda é mais lenta, contudo, mais confiável. Nesse caso o usuário pode especificar a segunda implementação como uma tarefa alternativa à primeira.

Redundância (*Redundancy*)

Essa técnica consiste em oferecer redundância para a execução de uma tarefa, de modo que diferentes implementações dessa tarefa são executadas ao mesmo tempo em recursos distintos. A tarefa computacional é concluída com sucesso quando uma das implementações em execução finaliza de forma bem sucedida. Nesse caso, as demais implementações em execução são eliminadas. Para que essa técnica possa ser utilizada, o usuário deve dispor de diferentes implementações para realizar a mesma tarefa computacional. Por exemplo, uma implementação mais confiável e mais lenta pode ser executada em um recurso, enquanto que uma segunda implementação para a mesma tarefa, porém menos confiável e mais rápida executa em outro.

Tratamento de Exceções Definidas pelo Usuário (*User Defined Exception Handling*)

Segundo Hwang e Kesselman, esta técnica permite que os usuários forneçam tratamento a uma falha específica de uma tarefa particular [33]. Por exemplo, o usuário dispõe de uma tarefa que consume muito espaço em disco, mas executa de

forma muito rápida. Entretanto, uma falha específica e bem previsível de acontecer durante a execução da tarefa é que o espaço em disco seja insuficiente para que essa tarefa continue sua execução. Nesse caso, o usuário pode definir um tratamento específico para esse tipo de falha baseado na técnica de tarefa alternativa e executar uma implementação que não consuma tanto espaço em disco. O que diferencia esta técnica das demais é o tratamento de falhas específicas, que é permitido ao usuário determinar.

Workflow de Resgate (*Rescue Workflow*)

A principal ideia por trás dessa técnica consiste em gerar um novo *workflow* (chamado *workflow* de resgate) para executar as tarefas do fluxo original que falharam e as que não puderam executar por dependerem de alguma tarefa que falhou. Para isso, um fluxo idêntico ao original é gerado de forma que as tarefas que conseguiram concluir com sucesso são marcadas para não executarem novamente. Esse novo *workflow* deve então ser ressubmetido para execução. Caso o *workflow* de resgate falhe, novos resgates podem ser gerados e ressubmetidos [10].

3.4 Conclusão

Pudemos ver neste capítulo que um sistema tolerante a falhas deve manter seu comportamento consistente com suas especificações a despeito da ocorrência de falhas. Em sistemas distribuídos, tem-se empregado abordagens baseadas na prevenção e na tolerância a falhas. A abordagem tolerante a falhas possui as fases de detecção de erro, contenção do dano, recuperação do erro e tratamento de falhas.

Também apresentamos alguns tipos de falhas e os diversos fatores que podem causá-las. As falhas podem provocar diferentes comportamentos aos sistemas, por exemplo, tornando-se completamente inativo ou inconsistente. Além disso, foram apresentadas as principais técnicas de tolerância a falhas empregadas nos atuais sistemas de grades computacionais.

Os conceitos abordados neste capítulo serão utilizados em nosso trabalho para elaboração de um mecanismo de tolerância falhas autonômico baseado na

adaptação paramétrica e estrutural de algumas técnicas de tolerância a falhas aqui apresentadas.

4 Abordagem Autonômica de Tolerância a Falhas

Nesse capítulo apresentaremos uma estratégia autonômica para prover tolerância a falhas para a execução de aplicações em *desktop grids* oportunistas. Uma vez que o ambiente de execução provido por este tipo de grade computacional é tipicamente dinâmico, a abordagem que será apresentada foi concebida buscando alcançar dois principais objetivos: maximizar o número de aplicações que concluem com sucesso e minimizar o tempo de conclusão dessas aplicações. Para isso, a abordagem tenta explorar as vantagens das técnicas de replicação e *checkpointing* de acordo com diferentes condições do ambiente de execução. Apresentaremos como a estratégia foi estruturada de modo que pudesse oferecer tanto adaptações no nível paramétrico quanto no nível estrutural. Veremos também os mecanismos introduzidos na estratégia autonômica desenvolvida com o objetivo de prover um bom desempenho para aplicações *bag-of-tasks*, dado que esta classe de aplicações é muito comum em ambientes de grades oportunistas. Além disso, descreveremos como pode acontecer o efeito *ping-pong*, causado por oscilações em torno dos pontos da tomada de decisão, e como buscamos tratar adequadamente esse efeito, evitando perda de desempenho da abordagem proposta.

4.1 Estratégia de Tolerância a Falhas Proposta

Desktop grids oportunistas são ambientes computacionais muito dinâmicos. Existem diversos fatores que contribuem para que eles tenham essa característica. Dentre eles, podemos citar: as variações na taxa de disponibilidade de recursos (dado que os mesmos podem se registrar e sair da grade de forma imprevisível, de acordo com a necessidade de seus proprietários), as variações na taxa de ocorrência de falhas de recursos (por estes serem não-dedicados e controlados por seus usuários), no volume de aplicações submetidas para execução, no grau de heterogeneidade das tarefas que compõem as aplicações submetidas, entre outros.

Esses fatores influenciam diretamente no desempenho da técnica de tolerância a falhas na execução de aplicações empregada pelo *middleware* da grade. Por exemplo, em um ambiente de grade com muitos recursos disponíveis para execução de aplicações, o uso de replicação das tarefas irá aumentar a probabilidade de pelo menos uma réplica ser bem sucedida em sua execução. Isso também poderia aumentar as chances de uma réplica ser escalonada para um recurso rápido, resultando em um menor tempo de conclusão da aplicação. Entretanto, se no ambiente da grade a quantidade de recursos disponíveis não for grande, essa configuração poderá causar uma sobrecarga do sistema, ocupando os recursos para a execução de réplicas e atrasando o início da execução dos processos de novas submissões. Nesse sentido, quanto menor for a disponibilidade de recursos maior será essa sobrecarga, degradando cada vez mais o tempo médio de conclusão das aplicações. Dependendo da quantidade de recursos, poderiam ser utilizadas menos réplicas, mas é preciso saber a dosagem exata para que a quantidade de réplicas não seja insuficiente para garantir que a tarefa conclua com sucesso. Em outros casos, há tão poucos recursos no ambiente da grade que mesmo uma solução com poucas réplicas causaria uma sobrecarrega ao sistema, podendo ser utilizado nesses casos *checkpointing*. A desvantagem do uso da técnica de *checkpointing* é que ela naturalmente produz uma sobrecarga sobre o tempo de execução das tarefas, uma vez que é necessário parar o processo a cada vez que é salvo o estado do progresso da tarefa. Contudo, em ambientes com poucos recursos, o atraso causado pelo *checkpointing* ainda é menor que o da replicação.

As abordagens de replicação e *checkpointing* apresentam vantagens em diferentes condições do ambiente do ambiente de execução da grade. Entretanto o ambiente de grades é altamente dinâmico, sofrendo frequentes mudanças no decorrer de sua execução. Se essas técnicas de tolerância a falhas forem utilizadas de modo estático, ou seja, não se adequarem dinamicamente às mudanças sofridas pelo ambiente, elas podem causar o desequilíbrio do sistema, degradando seu desempenho e eficiência.

Levando-se em consideração o exposto, que evidencia as desvantagens de se adotar uma abordagem estática para prover tolerância a falhas na execução de aplicações em *desktop grids*, apresentamos, neste trabalho, uma abordagem autonômica que utiliza as vantagens do uso das técnicas de replicação e *checkpointing* e que tem como base dois níveis de adaptação. No primeiro nível, são levadas em consideração

as variações do ambiente de execução para as quais os ajustes em parâmetros usados pela técnica de tolerância a falhas em uso sejam suficientes para manter o sistema em equilíbrio com seus objetivos. O segundo nível de adaptação lida com variações mais significativas do ambiente de execução que provocam uma degradação de desempenho da abordagem de tolerância a falhas em uso que não podem ser contornadas apenas através de adaptações paramétricas, exigindo uma reconfiguração estrutural da técnica de tolerância a falhas, substituindo-se a abordagem em uso por outra. As heurísticas utilizadas nestes dois níveis de adaptação foram baseadas no trabalho de Chtepen [5], estendendo-se suas regras, como o objetivo de se obter um melhor desempenho com aplicações do tipo *bag-of-tasks*, que compreende a classe de aplicações mais utilizada em *desktop grids* e ajustando seus parâmetros para uso eficiente nesta classe de grades computacionais. Também tratamos um efeito que causa repetidas reconfigurações na técnica de tolerância a falhas, o qual chamamos de efeito *ping-pong*.

4.2 Primeiro Nível: Adaptação Paramétrica Usando *Checkpointing*

A técnica de *checkpointing* naturalmente produz uma sobrecarga sobre o tempo de execução das tarefas, uma vez que é necessário parar o processo a cada vez que é salvo o estado do progresso da aplicação. A periodicidade estática nas tomadas dos *checkpoints* das tarefas torna esta abordagem não muito vantajosa quando levamos em consideração a volatilidade dos recursos em *desktop grids*. Um recurso é dito volátil quando, em seu histórico de funcionamento, ele apresenta um grande número de falhas ou, em se tratando de *desktop grids*, o proprietário do recurso não o disponibiliza com muita frequência para executar as tarefas da grade. Já os recursos estáveis são aqueles que são menos suscetíveis a falhas e, portanto, estão na maior parte do tempo disponíveis ou executando as computações submetidas à grade.

Nas abordagens estáticas, o *checkpointing* pode ser configurado com intervalos curtos para evitar que as tarefas, ao serem recuperadas das falhas, executem o mínimo possível para retornar ao mesmo estado de quando falharam. Contudo, essa configuração irá executar muitas vezes o procedimento que para e salva o estado

da tarefa, prolongando o tempo de sua conclusão. Quando o ambiente se torna estável em relação à disponibilidade dos recursos, essa sobrecarga é desnecessária. Por outro lado, em ambientes voláteis, a definição de longos intervalos para a tomada do *checkpointing* pode levar a muita reexecução de código da aplicação e, dependo do grau de volatilidade dos recursos do ambiente, o *checkpointing* das tarefas pode nem chegar a ser feito. Se conhecermos o grau de volatilidade dos recursos da grade, podemos estimar intervalos de *checkpointing* que se ajustam melhor a cada situação.

A fim de reduzir a sobrecarga desnecessária causada pelo *checkpointing*, o modelo utilizado em nossa abordagem ajusta os intervalos entre os *checkpoints* de cada tarefa de acordo com a volatilidade do recurso em que está executando. Então, se os processos executam em recursos estáveis, esse ajuste é feito de forma que os intervalos entre *checkpoints* são alargados e, portanto, é reduzida a quantidade de vezes que são feitos. De outra forma, se os processos executam em recursos voláteis, os intervalos são reduzidos e, consequentemente, uma maior quantidade de *checkpoint* será feita, garantindo que sejam salvos estados mais próximos do estado de execução do momento da falha.

Nossa abordagem configura o intervalo de periodicidade por meio de regras baseadas em duas estimativas: uma previsão sobre o tempo que irá ocorrer a próxima falha do recurso (`failPrediction`) e outra sobre o tempo de conclusão da tarefa (`taskConcTimePrediction`). A primeira, `failPrediction`, é calculada obtendo-se, primeiramente, o instante em que ocorreu a última falha do recurso. Após isso, calculamos o MTBF do recurso e adicionamos esse valor ao instante da sua última falha. Já o `taskConcTimePrediction`, pode ser estimado utilizando-se algoritmos de predição [46] que seguem basicamente duas abordagens. Na primeira abordagem, calcula-se a estimativa do tempo de execução da aplicação baseado no registro de execuções anteriores da mesma ou de aplicações semelhantes. A segunda abordagem é baseada no conhecimento do modelo de execução da aplicação. O código da aplicação é analisado, estimando-se o tempo de execução de cada tarefa de acordo com a capacidade dos recursos da grade [12]. Para estimar a capacidade de processamento de um recurso, pode ser tomada como base uma medição do hardware (como CPU e memória) ao realizar o processamento de um dado tipo de aplicação. Um *benchmark* analítico poderia ser usado como forma de ordenar os recursos de acordo com a sua eficiência para executar um determinado tipo de código computacional [16].

O algoritmo da Figura 4.1 descreve as condições em que o intervalo entre *checkpointing* de cada tarefa aumenta ou diminui na estratégia autonômica proposta neste trabalho.

```
1  para cada recurso faça
2      para cada tarefa em execução no recurso faça
3          se estRemExecTime < resMTBF então
4              newInterval = 2 * prevInterval;
5          senão se estRemExecTime >= resMTBF então
6              newInterval = prevInterval/2;
7          fim
8          se newInterval < minInterval então
9              newInterval = minInterval;
10         fim
11         se newInterval > maxInterval então
12             newInterval = maxInterval;
13         fim
14     fim
15 fim
```

Figura 4.1: Heurística de adaptação paramétrica usando *checkpointing*.

Quando o tempo restante estimado para o término da tarefa for menor que o tempo médio entre falhas do recurso ($estRemExecTime < resMTBF$), há menor probabilidade de ocorrer uma falha com esse recurso enquanto a tarefa estiver executando nele. Baseado nessa probabilidade, reduzimos a frequência com que se faz o *checkpoint* dessa tarefa, através do incremento do intervalo que estabelece essa periodicidade, a fim de reduzir o *overhead* introduzido por esta técnica. Caso a situação seja a inversa, quando o tempo estimado para o restante da execução da tarefa for maior ou igual ao tempo médio entre falhas do recurso ($estRemExecTime \geq resMTBF$), existe uma probabilidade maior de que ocorra uma falha enquanto a tarefa estiver executando e, por isso, o intervalo da periodicidade tem seu valor subtraído, fazendo-se *checkpointing* com mais frequência. Para evitar que o intervalo cresça

ou diminua indefinidamente, deve-se estabelecer valores máximos ($maxInterval$) e mínimos ($minInterval$) para o intervalo.

4.3 Primeiro Nível: Adaptação Paramétrica Usando Replicação

Na técnica de replicação um dos pontos mais importantes para se alcançar um bom desempenho é a definição da quantidade de réplicas a ser utilizada. Essa decisão é afetada diretamente pela quantidade de recursos disponíveis para processar essas réplicas. Quanto mais recursos houver na grade, melhor será para a replicação. Quando utilizamos uma abordagem estática de replicação, a configuração da quantidade necessária de réplicas se torna uma decisão difícil. Se escolhermos instanciar um grande número de réplicas para aumentar a probabilidade das tarefas finalizarem com sucesso, corremos o risco de sobrecarregar a grade executando réplicas, enquanto seus nós poderiam estar processando outras tarefas. Por outro lado, se escolhermos instanciar poucas réplicas, maior será a probabilidade de nenhuma delas concluir sua tarefa computacional.

Dessa forma, o principal fator de nossa abordagem, no que diz respeito à tomada de decisão quanto a reconfiguração da quantidade de réplicas utilizadas, é a quantidade de recursos ocupados da grade. Os recursos da grade podem ser divididos em ativos e inativos. Os recursos ativos são aqueles que estão executando alguma tarefa ou estão à espera de tarefas, prontos para executá-las. Os inativos, por sua vez, estão indisponíveis por algum motivo, seja por falha ou porque seus proprietários os tenham retirado da grade por qualquer razão. Os recursos ativos são ditos livres quando não estão executando nenhuma tarefa e caso contrário, são ditos ocupados. A heurística empregada na nossa abordagem utiliza como base o percentual de ocupação da grade em relação aos recursos ativos da grade.

Outro fator que pode influenciar nessa decisão está relacionado à taxa de falhas dos recursos da grade, dado que quanto maior a quantidade de falhas, menores serão as chances de uma das réplicas terminar sua execução com sucesso. No entanto, preferimos não incluir a taxa de falhas na tomada de decisão, porquanto que

a ocorrência dessas falhas, consequentemente, provoca uma redução na quantidade de recursos na grade, e dessa forma entram indiretamente na regra da tomada de decisão.

A abordagem autonômica utiliza diferentes faixas de valores (intervalos) do percentual de recursos ocupados para realizar as adaptações paramétricas. À medida que esse percentual cresce ou diminui, mudando de faixa, uma quantidade diferente de réplicas deve ser instanciada para cada tarefa no processo de replicação, de modo que seja mais bem adequada ao contexto do ambiente de execução da grade. A Figura 4.2 exemplifica a heurística, que contém um conjunto de condições para verificar entre qual intervalo se encontra o atual valor do percentual de recursos ocupados.

```
1 se percResOccup >= 0 and percResOccup < 5 então
2 |   numRep = 4;
3 senão se percResOccup >= 5 and percResOccup < 10 então
4 |   numRep = 3;
5 senão se percResOccup >= 10 and percResOccup < 30 então
6 |   numRep = 2;
7 fim
```

Figura 4.2: Heurística de adaptação paramétrica usando replicação.

No exemplo da Figura 4.2 definimos 3 condições. A primeira, corresponde a tomada de decisão que ajusta o número de réplicas para quatro ($numRep = 4$), quando o percentual de recursos ocupados ($percResOccup$) estiver na faixa de valores que pode expressa por um intervalo semi-aberto entre zero e cinco por cento ($[0, 5)$). Essa faixa indica que há muitos recursos livres e, portanto, deve ser bom utilizar bastante réplicas. A segunda e terceira condições, indicam que esse percentual está mais alto e, portanto, há uma redução gradual do número de réplicas. Essa redução ocorre em função do ajuste necessário da técnica de replicação para evitar que o sistema seja rapidamente sobrecarregado. Quando os valores superam esses intervalos, significa que a técnica de replicação não pode ser mais ajustada e necessita ser substituída por outra técnica, conforme veremos na reconfiguração estrutural da Seção 4.4.

Nessa abordagem, o usuário pode definir seus próprios intervalos, bem como a quantidade de réplicas que serão aplicadas na replicação quando o percentual

de recursos se encontrar em um desses intervalos. Entretanto, essas configurações devem ser feitas sempre dosando a quantidade de réplicas à quantidade de recursos disponíveis na grade.

A técnica de replicação baseia-se na probabilidade de que quanto maior o número de réplicas, maior as chances de sucesso na conclusão da tarefa. Contudo, essa técnica não oferece total garantia de que pelo menos uma das réplicas irá concluir. Nessa abordagem, são criadas réplicas para cada tarefa que é submetida e cada tarefa possui, portanto, um conjunto de réplicas em execução. No entanto, se todas as réplicas desse conjunto falharem, a tarefa é dada como falha. Por isso, elaboramos uma técnica híbrida que combina o uso das técnicas de replicação e *checkpointing* para aumentar as chances de sucesso na replicação. Nesta técnica, quando das n réplicas de um conjunto $n-1$ réplicas falham, sobrando uma única réplica em execução, a estratégia autonômica deve fazer o *checkpoint* da tarefa remanescente e persisti-lo em um armazém estável. Se após isso, essa tarefa falhar, ela será ressubmetida mais uma vez apenas para um dos recursos da grade, o qual é escolhido de acordo com a heurística de escalonamento adotada, recuperando-se o último estado salvo.

Na prática, para que essa estratégia seja implementada, deve-se possuir um controle sobre os conjuntos de réplicas de cada tarefa. Esse controle também deve possuir informações sobre os recursos onde essas réplicas foram escalonadas para executar. Além disso, deve-se também monitorar todas as ocorrências de falhas de réplicas na grade. Esse monitoramento pode ser feito utilizando-se um componente capaz de detectar a ocorrência de falhas. Sempre que uma falha for detectada, esta deve ser notificada ao mecanismo, o qual verificará se a falha resultou em um conjunto de réplicas com uma única réplica remanescente. Quando essa situação acontece, deve-se fazer o *checkpointing* da tarefa a fim de proteger a execução da mesma.

As tarefas também devem estar previamente habilitadas para iniciar o processo de *checkpointing* a qualquer momento. Isso implica que elas devem ter uma instrumentação adequada antes de serem submetidas para execução. Em alguns *middleware* de grade essa instrumentação é feita através da recompilação do código da aplicação junto com bibliotecas que são disponibilizadas para esse fim. No *middleware* do InteGrade[14], por exemplo, esse tipo de instrumentação insere código na aplicação que permite parar a execução da aplicação e coletar o seu estado a qualquer momento. Na verdade, essa instrumentação faz com que quando a aplicação seja iniciada, uma

thread seja instanciada, a qual recebe o sinal do mecanismo de *checkpointing* para parar e coletar estado de execução da aplicação [15].

4.4 Segundo Nível de Adaptação: Reconfiguração Estrutural

No primeiro nível de adaptação, nossa abordagem busca ajustar parâmetros da técnica de tolerância a falhas em uso pelo *middleware* da grade de forma a manter o equilíbrio do sistema. No entanto, mudanças no ambiente de execução podem tornar estas ações insuficientes para a manutenção desse equilíbrio. Neste caso, é necessária a realização de uma reconfiguração estrutural, substituindo-se a técnica de tolerância a falhas em uso. Para isto, o modelo proposto possui um segundo nível de adaptação que prevê a troca da abordagem de tolerância a falhas entre duas possibilidades: o uso de replicação e a adoção de *checkpointing*. O uso da técnica de *checkpointing* impõe um custo adicional ao tempo de execução da aplicação em decorrência das paradas para realizar a obtenção e persistência do estado de execução das tarefas. A técnica de replicação não impõe este custo, mas requer maior uso de recursos da grade, podendo atrasar a execução de novas tarefas submetidas à mesma, dependendo da disponibilidade de seus recursos.

A abordagem autonômica leva em consideração a possibilidade de alternar dinamicamente entre as técnicas de replicação e *checkpointing*, de modo que possa explorar as vantagens dessas técnicas de acordo com o ambiente que for mais favorável para utilização de cada uma delas. O fator utilizado para a tomada de decisão neste nível de adaptação é o percentual de ocupação dos recursos. As regras que utilizam esse percentual são uma extensão das regras que são utilizadas para ajustar o número de réplicas na técnica de replicação. A abordagem inicia utilizando replicação quando o percentual de ocupação dos recursos é igual a zero. À medida que os recursos vão sendo ocupados com a execução de tarefas e réplicas, esse percentual cresce e os ajustes paramétricos na técnica de replicação passam a não produzir mais os efeitos desejados, tornando-se necessário a substituição desta técnica pelo *checkpointing*. Quando esses recursos vão sendo liberados para novas execuções, ou mais recursos vão se registrando na grade, o percentual diminui, tornando o ambiente

novamente favorável à utilização da técnica de replicação. Exemplificamos na Figura 4.2 a heurística que descreve essas regras.

```
1 se percResOccup >= 0 and percResOccup < 30 então
2 |   useReplication();
3 senão se percResOccup >= 30 and percResOccup <= 100 então
4 |   useCheckpointing();
5 |   se percResOccup > 80 então
6 |   |   killReplicas();
7 |   fim
8 fim
```

Figura 4.3: Heurística de adaptação paramétrica usando replicação.

Quando o percentual de recursos ocupados ($percResOccup$) estiver na faixa de valores expressa por um intervalo semi-aberto entre zero e trinta por cento ($[0, 30)$), a abordagem utiliza a técnica de replicação. A quantidade de réplicas será determinada conforme a heurística descrita na Seção 4.3. Quando o percentual de ocupação de recursos ($percResOccup$) alcançar valores entre trinta e cem por cento, a técnica que deve ser empregada será o *checkpointing*. Uma terceira condição nessa regra determina que se esse percentual for superior a oitenta por cento ($percResOccup >= 80$) a estratégia deve optar, além de uso de *checkpointing*, também pelo cancelamento de réplicas ($killReplicas()$). Descreveremos como é feito esse cancelamento na Seção 4.5.

Nessa abordagem, o usuário pode definir os intervalos para definir o uso de cada uma das estratégias. Entretanto, deve ser observada a compatibilidade com os intervalos definidos para a heurística das adaptações paramétricas na replicação e no cancelamento de réplicas.

Quando a técnica de *checkpointing* é substituída pela replicação, a abordagem autonômica realiza a configuração para as novas submissões de usuários utilizarem esta última. As tarefas que estavam anteriormente sob o regime do *checkpointing* param de realizar a salva periódica de estado, pois o *checkpointing* de várias tarefas atrasa o tempo de conclusão da estratégia. Caso alguma dessas tarefas

sofram alguma falha, ela terá de ser executada a partir do último *checkpoint* salvo antes da troca da estratégia.

Quando substituímos a técnica de replicação por *checkpointing*, a abordagem autonômica realiza a configuração para as novas submissões de usuários utilizarem a técnica de *checkpointing*. As tarefas que estavam, anteriormente à reconfiguração, utilizando replicação continuam executando com réplicas até que o limite definido para o cancelamento de réplicas seja alcançado, o qual será visto na Seção 4.5. A partir desse ponto, essas tarefas deixam de executar com réplicas e passam a adotar a técnica de *checkpointing*.

Na prática, uma possível implementação deste mecanismo de adaptação poderia empregar um sistema com múltiplas *threads*, conforme é utilizado por Guimarães em [25], o qual coordenaria a tolerância a falhas na execução de cada tarefa. O mecanismo autonômico seria responsável por realizar as adaptações nessas *threads*.

4.5 Cancelamento de Réplicas

Apesar de a estratégia proposta evitar o cancelamento de réplicas, torna-se necessário realizar esse procedimento quando, após ter sido efetuada a troca da técnica de replicação por *checkpointing*, o percentual de recursos ocupados da grade continuar crescendo e ultrapassar um determinado limiar. Esse limiar é responsável por evitar que a grade chegue à ocupação total dos recursos, garantindo uma reserva deles para as novas requisições de usuários. Se a grade atingisse o uso total de seus recursos, as próximas tarefas que chegassem teriam que esperar que as réplicas das tarefas em execução concluíssem, sofrendo um atraso no tempo total de conclusão da aplicação. Com o objetivo de manter um conjunto mínimo de recursos livres para evitar que a computação de novas requisições sejam atrasadas, este modelo adota uma política de cancelamento de réplicas baseado no percentual de ocupação dos recursos. Quando esse percentual atinge um determinado limiar, ele inicia o procedimento de cancelamento de modo que sejam liberados recursos na grade.

Observamos, entretanto, que esse procedimento deve ser feito de forma segura, evitando-se que o cancelamento de uma réplica venha causar a falha de toda a aplicação. Aplicações do tipo *bag-of-tasks* são compostas por várias tarefas que não

se comunicam entre si. Apesar dessa independência, os resultados produzidos por todas as tarefas constituem a solução de um único problema e na maioria dos casos os usuários precisam do conjunto completo de soluções das tarefas para poder realizar o pós-processamento ou a análise dos resultados [53]. Portanto, para tratar corretamente a tolerância a falhas de aplicações *bag-of-tasks*, deve ser considerado todo o conjunto de resultados e não apenas o de uma tarefa em particular ou de um grupo delas [2], [52]. Para isso, a abordagem deve conhecer cada aplicação que está executando, seu conjunto de tarefas, e de cada tarefa, o conjunto das réplicas que foram instanciadas durante a replicação. Tendo esse conhecimento, pelo menos uma réplica de cada tarefa deve ser selecionada para continuar em execução, enquanto as outras podem ser canceladas para liberar os recursos.

Outro critério que deve ser levado em consideração nessa tomada de decisão é o tempo restante para a conclusão das réplicas, pois deve ser dada prioridade para as réplicas que estiverem mais próximas do fim de sua execução. Para obter essa informação, pode ser feito uma estimativa baseada na previsão do tempo total de execução da tarefa para o recurso em que está executando e subtrair o tempo decorrido no momento do cancelamento de réplicas. Com base nessa estimativa, as réplicas mais próximas da conclusão seriam poupadas e as demais sofreriam o cancelamento. A heurística que descreve a tomada de decisão desta abordagem está expressa no algoritmo da Figura 4.4.

Quando o percentual de ocupação dos recursos for maior ou igual ao limite estipulado, que no exemplo foi de oitenta por cento ($percResOccup >= 80\%$), deve ser iniciado o procedimento de cancelamento de réplicas (linha 1) . Esse procedimento inicia selecionando aleatoriamente as aplicações que tem tarefas executando com réplicas (linha 3). Para cada tarefa da aplicação, pega-se a lista de réplicas (linhas 4 a 6). Essa lista é ordenada de modo que as réplicas mais próximas do final de sua execução estejam no início da lista (linha 7). A réplica mais avançada é retirada da fila e feita o seu *checkpoint* (linhas 8 e 9). Além da réplica mais avançada, a abordagem permite que outras réplicas possam ser mantidas. Para isso, deve ser configurada a quantidade de réplicas que devem ser poupadas. No exemplo, foram poupadas mais de uma réplica (linhas 10 a 13). Entretanto, apenas a réplica mais avançada passará a ser regida pela técnica de *checkpointing*. Após esse passo, a abordagem deve enviar um sinal de cancelamento para o recurso onde as demais réplicas estão executando (linhas 14 a 17).

```
1  se percResOccup > 80 então
2      repita
3          app = getNextAppExecutingWithReplicas();
4          enquanto app.hasNextTask() faça
5              task = app.getNextTask();
6              replicasList = task.getReplicas();
7              sortReplicasByRemainingTime(replicasList);
8              mostAdvancedReplica = replicasList.removeFirst();
9              takeCheckpoint(mostAdvancedReplica);
10             enquanto replicasList.hasNextReplica() and
               numRepKeptOn < 1 faça
11                 savedReplica = replicasList.removeFirst();
12                 numRepKeptOn++
13             fim
14             enquanto replicasList.hasNextReplica() faça
15                 replica = replicasList.getNextReplica();
16                 cancel(replica);
17             fim
18             percResOccup = getNewPercResOccup();
19             se percResOccup < 65 então
20                 stop;
21             fim
22         fim
23     até hasNoMoreAppExecutingWithReplicas() ;
24 fim
```

Figura 4.4: Heurística para o cancelamento de réplicas.

Após o cancelamento das réplicas é verificado se o percentual de ocupação reduziu para valores inferiores a sessenta e cinco por cento e, em caso positivo, o procedimento deve ser parado (linhas 18 a 21). Caso contrário, todo o procedimento será repetido até que todas as aplicações com réplicas tenham sido tratadas (linha 23).

Nessa abordagem, o usuário pode definir um valor para o limiar que irá definir o início do procedimento do cancelamento de réplicas. Também é permitido definir o valor de parada do procedimento, bem como a quantidade de réplicas que devem ser mantidas.

4.6 Efeito *Ping-Pong*

Um dos problemas em se estabelecer um limiar para realizar reconfigurações em qualquer estratégia adaptativa é que as medições do contexto podem oscilar em torno desse limite. Esse comportamento pode levar o mecanismo adaptativo a efetuar diversas transições entre as configurações que se deseja ajustar, oscilando rapidamente entre elas. Um comportamento semelhante acontece na computação móvel, quando um determinado dispositivo móvel atravessa os limites de cobertura dos sinais de duas torres de transmissão. Chama-se *handoff* o mecanismo que realiza a transferência do dispositivo de uma área de cobertura para outra. Esse mecanismo usualmente é baseado na intensidade do sinal das duas torres percebido pelo dispositivo móvel. Aquela que tiver valor mais alto irá determinar em qual área o dispositivo irá ingressar. Quando o dispositivo se desloca na região limite entre essas duas áreas, a intensidade do sinal pode oscilar, ora uma sendo mais intensa, ora a outra, causando trocas contínuas. Esse fenômeno é conhecido como efeito *ping-pong*[66]. Existem dois possíveis casos que foram levados em consideração na abordagem autonômica proposta neste trabalho para evitar o este efeito: o primeiro na adaptação estrutural e o segundo no cancelamento de réplicas.

A adaptação estrutural de nossa abordagem está baseada em um limite, que na Figura 4.3 foi definido em 30% para os valores do percentual de ocupação dos recursos. Qualquer medição que oscile em torno desse limite poderá causar esse efeito *ping-pong* na reconfiguração das técnicas de replicação e *checkpointing*. Para que a reconfiguração estrutural de nossa abordagem não produza esse efeito *ping-pong*, adicionamos uma margem de tolerância, definida em dez por cento (10%), para o limiar que determina quando uma troca da técnica de *checkpointing* para replicação deve ser realizada. Dessa forma, se o percentual alcançar 30%, a estratégia autonômica passará a utilizar *checkpointing*, e caso logo em seguida a disponibilidade de recursos retroceder

para 29%, a estratégia autonômica continuará usando *checkpointing*, a menos que esse percentual continue caindo até atingir os 20% de ocupação dos recursos da grade.

O cancelamento de réplicas também poderia causar o efeito *ping-pong*, dado que à medida que as réplicas são canceladas, mais recursos vão sendo liberados, reduzindo o percentual de ocupação. Essa redução pode fazer com que a estratégia autonômica volte a se configurar, retornando ao uso da técnica de replicação. Por esse motivo, no próprio procedimento de cancelamento de réplicas foi definido um momento de parada com o objetivo de evitar que esse efeito aconteça. Como pode ser visto na linha 19 do algoritmo da Figura 4.4, esse momento consiste na definição de um limite para o percentual de ocupação de recursos, de modo que o cancelamento de réplicas é parado sempre que o percentual de ocupação dos recursos atinge valores abaixo de 65%.

4.7 Conclusão

Este capítulo apresentou a estratégia autonômica para prover tolerância a falhas na execução de aplicações em ambientes de grades oportunistas proposta neste trabalho. Vimos que o ambiente de grades é muito dinâmico e que essa dinamicidade influencia diretamente nas técnicas de tolerância a falhas utilizada pelo *middleware* da grade. Por esse motivo, propusemos um modelo que pudesse aproveitar as vantagens de cada técnica em face às variações de contexto, realizando reconfigurações no nível paramétrico e estrutural da estratégia de tolerância a falhas adotada pelo *middleware* da grade. No nível paramétrico, são feitas reconfigurações de parâmetros em uma determinada técnica, a fim de adaptá-la ao contexto de execução da grade. No nível estrutural, substituímos uma técnica por outra, quando as configurações paramétricas não forem suficientes para manter o equilíbrio do sistema devido à degradação de seu desempenho.

Utilizamos o cancelamento de réplicas para liberar recursos quando os níveis de ocupação dos recursos na grade chegam próximos da ocupação total. Contudo, esse procedimento é realizado de modo que as réplicas de tarefas *bag-of-task* sejam canceladas de forma segura, evitando que o cancelamento de uma réplica possa comprometer o resultado de toda a aplicação. Esse cancelamento também visa

cancelar as réplicas mais lentas e manter as réplicas que estão mais avançadas em sua computação. Além disso, nossa estratégia provê um tratamento para evitar o efeito *ping-pong*, causado pelas oscilações entre as técnicas durante as reconfigurações estruturais desta abordagem.

Para analisar a eficiência desta abordagem, utilizamos um ambiente de grade simulado e implementamos os componentes da estratégia autonômica segundo o modelo MAPE-K que realizam as fases de monitoramento, análise, planejamento e execução, utilizando as heurísticas descritas como regras que formam a base de conhecimento do gerente autonômico. A implementação deste modelo e a sua avaliação serão descritas no capítulo seguinte.

5 Avaliação da Abordagem Autonômica

Este capítulo mostra uma avaliação formal do modelo da abordagem descrita no capítulo anterior. Para esta avaliação, utilizamos técnicas de simulação devido à dificuldade de avaliar o comportamento de nossa abordagem num ambiente de grade larga escala, envolvendo um grande número de recursos e de usuários. Pois através da simulação é possível realizar experimentos e repeti-los dentro de um ambiente controlado, facilitando a análise e compreensão dos resultados. Experimentos em ambientes de grades reais também podem ser bem mais custosos, uma vez que podem demorar várias semanas ou até meses, alocando grandes quantidades de recursos.

Em nossas simulações utilizamos o AGST (*Autonomic Grid Simulation Tool*) [17]. Esta ferramenta foi desenvolvida por nosso grupo de pesquisa no Laboratório de Sistemas Distribuídos da Universidade Federal do Maranhão e tem como principal objetivo oferecer suporte a modelagem de sistemas e avaliação de mecanismos e componentes autonômicos para grades computacionais. Na implementação da abordagem proposta, vários componentes do AGST foram utilizados e, por isso, iremos fazer uma breve apresentação dessa ferramenta, abordando seu funcionamento, seus componentes e sua utilização. Através do AGST pudemos criar um modelo que representasse a abordagem autonômica do nosso trabalho, criar os cenários para o ambiente de grades e realizar simulações que permitissem avaliar melhor o desempenho da abordagem proposta neste trabalho. Dessa forma, descreveremos esse modelo, os cenários das simulações realizadas e a análise e discussão dos resultados obtidos.

5.1 O AGST

Nos últimos anos vários simuladores foram desenvolvidos como o objetivo de avaliar o desempenho de modelos computacionais complexos elaborados para o ambiente de grades computacionais. Como exemplo, podemos citar vários

simuladores como: o GridSim [4], SimGrid [29], Alea [41], GSSIM [43], DSiDE [6], and OGST [13]. Embora esses simuladores permitam a implementação e avaliação de comportamento adaptativo em grades computacionais, nenhum deles foi desenvolvido visando esse tipo modelagem e não oferecem recursos que facilitem a implementação e avaliação deste tipo de comportamento.

Como visto na Seção 2.2.1, o modelo mais utilizado no desenvolvimento de componentes de software autonômico é a arquitetura MAPE-K proposta pela IBM [11]. O qual é dividido em dois principais componentes: o gerente autonômico e o elemento gerenciado. O elemento gerenciado corresponde ao sistema ou a um componente que será adaptado dinamicamente. O gerente autonômico executa as funções que envolvem a lógica de adaptação do elemento gerenciado: monitoramento, análise, planejamento e execução. O modelo MAPE-K possui dois componentes que têm acesso direto ao elemento gerenciado, que são os sensores e os atuadores. Os sensores são responsáveis por coletar as informações do elemento gerenciado e enviá-las para os monitores onde são interpretadas, pré-processadas e filtradas antes de serem enviadas para os analisadores. Na próxima fase do ciclo, análise e planejamento, são produzidos os planos de ações, que consistem de um conjunto de ações de adaptações que serão executadas pelo executor. Os atuadores são componentes que permitem que os gerentes autonômicos façam ajustes no elemento gerenciado. A decisão de qual ação de adaptação deve ser aplicada em uma dada situação requer uma representação do conhecimento do sistema computacional e do ambiente.

O AGST[1] constitui uma ferramenta de simulação que visa fornecer suporte a modelagem de sistemas e componentes autonômicos de grades computacionais para simulação e avaliação de diversas características, comportamentos e funcionalidades que estão presentes nesse tipo de ambiente, como por exemplo: o monitoramento do ambiente da grade, a análise de informações de contexto, planejamento de reconfiguração e a implementação de estratégias que permitam adaptação dinâmica dos componentes da grade. O ASGT foi construído baseado no OGST (*Opportunistic Grid Simulation Tool*) [22], no GridSim *toolkit* [4]e no *framework* SimJava [29] como demonstra a Figura 5.1, que ilustra a arquitetura do AGST.

[1]Mais informações sobre o AGST estão disponíveis em `www.lsd.ufma.br/\textasciitildeagst`

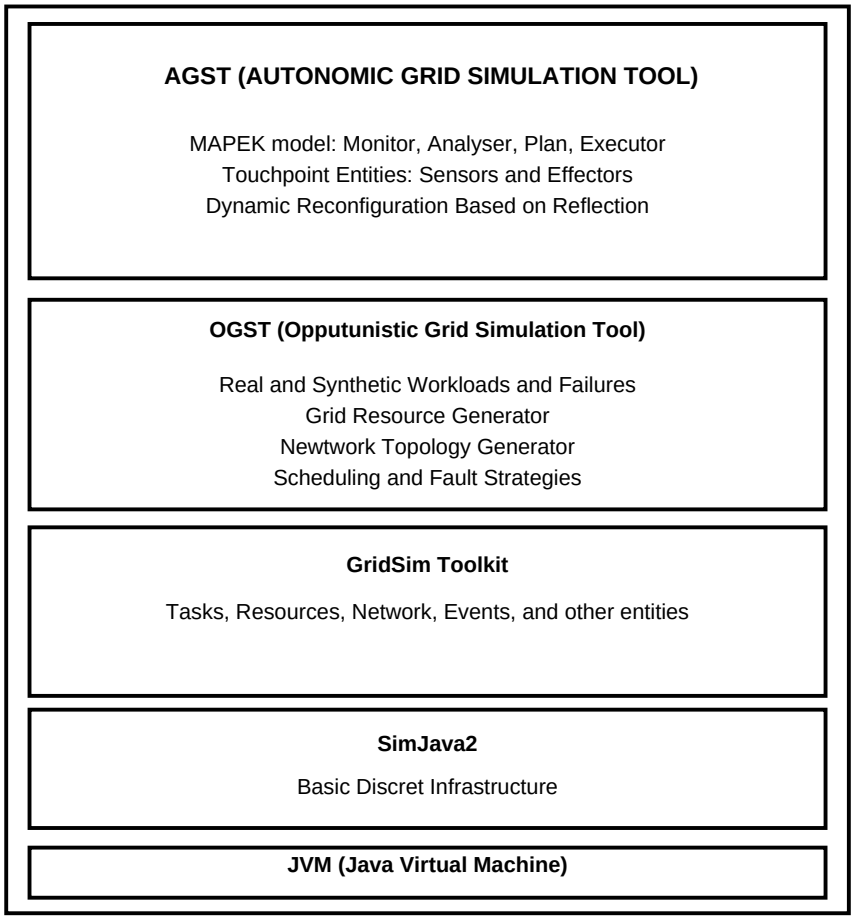

Figura 5.1: Camadas de software do AGST.

O SimJava é um *framework* de simulação que modela sistemas distribuídos como um conjunto de entidades que se comunicam entre si através de eventos. O controle do relógio da simulação é feito baseado na ocorrência de eventos. Cada evento é adicionado na fila do simulador de acordo com sua ordem cronológica. Dessa forma, o SimJava permite que na simulação o tempo seja incrementado em saltos, se antecipando em relação ao tempo real de execução. O GridSim é uma plataforma de software que permite aos usuários modelar e simular características dos recursos de grades e de redes com diferentes configurações. O OGST oferece um conjunto de ferramentas para geração de nós e aplicações para grades. Além disso, o OGST fornece também uma arquitetura básica de grades oportunistas estruturada como mostra a Figura 5.2. Todas essas ferramentas foram desenvolvidas utilizando a plataforma Java e, portanto, dependem de uma JVM (*Java Virtual Machine*) para execução.

A Figura 5.2 ilustra os principais componentes do OGST. O `Grid Feature Generator` (GFG) é um componente usado para definir um ambiente de grade simulado (nós da grade) e as aplicações a serem executadas com suas respectivas taxas

de chegada. O OGST atualmente permite a simulação de aplicações regulares e *bag-of-tasks*. Para cada tarefa de uma aplicação deve-se prover seu tamanho, definido em milhões de instruções. O `User Application Submission Tool` (UAST) representa o usuário da grade e é o componente responsável pela submissão da aplicação, o qual recebe uma notificação sobre sua conclusão.

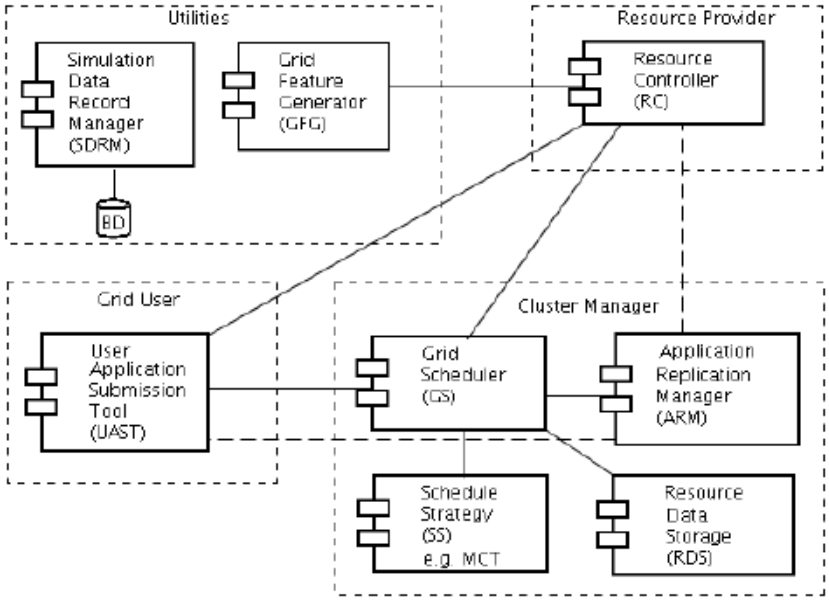

Figura 5.2: Arquitetura do AGST.

O `Grid Scheduler` (GS) recebe submissões para execução de aplicações do UAST e executa o algoritmo de escalonamento, encapsulado no componente `Scheduling Strategy` (SS). A estratégia de escalonamento usa dados sobre a disponibilidade dos recursos da grade providos pelo componente Resource Data Storage (RDS). Cada tarefa da aplicação é então mapeada para a execução em um nó específico da grade. Cada nó da grade executa um `Resource Controller` (RC), responsável pela instanciação e execução das tarefas das aplicações escalonadas para o nó, mantendo uma lista de tarefas esperando pela execução. Ele também é responsável pela simulação da variação da carga do recurso local. O `Simulation Data Record Manager` (SDRM) usa uma base de dados relacional para o armazenamento dos dados da simulação coletados, tais como os tempos de inicio e conclusão das tarefas. O componente RDS é também responsável pela simulação da ocorrência de falhas e recuperação de nós.

O OGST permite ainda a simulação da execução de aplicações com replicação, técnica comumente usada em ambientes de grades oportunistas a fim

de contornar eventuais falhas de nós. O `Application Replication Manager` (ARM) é o componente responsável pelo gerenciamento da execução de réplicas.

Para implementar o modelo MAPE-K no simulador AGST, foram criadas as seguintes entidades: `AbstractSensor`, `AbstractMonitor`, `AbstractAnalyser`, `Executor`, e `Effector`. Essas entidades são responsáveis pelas funções de monitoramento, análise e planejamento, controle e execução do gerenciamento autonômico. Essas entidades se comunicam entre si através de eventos. Cada entidade executa uma função definida de acordo com o ciclo autonômico da arquitetura MAPE-K, formando gerente autonômico. Além disso, o AGST também permite criar mais de um ciclo de gerenciamento autonômico de modo que possa ser criado um gerenciador global, capaz de controlar o comportamento autonômico de todo o sistema, bem como ativar ou desativar os gerentes autonômicos específicos, que atuam sobre determinados elementos do sistema. Por fim, o AGST permite que qualquer entidade da grade possa ser um recurso gerenciado.

Além disso, o AGST provê ferramentas para modelagem dos recursos que compõem a grade simulada e suas interconexões de rede, o uso de arquivos de *trace* de carga de trabalho (*workload*) segundo o padrão GWF (*Grid Workload Format*) [2] [35], especificado pelo GWA (*Grid Workload Archive*), o uso de arquivos de *trace* contendo falhas, conforme definido pelo padrão FTA (*Failure Trace Archive*) [3] [42] e uma implementação do modelo de *workload* criado por Lublin e Feitelson [47] para modelar a carga de trabalho dos recursos da grade.

5.2 Implementação da Abordagem

Como visto no Capítulo 4, nossa abordagem autonômica para tolerância a falhas na execução de aplicações em grades computacionais é composta por dois níveis. No primeiro nível, realiza adaptações paramétricas durante a execução da técnica de replicação ou adaptações paramétricas no uso da técnica de *checkpointing*. Enquanto que no segundo nível, quando os ajustes de parâmetros não são suficientes, realiza-se adaptações estruturais, substituindo-se a técnica que sofreu perda de performance por outra mais adequada ao tipo de ambiente. Para a implementação

[2] `http://gwa.ewi.tudelft.nl/pmwiki/`

[3] `http://fta.inria.fr/`

deste modelo foram definidos três ciclos de gerenciamento autonômico, conforme podemos observar na Figura 5.3. Cada um desses ciclos possui um gerente autonômico que atua sobre o seu elemento gerenciado, os quais são diversos componentes do *middleware* da grade. Dois desses ciclos são responsáveis por realizar adaptações paramétricas: um deles controla o ciclo de gerenciamento autonômico do *checkpointing* e o outro o ciclo da replicação. O terceiro ciclo é responsável por realizar as adaptações estruturais, coordenando o ciclo de gerenciamento global, que exerce a função de ativar e desativar os outros dois ciclos. Em cada ciclo são processadas as funções de monitoramento, análise, planejamento e execução. Dessa forma, cada ciclo de gerenciamento autonômico é formado por componentes que desempenham o papel de sensor, monitor, analisador, executor e atuador.

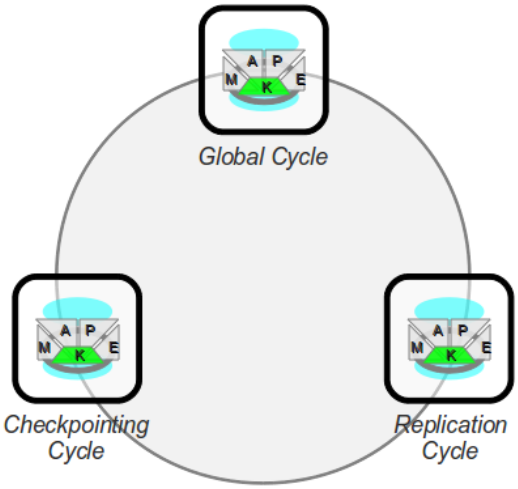

Figura 5.3: Ciclos de gerenciamento autonômico utilizados na abordagem.

5.2.1 Monitoramento

Conforme vimos na descrição da abordagem, a estratégia autonômica de tolerância a falhas necessita obter diversas informações sobre o contexto do ambiente de execução da grade. Para isso, foram desenvolvidos diversos sensores e monitores, utilizando como base as classes fornecidas pelo AGST. Os sensores foram desenvolvidos utilizando a classe `Sensor`, que permite a utilização de tecnologias baseadas em Reflexão Computacional ou no Paradigma da Orientação a Aspectos para obter dados sobre o elemento monitorado, sem que seja necessário realizar modificações em sua estrutura original. Os monitores foram implementados a partir

da classe `AbstractMonitor`, a qual agrega um objeto do tipo *Sensor* com o qual irá se comunicar para realizar suas operações de monitoramento sobre o elemento monitorado. Os monitores foram desenvolvidos para obter os dados coletados dos sensores de diferentes formas, pois os monitores podem realizar essa operação periodicamente, podem ser requisitados por outros componentes (como por exemplo, os analisadores) para realizar o monitoramento de forma instantânea, ou ainda podem aguardar que determinados eventos ocorram no elemento monitorado.

A classe `AbstractMonitor`, também permite que, após receber os dados de monitoramento do sensor, possa ser realizado sobre esses dados coletados um pré-processamento, a análise dos dados através das regras e um filtro para notificação de eventos. O pré-processamento consiste em lapidar os dados obtidos dos sensores de modo que sejam facilmente utilizados no processo de análise. Para isso, basta implementar o método abstrato `preprocess(MonitoringEvent ev)` que recebe como argumento um evento de monitoramento que contém os dados coletados. A classe `DefaultMonitor` implementa um monitor que não realiza o pré-processamento. O monitor também pode realizar a análise dos dados através das regras, essas regras devem ser criadas estendendo-se a classe abstrata `AbstractRule` e implementando o método `executeRule(MonitoringEvent ev)`, o método deve retornar um valor lógico indicando se houve alguma mudança significativa no contexto do ambiente monitorado.

Às vezes, uma informação pode ser importante para um analisador e não ser do interesse dos outros. Dessa forma, para que o monitor envie os dados somente para os analisadores que realmente tenham interesse neles, devem ser passados para o monitor filtros que indiquem os interesses de cada analisador. Para receber eventos de monitoramento os analisadores devem ser registrados junto ao `MAPEKRegistry`, que funciona como um serviço que cria as associações entre os analisadores e os monitores. Ao ser registrado o analisador, deve ser fornecido o nome do monitor que deseja se associar e uma filtro, o qual deve estender da classe `AbstractFilter`. O filtro deve indicar em quais situações o analisador deseja receber notificações do monitor. Para isso, deve ser implementado o método `executeFiltering(MonitoringEvent ev)`. O método deve retornar um valor lógico para indicar se a situação identificada corresponde a qual o analisador deseja ser notificado.

Para obter as informações necessárias ao funcionamento da estratégia autonômica, desenvolvemos três monitores, sendo que cada um deles está associado a um sensor específico. O diagrama de classes que representa essa estrutura pode ser visto na Figura 5.4.

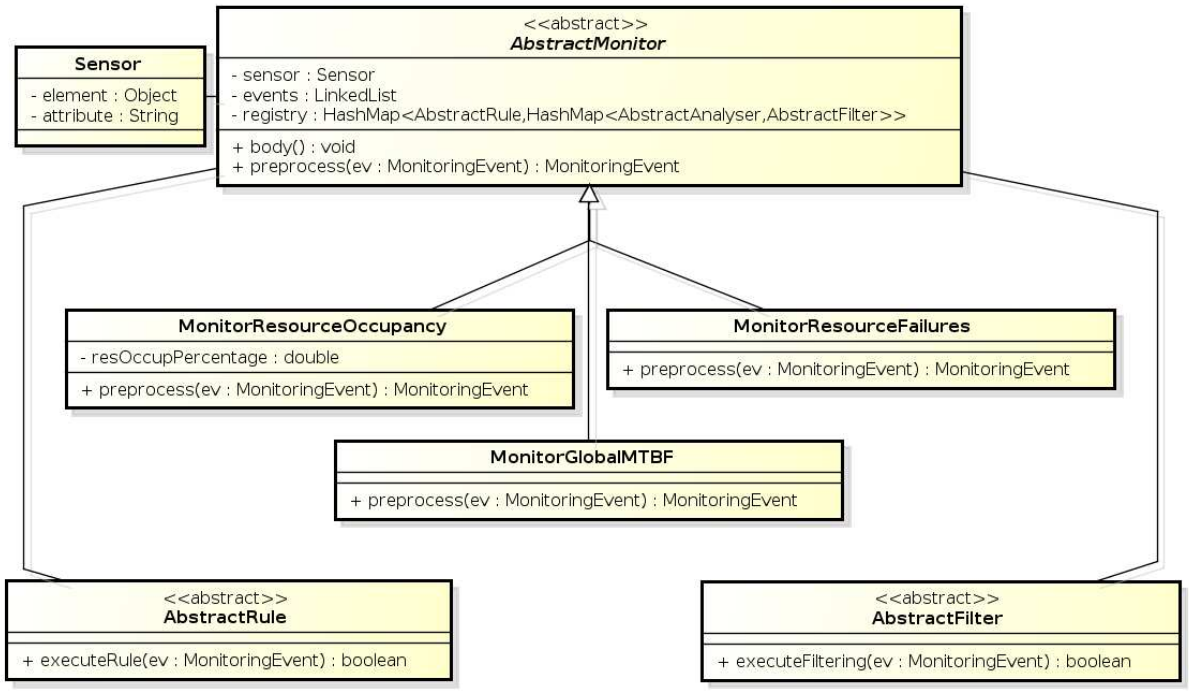

Figura 5.4: Diagrama de classes que representa os monitores da estratégia autonômica.

Como vimos no capítulo anterior, várias heurísticas como as da Seção 4.4, que descreve o comportamento autonômico que realiza reconfigurações estruturais efetuando trocas entre as técnicas de tolerância a falhas, e da Seção 4.3, sobre ajuste paramétrico no número de réplicas, que elas precisam de informações sobre o percentual de ocupação dos recursos da grade. Para obter essas informações foi desenvolvido um monitor chamado `resourceOccupancyMonitor` que funciona coletando periodicamente os dados sobre o estado de ocupação de todos os recursos da grade, em um intervalo de tempo que pode ser configurado pelo usuário. O monitor agrega um objeto do tipo `ResourceDynamicInfoSensor`, o qual foi implementado para obter a lista dos recursos ativos, contendo seus estados de ocupação ou livre. O `ResourceDynamicInfoSensor` utiliza Reflexão Computacional para isso, evitando que sejam feitas modificações no RDS, que é o componente monitorado da grade. O sensor repassa esses dados ao monitor, que realiza o pré-processamento, executa a análise de dados através de regras e utiliza filtros para descobrir qual analisador está interessado na informação.

No pré-processamento, o monitor calcula o percentual de ocupação da grade, que é feito de modo simples resultando da relação entre a quantidade de recursos ocupados e a quantidade de recursos ativos. Após o pré-processamento, o monitor faz a análise da informação utilizando regras para verificar se houve uma mudança significativa em relação ao contexto utilizado na última tomada de decisão. Isso é feito para que nem toda mudança no valor do percentual de ocupação seja notificada aos analisadores. Por exemplo, se o percentual de ocupação atinge 30%, o analisador do ciclo de replicação é notificado para diminuir o numero de replicas. Se no próximo monitoramento, a medição indicar 31%, certamente essa não é uma mudança significativa. Mas se alguns monitoramentos depois, o percentual alcançar 60%, essa é uma mudança significativa em relação à última tomada de decisão.

Esse monitor atua nos três ciclos da estratégia autonômica: na replicação, no *checkpointing* e no ciclo global. Cada analisador se registra junto a esse monitor fornecendo um filtro específico que irá determinar quais informações são importantes para cada um deles. Por exemplo, o analisador global só tem interesse em mudanças de contexto que levem a tomada de decisão no ciclo global.

Além do monitoramento periódico, o `resourceOccupancyMonitor` permite que analisadores possam requisitar monitoramentos instantâneos, como é o caso do analisador que realiza a tomadas de decisão sobre o cancelamento de réplicas. Pois, como vimos na Seção 4.5, para evitar o efeito ping-pong, o mecanismo continua cancelando réplicas enquanto exista réplicas e o nível de ocupação for superior ao limiar que determina a mudança para replicação. No monitoramento instantâneo, o monitor coleta dados e realiza o pré-processamento. Não há necessidade do uso de regras ou filtros, pois o analisador que requisitou será o único que receberá a informação.

Como vimos na Seção 4.2, para implementar o comportamento autonômico que realiza ajuste no intervalo entre os *checkpoints* de cada tarefa, são necessárias informações sobre o tempo médio entre falhas de cada recurso. Para obter essas informações foi implementado um monitor chamado `resourceFailureMonitor`, cuja função obter e pré-processar informações sobre o histórico da ocorrência de falhas de cada recurso. Essas informações são obtidas através do monitoramento periódico, cujo intervalo pode ser configurado pelo usuário. O sensor `resourceFailuresSensor` através de Reflexão Computacional

obtém do RDS uma lista que contém as ocorrências de falhas dos recursos. O `resourceFailuresMonitor` preprocessa essas informações calculando as previsões de falhas de cada recurso. Esse cálculo é feito somando-se o MTBF ao tempo de ocorrência da última falhas de cada recurso. Em seguida, essas informações são enviadas ao analisador do ciclo de *checkpointing*, o qual veremos na Subseção 5.2.2.

De acordo com o que foi apresentado na Seção 4.3, nosso modelo utiliza uma abordagem de adaptação paramétrica para ajustar o número de réplicas de acordo com o percentual de recursos ocupados e a taxa de falhas da grade. Entretanto, essa adaptação só é viável quando a taxa de falhas da grade como um todo for baixa. A fim de obter essas informações, foi implementado um monitor `globalMTBFMonitor`, cuja função é realizar o monitoramento instantâneo sempre que o analisador da replicação requisita. Esse monitor recebe dados do mesmo sensor (`resourceFailuresSensor`) usado pelo `resourceFailureMonitor`, porém o pré-processamento desses dados é realizado de forma diferente, buscando-se obter o MTBF global. Esse valor é obtido através da relação entre o tempo de execução decorrido a e a quantidade de falhas ocorridas nesse intervalo. Como o monitoramento é realizado sobre demanda, o `globalMTBFMonitor` não utiliza regras ou filtros.

Uma outra regra de adaptação vista na Seção 4.3 se baseia em tentar salvar o progresso da última réplica em execução, quando as demais já falharam. Para isso a estratégia autonômica precisa ser informada imediatamente quando a penúltima réplica de uma tarefa falhar, pois nesse momento, será feito o *checkpoint* da tarefa, sendo submetida mais uma vez, para executar a partir desse ponto, caso ela também venha a falhar. O componente responsável por monitorar e informar o analisador desse evento é o monitor chamado `lastExecutingReplicaMonitor`, que recebe os dados do sensor `lastExecutingReplicaSensor`, o qual intercepta (utilizando AOP) as notificações de ocorrências de falhas de cada réplica recebida pelo ARM. Dessa forma, sempre que o ARM recebe esse tipo de notificação, o sensor verifica, através da tabela de controle de execução das réplicas armazenada pelo ARM (a qual é obtida por *Reflexão Computacional*), se a réplica que falhou era a penúltima das réplicas dessa tarefa. Em caso positivo, a notificação será transmitida ao monitor que por sua vez retransmitirá ao analisador, caso contrário, o evento será ignorado.

5.2.2 Análise e Planejamento

Sempre que mudanças significativas ocorrerem no ambiente da grade, torna-se necessário analisar a informação do contexto, a fim de planejar as ações a serem executas para que o sistema volte ao estado de equilíbrio ou melhore a sua performance. A tomada de decisão na estratégia autonômica proposta é implementada por meio de regras que são executadas pelos analisadores. Para implementar um analisador, deve-se estender a classe `AbstractAnalyser`. Essa classe é provida pelo AGST para facilitar a implementação de analisadores, uma vez que suas subclasses não precisam se preocupar com detalhes da comunicação com os monitores e com o executor, pois esses detalhes já são tratados pela superclasse `AbstractAnalyser`. Cada subclasse de `AbstractAnalyser` deve apenas implementar o método `analyzeAndPlan()`, o qual recebe como parâmetro um evento de monitoramento (um objeto da classe `MonitoringEvent`) sempre que o monitor envia as notificações de mudança no contexto. Toda a lógica da análise e planejamento deve ser implementada nesse método.

Nosso modelo implementou três analisadores, conforme pode ser observado no diagrama de classes da Figura 5.5. O `AnalyserReplication` recebe eventos de monitoramento do `resourceOccupancyMonitor` sobre a percentual de ocupação dos recursos da grade. Essa informação é utilizada para que sejam tomadas decisões sobre adaptações paramétricas na técnica de replicação. De acordo com percentual informado o analisador deve decidir qual o número de réplicas que deve ser utilizado, criar a ação de reconfiguração apropriada e enviá-la para o executor. Para isso, basta instanciar uma ação do tipo `ActionChangeReplicasNum` passando como parâmetro no construtor da classe o número de réplicas que deve configurado e chamar o método `createActionPlan()` passando a ação instanciada como argumento.

O `AnalyserCheckpointing` recebe do `resourceFailuresMonitor` eventos de monitoramento das taxas de falhas de cada recurso da grade. Através dessas informações ele pode decidir se deve realizar a reconfiguração sobre o intervalo de *checkpointing* das tarefas em execução na grade. Para isso, o `AnalyserCheckpointing` precisar instanciar ações do tipo `ActionChangeChekpointingInterval` passando como parâmetro no construtor da classe o valor do novo intervalo. Além disso, este analisador recebe também

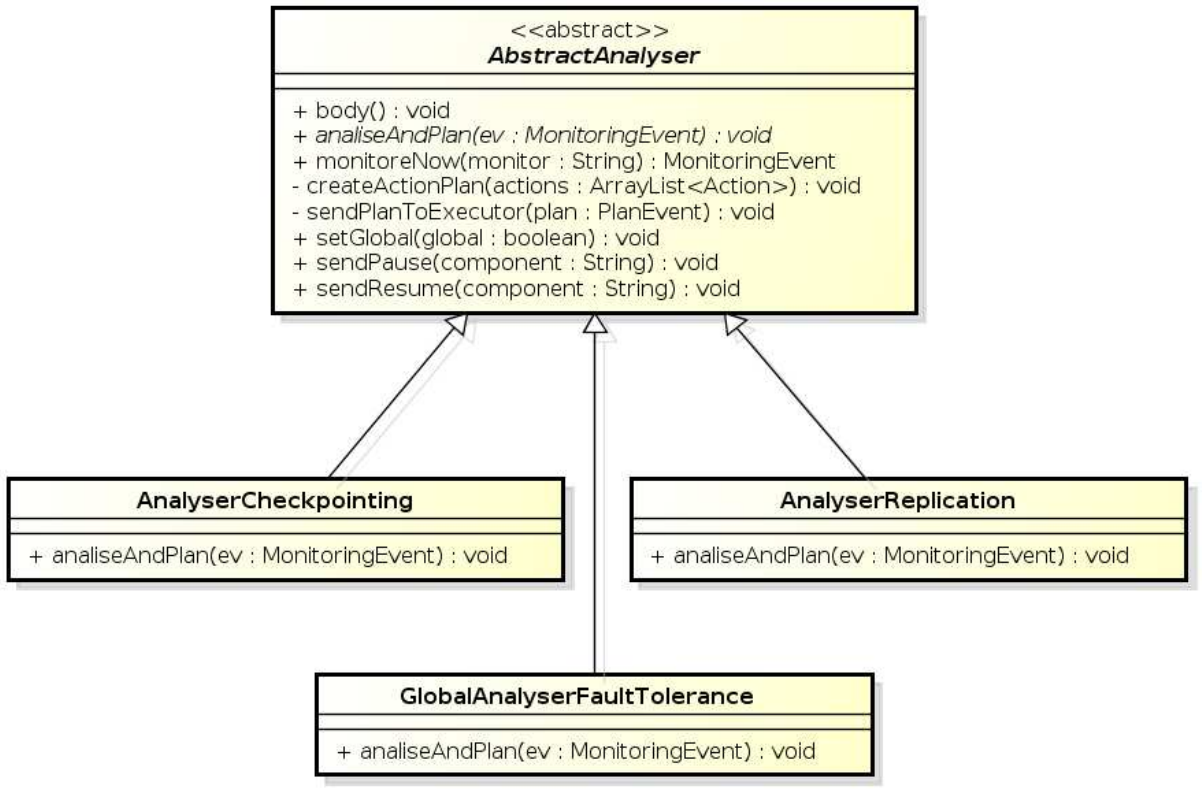

Figura 5.5: Diagrama de classes que representa os analisadores da estratégia autonômica.

eventos do `resourceOccupancyMonitor` sobre o percentual de ocupação dos recursos da grade. Essa informação é utilizada para que sejam tomadas decisões acerca do cancelamento de réplicas. Nesse caso, para cada réplica que deve ser cancelada, deve ser criado uma ação do tipo `ActionKillReplica`, a qual deve ser instanciada passando a referência da réplica. Como uma das réplicas deve continuar executando outra ação deve ser criada para alterar o regime de tolerância a falhas que essa tarefa obedece. Assim, a ação `ActionChangeTaskFTToCheckpointing` é instanciada passando a referência da réplica como argumento.

O `GlobalAnalyserFaultTolerance` recebe eventos de monitoramento do `resourceOccupancyMonitor`, sobre a percentual de ocupação dos recursos da grade. Bem como informações sobre a taxa de falhas global dos recursos da grade. Essa taxa leva em consideração falhas que ocorrem com qualquer recurso da grade. Essas informações são utilizadas nas tomadas de decisão sobre adaptações no nível estrutural que devem ser realizadas pela estratégia autonômica. De acordo com essas decisões o `GlobalAnalyserFaultTolerance` uma técnica de tolerância a falhas deve ser substituída por outra. Para isso ele deve criar ações do tipo `ActionChangeToCheckpointing` se a substituição for feita no sentido da replicação

para o *checkpointing*, ou então, se a substituição for no sentido inverso, ele deve criar a ação `ActionChangeToReplication` e nesse caso deve passar o número de réplicas como argumento no construtor da classe. Dessa forma, os analisadores que controlam o ciclo de gerenciamento autonômico das técnicas de replicação e de *checkpointing* devem ser iniciados ou parados de acordo com a reconfiguração que será realizada. Por isso, o `GlobalAnalyserFaultTolerance` também é responsável por controlar o ciclo global do sistema e representa o analisador global de nosso modelo. Ele controla os demais analisadores enviando eventos de pausa e continuação invocando os métodos sendPause() e sendResume(), respectivamente, da superclasse. Esses métodos recebem como parâmetro o nome dos componentes que devem entrar no estado de pausa ou retornar à execução. Os componentes manipulados por um analisador global pode ser um monitor ou um outro analisador.

Outra facilidade provida pela superclasse `AbstractAnalyser` é a criação e envio do plano de ações para o executor. Para criar o plano de ações, os analisadores devem invocar o método `createActionPlan()`, implementado na superclasse, que recebe como parâmetro um `ArrayList` de ações que irão compor o plano de adaptação. Cada ação implementada obrigatoriamente herda da classe abstrata `Action`, e cada plano de ação gerado é uma instância da classe `ActionPlan`. O plano que contém as ações de adaptação é enviado para o executor através da chamada do método `sendPlanToExecutor()`, o qual recebe um objeto do tipo `ActionPlan` como parâmetro. Esse método é implementado pela superclasse `AbstractAnalyser`.

5.2.3 Execution

Após elaborar o plano de adaptação, é necessário executar as ações contidas nesse plano. A execução da adaptação é implementada pelas classes `Executor` e `Effector`. No AGST apenas um objeto do tipo `Executor` pode ser instanciado, pois ele segue o padrão *Singleton*. O executor pode usar múltiplos atuadores para executar um plano de adaptação. Na implementação da abordagem autonômica os atuadores foram usados para alterar parâmetros como o número de réplicas da técnica de replicação e o intervalo que determina a periodicidade do *checkpointing* das tarefas,

bem como realizar adaptações estruturais, como a substituição da técnica de tolerância a falhas.

O método `executeActionPlan()` da classe `Executor` processa o plano de ações. Esse método tem o modificador de acesso *synchronized* para garantir que seja acionado apenas um plano de ações por vez, prevenindo que dois ou mais planos executem concorrentemente e assim seja preservada consistência do recurso gerenciado. O método `body()` da classe `Executor` processa um laço que dura até o fim da simulação. Cada iteração do laço verifica se um plano de ação foi submetido pelo `Analyser`. Quando um plano de ação é submetido, o `Executor` retira todas as ações contidas no plano e uma após a outra, seguindo a mesma ordem que elas ocupam no `ArrayList` de ações. Cada ação de adaptação é executada quando o `Executor` chama o método `execute()` da ação. Então tudo que deve ser feito é implementar as ações. Para isso, deve-se estender a classe `Action` e implementar o método `execute()`. Nesse método devem ser utilizados os atuadores que são responsáveis por alterar os parâmetros ou estrutura do sistema, de acordo com o tipo de adaptação realizada.

Um componente chamado `ExchangeComponent` foi provido para fornecer suporte no uso de reflexão computacional, durante o processo de adaptação pelos atuadores. O `ExchangeComponent` é uma classe concreta que implementa o padrão de projeto *Singleton*, isto é, foi projetado para que haja uma única instância dessa classe durante a simulação. O método `replace()` da classe `ExchangeComponent` é responsável por alterar o valor de um atributo ou realizar a substituição de um componente por outro. Esse método também tem modificador de acesso *synchronized* para garantir que apenas um processo de substituição seja executado por vez, prevenindo execução concorrente e preservando o estado de consistência. O método `replace()` recebe os seguintes parâmetros: o componente ou o atributo a ser substituído (`Object`), o componente ou o atributo de reposição (`Object`), e uma *flag* para indicar se deve ocorrer transferência de estado entre os componentes. Se o último parâmetro receber `true`, o método irá transferir o estado das variáveis internas de um componente para as variáveis internas do novo componente. Assim como os sensores, os atuadores também usam reflexão computacional para adquirir conhecimento a respeito da estrutura do recurso gerenciado, bem como ter acesso e alterar as variáveis que determinam seu estado.

5.3 Resultados e Discussões

Nesta seção descreveremos as simulações que executamos usando o AGST tendo como objetivo avaliar o desempenho da estratégia autonômica descrita no Capítulo 4 deste trabalho, tomando como base comparativa o desempenho das principais estratégias de tolerância a falhas para execução de aplicações em grades computacionais (Reinício, Replicação, *Checkpointing*). Estas simulações levaram em consideração diferentes condições do ambiente de execução. Nos diversos cenários simulados, variou-se a quantidade de recursos na grade e a taxa média da ocorrência de falhas de nós na grade.

As métricas utilizadas para avaliação da eficiência das estratégias nas simulações foram o percentual de sucesso alcançado pelas aplicações submetidas para execução e o tempo médio de conclusão das mesmas. O conjunto de resultados, que serão apresentados nos cenários a seguir, foram obtidos através de várias simulações. A combinação das estratégias de tolerância a falhas com os diversos cenários resultaram em um total de 24 simulações diferentes. Cada simulação foi repetida 20 vezes, resultando em 480 experimentos. Para cada experimento, medimos o tempo de conclusão por aplicação e calculamos o tempo médio de conclusão das aplicações. Em seguida, calculamos a média do tempo médio de conclusão das aplicações para cada um dos 20 conjuntos de aplicações que, para o restante do artigo, chamaremos apenas de tempo médio de conclusão por aplicação. Com base nos tempos médios de conclusão das aplicações, calculamos ainda as seguintes medidas estatísticas: valor mínimo, valor máximo, desvio padrão e o intervalo de confiança.

A estratégia de reinício avaliada emprega a técnica de tolerância a falhas de reinício no nível de tarefa, descrita na Seção 3.3. Nas simulações, essa estratégia foi configurada para reiniciar qualquer tarefa sempre que ela falhar. Cada evento de falha é notificado ao escalonador da grade (`GridScheduler`), o qual realiza um novo mapeamento da tarefa, de acordo com a heurística empregada pelo escalonador, e a envia para o recurso executar novamente toda a tarefa, sem levar em consideração a execução anterior à falha. Nessa configuração, a tarefa será reiniciada quantas vezes for necessário até que a tarefa conclua sua execução com sucesso.

A estratégia de replicação avaliada emprega a técnica de tolerância a falhas de replicação no nível de tarefa, também descrita na Seção 3.3. Nas simulações, a

replicação foi configurada para executar de forma estática com 3 réplicas. Cada vez que uma requisição de usuário chega à grade, o escalonador gera para cada tarefa da aplicação três mapeamentos, ou seja, uma mesma tarefa de uma aplicação é mapeada para executar com três instâncias, cada uma em um recurso diferente.

A estratégia de *checkpointing* avaliada emprega a técnica de tolerância a falhas de *checkpointing* no nível de tarefa (Seção 3.3). Nas simulações, ela foi configurado para realizar o *checkpoint* periódico das tarefas em um intervalo fixo de 1800 segundos (30 minutos). Cada vez que uma requisição de usuário chega à grade, o escalonador gera uma *thread* para cada tarefa da aplicação, a qual é responsável por gerenciar o *checkpointing* da sua tarefa.

Nas simulações, a estratégia autonômica, descrita no Capítulo 4, foi configurada para monitorar o percentual de ocupação dos recursos da grade a cada 10 segundos, monitorar o histórico de falhas dos recursos a cada 1.800 segundos e para ser notificado de cada ocorrência de falha de réplicas. Nas configurações de adaptações paramétricas no *checkpointing*, conforme descrevemos na Seção 4.2, foi utilizado como intervalo mínimo 5% do tempo estimado para execução da tarefa e no intervalo máximo 50% do tempo estimado para execução da tarefa. Nas configurações de adaptações paramétricas da replicação foram definidos os intervalos: $(0\% \leq percResOccup < 5\%)$ do percentual de ocupação de recursos para 5 réplicas, $(5\% \leq percResOccup < 10\%)$ para 4 réplicas e $(10\% \leq percResOccup < 30\%)$ para 3 réplicas, conforme visto na Seção 4.3. As trocas das técnicas de tolerância a falhas, conforme foi explanado na Seção 4.4, utilizou os intervalos $(0\% \leq percResOccup < 30\%)$ para replicação, $(30\% \leq percResOccup \leq 100\%)$ para *checkpointing* e $(80\% \leq percResOccup \leq 100\%)$ para efetuar o cancelamento de réplicas, segundo foi visto na Seção 4.5.

A seguir descreveremos as simulações realizadas em seis cenários diferentes, sendo que os quatro primeiros utilizaram o mecanismo de geração sintética de falhas do AGST e os dois últimos utilizaram um banco de dados de *traces* de falhas.

5.3.1 Simulações com Falhas Sintéticas

Nas primeiras simulações, foi configurado no AGST para a geração de falhas sintéticas o uso de uma distribuição exponencial, por caracterizar bem o tempo

entre falhas sucessivas, possuir uma grande variabilidade de valores e apresentar independência entre um valor e outro [7]. A geração de falhas sintéticas tornou possível criar ambientes de grades com ocorrência de falhas controladas, de modo que fosse possível avaliar o comportamento das estratégias em cenários com muitas e poucas falhas. Nessas simulações, consideramos ambientes com poucas falhas aqueles que utilizam um tempo médio entre falhas (MTBF) igual a 3.600 segundos e ambientes com muitas falhas o MTBF igual a 500 segundos. O tempo de duração da falha (*downtime*) foi determinado por uma distribuição exponencial com média variável, cujos valores mínimos e máximos foram de respectivamente de 300 a 600 segundos, representando falhas de recursos com recuperação rápida. Pois em ambientes de grades oportunistas, são mais frequentes as falhas de curta duração, tais como o reinício de máquinas por seu usuário local, ou as causadas por oscilação ou falta de corrente elétrica, do que as de as de longa duração, como as causadas por falhas no hardware das máquinas, por exemplo.

Além da variação no MTBF, foram criados cenários com muitos recursos, configurado com um total de 1.400 recursos, e com poucos recursos, configurado com 100 recursos. Da combinação dessas características, foram criados quatro cenários que descreveremos e discutiremos os seus resultados a seguir.

Cenário com Muitos Recursos e Poucas Falhas

O objetivo desta primeira simulação foi avaliar o desempenho das estratégias de tolerância a falhas em ambientes com muitos recursos e baixa taxa de falhas. Para isso, foram utilizados 1.400 recursos de grades gerados sinteticamente e para as falhas, foi configurado o tempo médio entre falhas (MTBF) igual a 3.600 segundos. O restante do ambiente da grade foi configurado como segue.

Os recursos foram interconectados de forma homogênea por uma rede de 100 Mbps. O poder de processamento médio definido foi equivalente a um Pentium IV com 1,6 GHz (1.858 MIPS, tendo-se por base o *benchmark* TSCP 5), considerado um valor representativo para computadores pessoais. A fim de levar em consideração a heterogeneidade do ambiente, as capacidades de processamento dos nós da grade foram geradas de acordo com uma distribuição uniforme U(938; 2.779) MIPS, onde o

poder de processamento da máquina mais rápida é aproximadamente 3 vezes maior que o poder de processamento da máquina mais lenta.

Uma outra característica de grades oportunistas levada em consideração foi a utilização de máquinas não dedicadas que, portanto, possuem carga de trabalho local. Para a geração dessas cargas, nos baseamos no trabalho de Conde [9]. Nesse trabalho foram coletados registros do uso de recursos (CPU e memória) de diversas máquinas pertencentes a laboratórios do Departamento de Ciência da Computação da Universidade de São Paulo, armazenados em arquivos de trace. Através de uma ferramenta no AGST, que realizou a leitura e análise desses arquivos foi gerado um outro arquivo contendo vetores com 24 posições, as quais representam as 24 horas do dia. Para usar essa carga de trabalho local dos nós nas simulações realizadas, o arquivo contendo esses vetores foi passado como parâmetro para o AGST, que se encarrega de simular a carga de trabalho local a partir dos mesmos.

Com relação à carga de trabalho da grade, foram geradas sinteticamente 100 aplicações do tipo *bag-of-tasks*, com 3 tarefas cada, resultando em um total de 300 tarefas. Essas aplicações foram geradas sinteticamente com uma variação de tamanho (em termos de milhões de instruções) através de uma distribuição uniforme $U(53,510 \times 10^3; 321,062 \times 10^3)$ MI. Considerando o poder de processamento médio utilizado para os nós da grade (1.858 MIPS), cada aplicação levaria, aproximadamente, de 8 a 48 horas para ser executada completamente. Neste cenário, a grade recebe em média uma requisição para execução de aplicação dos usuários a cada 40 segundos, o que resulta numa taxa de 0,025 aplicações por minuto. Para a modelagem e simulação dos eventos de chegada das aplicações em todos os experimentos foi utilizada a distribuição de *Poisson*, pois segundo Chwif e Medina [7] essa distribuição é uma das mais adequadas para essa finalidade.

No ambiente simulado, as tarefas são mapeadas para os recursos utilizando a heurística usada no InteGrade. O algoritmo de escalonamento do InteGrade foi escolhido porque este trabalho está inserido no contexto do desenvolvimento desse *middleware* de grade oportunista. Essa heurística é um algoritmo do tipo *on-line* que utiliza um filtro para selecionar os recursos a partir das restrições e preferências providas pelos usuários durante o processo de submissão das aplicações. As restrições definem requisitos mínimos para a seleção de máquinas como, por exemplo, a utilização de máquinas que tenham pelo menos 4 GB de memória RAM. Já as

preferências definem a ordem na escolha dos recursos como, por exemplo, ordenar as máquinas pela maior quantidade de memória disponível. As tarefas que compõem uma aplicação são então escalonadas para os nós de acordo com a lista de recursos ordenados. Caso não sejam especificados requisitos ou preferências, o algoritmo mapeia as tarefas que compõem a aplicação para recursos da grade escolhidos de forma aleatória.

A Figura 5.6 ilustra os resultados obtidos com este primeiro cenário. A estratégia de replicação apresentou o menor tempo médio de conclusão das aplicações. Atribuímos este resultado ao melhor aproveitamento que essa estratégia faz dos recursos disponíveis.

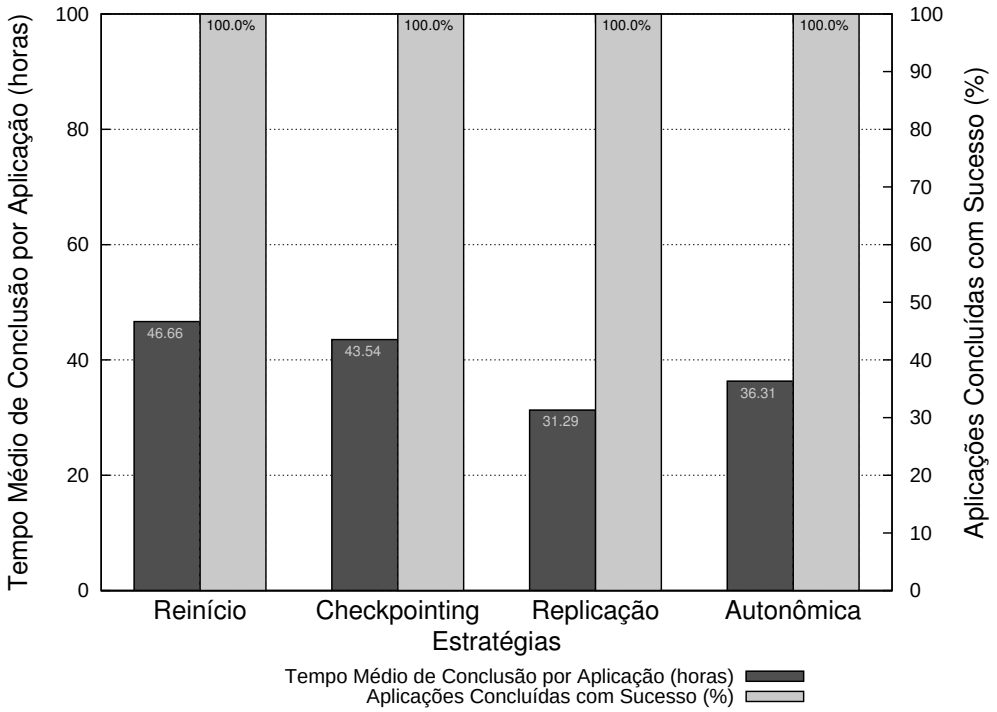

Figura 5.6: Tempo médio de conclusão por aplicação e percentual de conclusões com sucesso das estratégias de tolerância a falhas em uma grade de 1400 nós em um ambiente de poucas falhas.

Podemos perceber que o resultado da estratégia autonômica se aproxima do resultado da replicação, dado que ela faz uso da estratégia de replicação quando há muitos recursos disponíveis. A estratégia autonômica começa utilizando replicação, e a medida que as réplicas vão sendo geradas o percentual de ocupação de recursos aumenta e a estratégia muda fazendo com que as novas submissões de tarefas utilizem *checkpointing*. A sobrecarga gerado pelo *checkpointing* dessas tarefas eleva

o tempo média de conclusão desta estratégia como um todo. Isso aconteceu nas simulações realizadas neste cenário porque a estratégia foi configurada para realizar essa adaptação quando o percentual de ocupação alcançasse o valor de 30%. O reinício teve o maior tempo de conclusão porque se alguma tarefa falhar ela deve ser reexecutada desde o início, e como as tarefas são de longa duração esse tempo eleva o resultado da média. A baixa taxa de falhas do cenário justifica o 100 % de sucesso na conclusão das aplicações de todas as estratégias.

A Tabela 5.1 expressa alguns dados estatísticos sobre os resultados deste cenário. Podemos observar o desvio padrão (DP) dos tempos médios de conclusão (TMC) por aplicação foi baixo, e que por isso as simulações apresentaram um intervalo de confiança (de 95%) fechado. Foi considerado como tempo de conclusão o tempo que uma aplicação leva para concluir, desde a requisição do usuário até a conclusão da sua última tarefa. O mesmo aconteceu com o tempo médio de execução das aplicações (TME). O tempo de execução é contado desde o início da execução da primeira tarefa da aplicação até o término da execução da última tarefa. Outra constatação importante é que o tempo médio de conclusão das aplicações são superiores ao tempo de conclusão das tarefas, isso se deve ao fato de utilizarmos aplicações *bag-of-tasks*.

Resultados por Aplicações											Resultados por Tarefas	
Estratégias	Tempo Médio de Conclusão (TMC)					Tempo Médio de Execução (TME)				Sucesso	TMC	Sucesso
	Média	D.P.	Min.	Máx.	IC (95%)	Média	D.P.	Min.	Máx.	(%)		(%)
Reinício	46.66	0.62	45.43	47.79	46.39 46.93	46.66	0.54	45.63	47.72	100.0	30.22	100.0
Checkpointing	43.54	0.48	42.63	44.43	43.33 43.75	43.54	0.46	42.66	44.36	100.0	29.7	100.0
Replicação	31.29	0.25	30.79	31.78	31.18 31.40	31.29	0.25	30.87	31.76	100.0	22.21	100.0
Autonômica	36.31	0.39	35.58	37.08	36.14 36.48	36.31	0.49	35.50	37.27	100.0	24.95	100.0

Tabela 5.1: Resultados das simulações de estratégias de tolerância a falhas em uma grade com 1400 nós em um ambiente de poucas falhas.

Muitos Recursos e Muitas Falhas

Neste cenário, foram realizadas simulações com o objetivo de avaliar o desempenho das estratégias de tolerância a falhas em ambientes com muitos recursos e alta taxa de falhas. Para esse ambiente, foram utilizados os 1.400 recursos gerados no cenário anterior e para a geração de falhas sintéticas foi utilizado o tempo médio entre falhas (MTBF) igual a 500 segundos. Outros aspectos de configuração deste cenário foram feitas para que se mantivessem iguais ao do cenário anterior, como o ambiente

da grade (topologia da rede e o poder de processamento dos recursos da grade), a carga de trabalho local (usando os *traces* do InteGrade), a carga de trabalho da grade (com 100 aplicações do tipo *bag-of-task* com três tarefas cada) e a heurística de escalonamento do InteGrade.

A Figura 5.7 ilustra os resultados obtidos com essas simulações. Neste cenário, podemos observar que a estratégia de replicação continua apresentando o menor tempo médio de conclusão das aplicações. Contudo, houve uma pequena redução no percentual de conclusões com sucesso. Atribuímos essa redução às falhas que ocorreram com maior intensidade, fazendo com que todo o conjunto de réplicas de algumas tarefas acabassem falhando. Em consequência disso, também foi impossibilitada a conclusão das aplicações dessas tarefas.

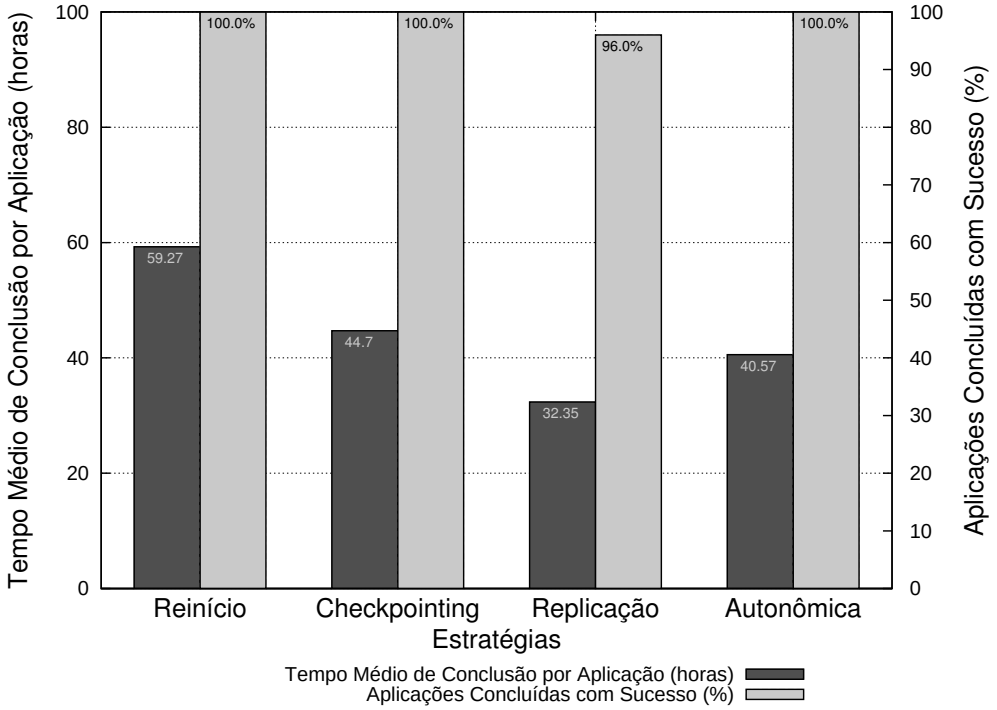

Figura 5.7: Tempo médio de conclusão por aplicação e percentual de conclusões com sucesso das estratégias de tolerância a falhas em uma grade de 1400 nós em um ambiente de muitas falhas.

Houve também um pequeno aumento no tempo médio de conclusão das aplicações, devido a re-submissão das tarefas que falharam pelas estratégias de reinício, *checkpointing* e a autonômica. O pequeno aumento no tempo de conclusão da replicação, pode ser explicado pela forma de escalonamento da heurística do InteGrade, que escalona tarefas para recursos com diferentes capacidades de

Resultados por Aplicações											Resultados por Tarefas	
Estratégias	Tempo Médio de Conclusão (TMC)					Tempo Médio de Execução (TME)				Sucesso	TMC	Sucesso
	Média	D.P.	Min.	Máx.	IC (95%)	Média	D.P.	Min.	Máx.	(%)		(%)
Reinício	59.27	0.94	57.51	61.04	58.86 59.68	59.27	0.96	57.47	60.93	100.0	37.45	100.0
Checkpointing	44.7	0.50	43.76	45.59	44.48 44.92	44.7	0.58	43.69	45.78	100.0	30.5	100.0
Replicação	32.35	0.39	31.65	33.07	32.18 32.52	32.35	0.35	31.73	33.01	96.0	24.1	98.67
Autonômica	40.57	0.46	39.85	41.46	40.37 40.77	40.57	0.36	39.94	41.29	100.0	27.39	100.0

Tabela 5.2: Resultados das simulações de estratégias de tolerância a falhas em uma grade com 1.400 nós em um ambiente de muitas falhas.

processamento de forma aleatória, bem como pela geração sintéticas de falhas, que seleciona aleatoriamente o recurso que irá falhar. De um modo geral, as estratégias de tolerância a falhas mantiveram um mesmo padrão de comportamento, em relação a simulação anterior com a mesma quantidade de recursos e com a taxa de falhas menor.

Os dados estatísticos deste cenário são apresentados na Tabela 5.2.

Poucos Recursos e Poucas Falhas

Neste cenário, o objetivo foi avaliar o desempenho dessas estratégias em ambientes com muitos recursos e baixa taxa de falhas. Para esse ambiente, foram gerados 100 recursos para a grade e para a geração de falhas sintéticas o MTBF utilizado foi igual a 3.600 segundos. Outros aspectos de configuração deste cenário foram feitas para que se mantivessem iguais aos dos cenários anteriores, como o ambiente da grade (topologia da rede e o poder de processamento dos recursos da grade), a carga de trabalho local (usando os *traces* do InteGrade), a carga de trabalho da grade (com 100 aplicações do tipo *bag-of-task* com três tarefas cada) e a heurística de escalonamento do InteGrade.

Os resultados obtidos com este cenário estão representados no gráfico da Figura 5.8. Conforme podemos observar, apesar da replicação apresentar o menor tempo médio de conclusão, o percentual de aplicações que concluíram com sucesso é muito pequeno. Nesse cenário, as primeiras aplicações executas com replicação conseguiram concluir rapidamente, enquanto que as demais não conseguiram concluir. Isso justifica porque o tempo desta técnica foi tão baixo.

A estratégia autonômica apresentou uma pequena redução no percentual de sucessos obtidos, isso se dá porque apesar do uso de re-submissão de tarefas para

execução, algumas tarefas podem não ser reexecutadas quando a estratégia faz uso de replicação. Em contrapartida, a estratégia autonômica apresentou o menor tempo de conclusão que o *checkpointing* e o reinício, que é justificado pela quantidade de re-submissões que essas duas últimas técnicas fazem para alcançar o sucesso de todas as tarefas. Observamos também que a tempo médio do *checkpointing* foi maior que o do reinício. Isso se explica porque o *checkpointing* causa uma sobrecarga no tempo de execução das tarefas, enquanto que a técnica de reinício não produz esse tipo de atraso. Essa sobrecarga só seria compensado se houvesse uma maior quantidade de falhas no ambiente, já que o reinício provocaria a re-execução completa das tarefas que falharam.

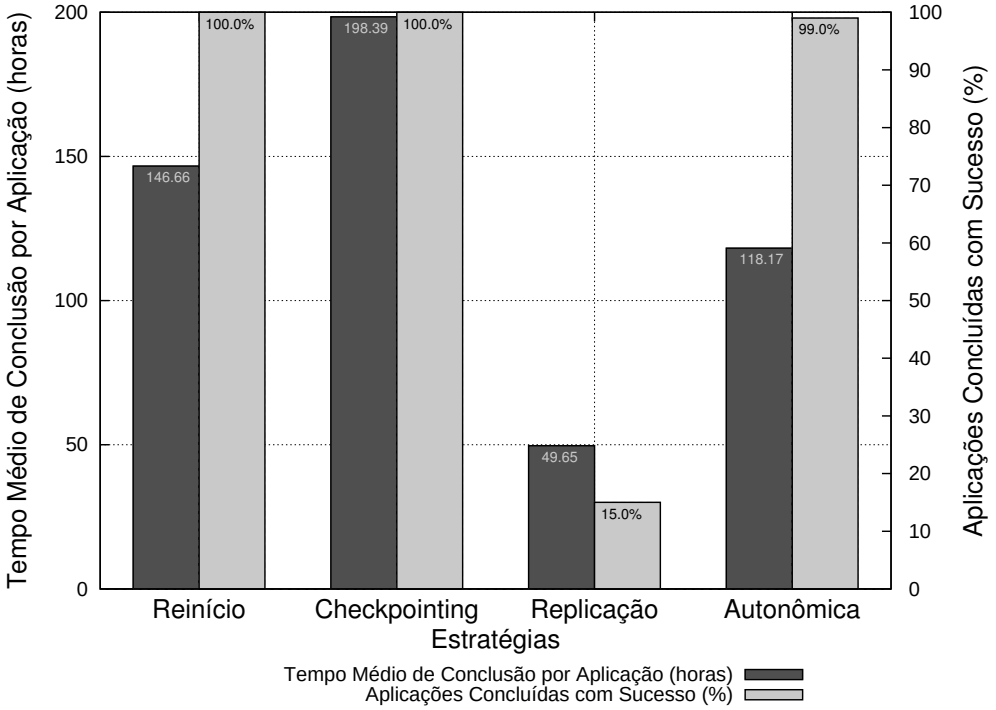

Figura 5.8: Tempo médio de conclusão por aplicação e percentual de conclusões com sucesso das estratégias de tolerância a falhas em uma grade de 100 nós em um ambiente de poucas falhas.

Os dados estatísticos deste cenário são apresentados na Tabela 5.3.

Resultados por Aplicações											Resultados por Tarefas	
Estratégias	Tempo Médio de Conclusão (TMC)					Tempo Médio de Execução (TME)				Sucesso	TMC	Sucesso
	Média	D.P.	Min.	Máx.	IC (95%)	Média	D.P.	Min.	Máx.	(%)		(%)
Reinício	146.66	3.17	140.87	151.88	145.27 148.05	146.66	3.25	141.85	152.79	100.0	101.31	100.0
Checkpointing	198.39	3.51	191.84	205.10	196.85 199.93	198.39	3.58	192.73	204.78	100.0	130.65	100.0
Replicação	49.65	1.05	47.71	51.68	49.19 50.11	49.65	1.08	48.16	51.63	15.0	38.21	15.0
Autonômica	118.17	1.71	114.89	121.26	117.42 118.92	118.17	1.80	114.80	121.47	99.0	89.05	99.67

Tabela 5.3: Resultados das simulações de estratégias de tolerância a falhas em uma grade com 100 nós em um ambiente de poucas falhas.

Poucos Recursos e Muitas Falhas

Neste cenário, avaliamos o desempenho das estratégias de tolerância a falhas em ambientes com muitos recursos e alta taxa de falhas. Para esse cenário, o ambiente da grade foi configurado com 100 recursos e o MTBF, para a geração de falhas sintéticas, igual a 500 segundos. Outros aspectos de configuração deste cenário foram feitas para que se mantivessem iguais ao dos cenários anteriores, como o ambiente da grade (topologia da rede e o poder de processamento dos recursos da grade), a carga de trabalho local (usando os *traces* do InteGrade), a carga de trabalho da grade (com 100 aplicações do tipo *bag-of-task* com três tarefas cada) e a heurística de escalonamento do InteGrade.

A Figura 5.7 ilustra os resultados obtidos com estas simulações. Analisando os dados, podemos observar que a estratégia de replicação apresentou um percentual de sucesso extremamente baixo, devido ao grande número de falhas e poucos recursos para serem utilizados. Devido a quantidade de falhas, o numero de ressubmissões no reinício também foi muito maior que no cenário anterior, o que justifica a elevação acentuada no tempo médio de conclusão das aplicações quando utilizada essa técnica. O *checkpointing* apresentou o melhor desempenho em relação ao percentual de sucessos e ao tempo médio de conclusão das aplicações. Isso se deve ao fato do *checkpointing* salvar o estado de execução da tarefa, o que evita ter que reexecutá-la do início em caso de falha. Como a quantidade de falhas é elevada neste cenário, o custo do *checkpoint* é compensado por não se ter que reiniciar tarefas que falharam do início.

A estratégia autonômica se aproximou deste resultado, apresentando um tempo médio de conclusão bastante próximo. Isto acontece em função da adaptação

realizada pela estratégia, trocando a abordagem da replicação pelo uso da técnica de *checkpointing* devido ao contexto do ambiente de execução.

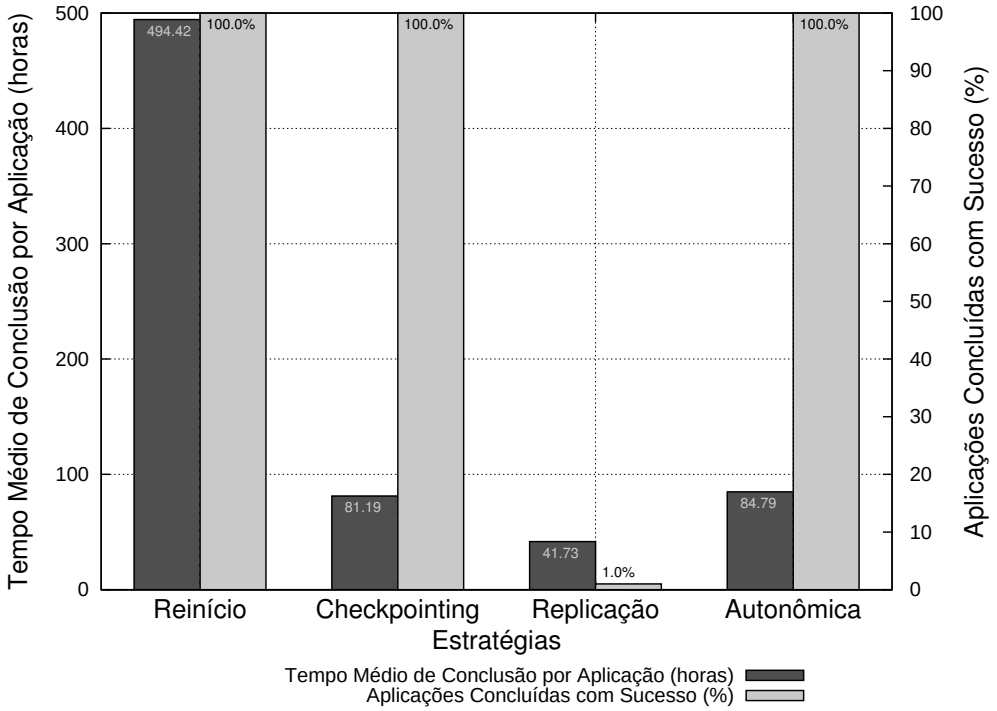

Figura 5.9: Tempo médio de conclusão por aplicação e percentual de conclusões com sucesso das estratégias de tolerância a falhas em uma grade de 100 nós em um ambiente de muitas falhas.

Os dados estatísticos deste cenário são apresentados na Tabela 5.4.

Resultados por Aplicações											Resultados por Tarefas	
Estratégias	Tempo Médio de Conclusão (TMC)					Tempo Médio de Execução (TME)				Sucesso	TMC	Sucesso
	Média	D.P.	Min.	Máx.	IC (95%)	Média	D.P.	Min.	Máx.	(%)		(%)
Reinício	494.42	29.30	448.00	541.41	481.58 507.26	494.42	29.21	439.86	552.30	100.0	241.44	100.0
Checkpointing	81.19	0.98	79.30	83.00	80.76 81.62	81.19	1.07	79.40	83.24	100.0	61.25	100.0
Replicação	41.73	0.02	41.68	41.77	41.72 41.74	41.73	0.02	41.69	41.77	1.0	21.83	9.0
Autonômica	84.79	0.89	83.02	86.44	84.40 85.18	84.79	0.79	83.25	86.16	100.0	63.38	100.0

Tabela 5.4: Resultados das simulações de estratégias de tolerância a falhas em uma grade com 100 nós em um ambiente de muitas falhas.

5.3.2 Simulações com Uso de *Traces* de Falhas

Nas simulações realizadas para os próximos cenários utilizamos uma base de dados com *traces* de falhas. Esses experimentos foram feitos com o objetivo de avaliar as estratégias de tolerância a falhas para aplicações em grades em um cenário

que fosse capaz de reproduzir o comportamento de falhas ocorrido em um ambiente real de grade. Os *traces* utilizados correspondem a eventos de disponibilidade e indisponibilidade de recursos não dedicados observados em uma grade real, registrados durante a sua execução por um período de aproximadamente 6 meses. Nesse *trace* constam registros de 700 nós, situados na Universidade de Notre Dame[4]. O registro desses dados foram feitos seguindo o padrão FTA[5], conforme são descritos em [42].

Simulações com Muitos Recursos

Neste cenário, avaliamos o desempenho das estratégias de tolerância a falhas em ambientes com muitos recursos (1.400 nós). O comportamento de falhas foi obtido a partir de *traces*. Apenas os dados de falhas, como o instante em que um recurso se torna indisponível ou disponível, foram carregados desses *traces* para as simulações. Isso foi feito de modo que cada recurso da grade assumisse o comportamento de falhas de um recurso do *trace*. Outros aspectos de configuração deste cenário foram feitas para que se mantivessem iguais ao dos cenários de falhas sintéticas, como o a topologia da rede e o poder de processamento dos recursos da grade, a carga de trabalho local (usando os *traces* do InteGrade), a carga de trabalho da grade (com 100 aplicações do tipo *bag-of-task* com três tarefas cada) e a heurística de escalonamento do InteGrade.

A Figura 5.10 ilustra os resultados obtidos neste cenário. Analisando os dados, podemos observar que a estratégia de replicação apresentou o menor tempo médio de conclusão das aplicações. Atribuímos este resultado ao melhor aproveitamento que essa estratégia faz dos recursos disponíveis. Se compararmos este cenário com os dois primeiros cenários com falhas sintéticas, podemos observar que as estratégias, de uma modo geral, apresentaram um mesmo padrão de comportamento. A estratégia autonômica se aproximou da técnica que obteve melhor resultado, que neste cenário foi a replicação.

[4] A Universidade de Notre Dame está situada em South Bend, Indiana, USA. O *Departament of Science Computing and Engineering* dessa universidade mantém um pool de execução de grade do Condor `http://www.cse.nd.edu/~ccl/operations/condor/status.shtml`.

[5] `http://fta.inria.fr/apache2-default/pmwiki/index.php`

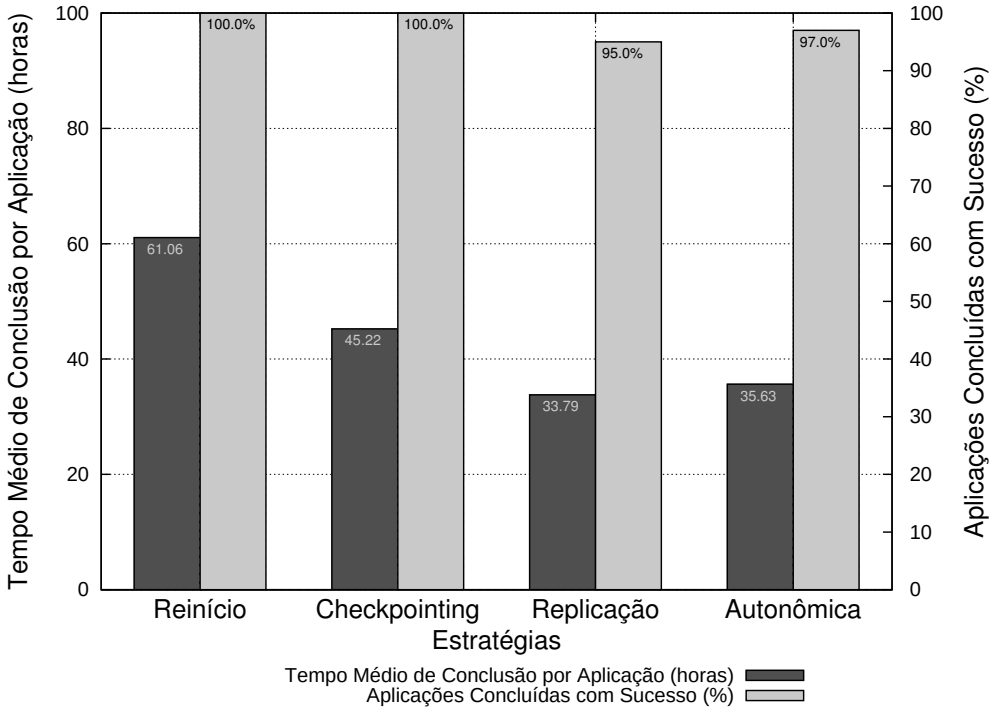

Figura 5.10: Tempo médio de conclusão por aplicação e percentual de conclusões com sucesso das estratégias de tolerância a falhas em uma grade de 1.400 nós em um ambiente com falhas de *traces*.

Os dados estatísticos deste cenário são apresentados na Tabela 5.5.

Resultados por Aplicações											Resultados por Tarefas	
Estratégias	Tempo Médio de Conclusão (TMC)					Tempo Médio de Execução (TME)				Sucesso	TMC	Sucesso
	Média	D.P.	Min.	Máx.	IC (95%)	Média	D.P.	Min.	Máx.	(%)		(%)
Reinício	61.06	2.26	57.13	65.03	60.07 62.05	61.06	2.28	57.27	65.42	100.0	36.87	100.0
Checkpointing	45.22	0.50	44.27	46.02	45.00 45.44	45.22	0.49	44.25	46.18	100.0	31.06	100.0
Replicação	33.79	0.32	33.18	34.41	33.65 33.93	33.79	0.38	33.17	34.55	95.0	23.7	98.33
Autonômica	35.63	0.41	34.88	36.36	35.45 35.81	35.63	0.44	34.78	36.48	97.0	25.73	99.0

Tabela 5.5: Resultados das simulações de estratégias de tolerância a falhas em uma grade com 1.400 nós em um ambiente com falhas de *traces*.

Simulações com Poucos Recursos

Neste cenário, avaliamos o desempenho das estratégias de tolerância a falhas em ambientes com poucos recursos (100 nós). O comportamento de falhas foi obtido a partir de *traces*. Da mesma forma que no cenário anterior, apenas os dados de falhas foram utilizados nas simulações. Outros aspectos de configuração deste cenário foram feitas para que se mantivessem iguais ao dos cenários de falhas sintéticas, como

o a topologia da rede e o poder de processamento dos recursos da grade, a carga de trabalho local (usando os *traces* do InteGrade), a carga de trabalho da grade (com 100 aplicações do tipo *bag-of-task* com três tarefas cada) e a heurística de escalonamento do InteGrade.

A Figura 5.11 ilustra os resultados obtidos com estas simulações. Analisando os dados, podemos concluir que as estratégias de tolerância a falhas apresentaram um comportamento semelhante ao apresentado no quarto cenário da simulação sintética, no qual o ambiente foi configurado para exibir alta taxa de falhas com poucos recursos. No entanto, a estratégia autonômica, neste cenário utilizando *trace* de um ambiente real, conseguiu alcançar um tempo médio de conclusão menor que a estratégia de *checkpointing* e também apresentou uma pequena redução no percentual de conclusões com sucesso. Isso é justificado pelo fato do comportamento da estratégia autonômica buscar utilizar as vantagens das duas técnicas de replicação e *checkpointing*, realizando para isso adaptações de acordo com o monitoramento que ela faz do ambiente de execução.

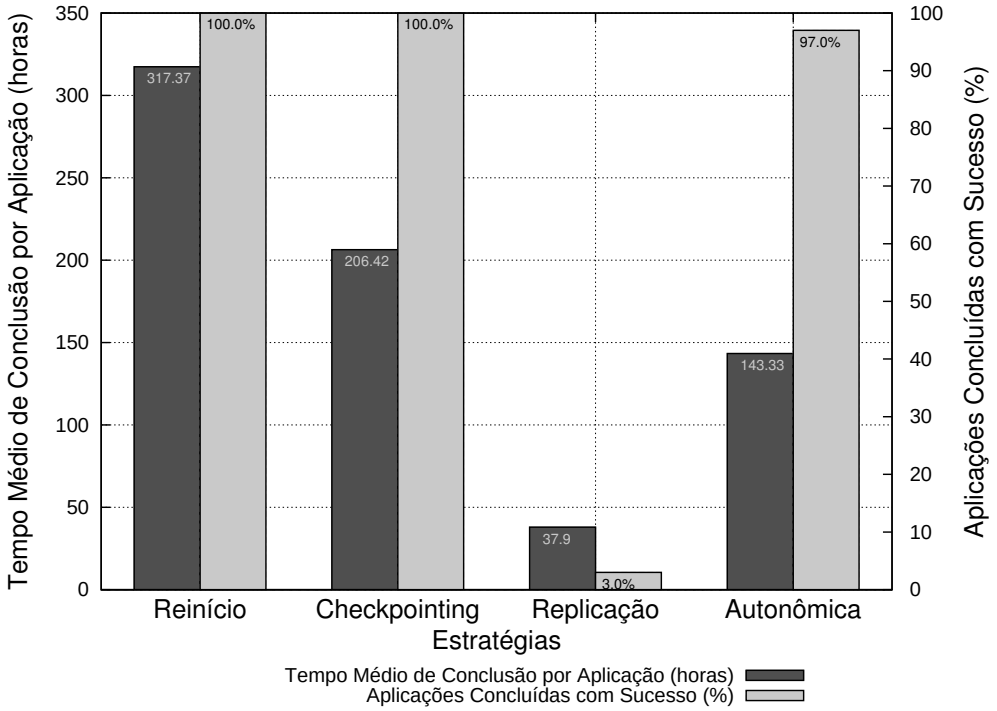

Figura 5.11: Tempo médio de conclusão por aplicação e percentual de conclusões com sucesso das estratégias de tolerância a falhas em uma grade de 100 nós utilizando-se *traces* de falha de recursos.

Os dados estatísticos deste cenário são apresentados na Tabela 5.6.

Resultados por Aplicações											Resultados por Tarefas	
Estratégias	Tempo Médio de Conclusão (TMC)					Tempo Médio de Execução (TME)				Sucesso	TMC	Sucesso
	Média	D.P.	Min.	Máx.	IC (95%)	Média	D.P.	Min.	Máx.	(%)		(%)
Reinício	317.37	5.02	307.62	326.46	315.17 319.57	317.37	5.04	308.59	327.30	100.0	189.07	100.0
Checkpointing	206.42	4.34	199.11	214.88	204.52 208.32	206.42	4.35	198.93	214.91	100.0	114.71	100.0
Replicação	37.9	0.23	37.48	38.33	37.80 38.00	37.9	0.23	37.45	38.31	3.0	28.42	9.67
Autonômica	143.33	2.60	138.38	148.40	142.19 144.47	143.33	2.64	138.42	148.59	97.0	93.27	99.0

Tabela 5.6: Resultados das simulações de estratégias de tolerância a falhas em uma grade com 100 nós utilizando-se *traces* de falha de recursos.

5.4 Conclusão

Vimos neste capítulo que o AGST fornece um conjunto de componentes que facilitam o desenvolvimento de modelos de simulação que visam avaliar mecanismos autonômicos para sistemas de grades computacionais. O AGST permite criar componentes seguindo o modelo MAPE-K de gerenciamento autonômico. Na nossa abordagem utilizamos diversos componentes fornecidos pelo AGST para implementar um modelo composto por três ciclos de gerenciamento autonômico: um ciclo global responsável por realizar adaptações no nível estrutural da estratégia autonômica de tolerância a falhas e que controla os demais ciclos; um ciclo responsável por controlar adaptações paramétricas na técnica de replicação; e outro ciclo que também controla adaptações a nível paramétrico, mas na técnica de *checkpointing*.

A utilização dos recursos disponibilizados pelo AGST simplificou consideravelmente o esforço necessário para implementar o modelo de simulação da abordagem autonômica de tolerância a falhas proposto neste trabalho. Para a fase de monitoramento do ciclo MAPE-K de nossa abordagem implementamos três monitores: um responsável por colher dados de ocupação dos recursos da grade, outro por colher dados sobre a taxa de falhas de cada recurso e um último para calcular a taxa de falhas global dos recursos da grade. O maior esforço foi concentrado na elaboração da lógica da tomada de decisão e na criação de ações de reconfiguração. Nesse sentido, o AGST permitiu que nossa implementação estivesse focada principalmente na elaboração das heurísticas descritas no Capítulo 4, que constituem a base de conhecimento do modelo autonômico de nossa abordagem de tolerância a falhas.

Para avaliar a abordagem autonômica descrita no Capítulo 4 realizamos diversas simulações usando o AGST, levando-se em consideração diferentes condições do ambiente de execução. Nos diversos cenários simulados, variou-se a quantidade de recursos na grade e a taxa média da ocorrência de falhas de nós, comparando os resultados obtidos com a abordagem autonômica com os resultados das estratégias de tolerância a falhas usualmente empregadas em grades de computadores.

Segundo os resultados apresentados pelas simulações realizadas, observamos que a utilização das técnicas de reinício, replicação e *checkpointing* apresentam desempenhos diferentes de acordo com o estado do ambiente de execução. Através das simulações realizadas, pudemos observar que a estratégia autonômica produziu bons resultados, aproximando-se da estratégia que obteve o melhor desempenho em cada um dos 6 cenários avaliados. Portanto, podemos concluir que em ambientes de grade que apresentem variações na disponibilidade de recursos e falhas de nós, a abordagem autonômica proposta contribui significativamente para reduzir o tempo de conclusão das aplicações, mantendo elevado percentual de sucesso em suas execuções.

6 Trabalhos Relacionados

Diversos trabalhos têm surgido com o propósito de prover tolerância a falhas às aplicações submetidas para execução em grades computacionais. Podemos perceber que ao longo dos anos esses trabalhos apresentaram uma evolução em relação ao desempenho das estratégias de tolerância a falhas propostas. Os primeiros trabalhos envolviam alguma técnica de tolerância a falhas que era implementada de forma estática nos *middleware* de grades. Em seguida, surgiram alguns trabalhos propondo mecanismos flexíveis que permitiam que os usuários ajustassem parâmetros ou escolhessem qual técnica deveria ser empregada durante a execução das aplicações que eles submetiam à grade. Mais recentemente, têm surgido alguns poucos trabalhos que propõem a utilização de técnicas adaptativas para prover reconfiguração dinâmica nas técnicas de tolerância falhas utilizadas pelo *middleware* da grade.

Este capítulo descreve relevantes projetos, cujo objetivo é o desenvolvimento de técnicas de tolerância a falhas para execução de aplicações em grades de computadores. Apresentaremos alguns trabalhos que propõem mecanismo flexíveis para prover tolerância a falhas na execução de aplicações em grades computacionais e alguns recentes trabalhos que propõem abordagens adaptativas. Após um resumo de cada projeto realizaremos uma análise comparativa com a estratégia proposta neste trabalho de dissertação.

6.1 Trabalhos que Empregam Técnicas de Tolerância a Falhas de Forma Flexível

Vários trabalhos de tolerância a falhas para execução de aplicações em grades apresentam mecanismos básicos que utilizam apenas uma das técnicas de forma estática: reinício, replicação ou *checkpointing*. Pruyne e Livny desenvolveram um mecanismo que faz *checkpointing* de aplicações paralelas no Condor [59] e Litzkow et al. deram continuidade a esse trabalho permitindo que processos UNIX pudessem ser migrados entre duas máquinas disponíveis na grade, ou seja, utilizam o *checkpoint*

da execução de um processo em uma máquina para ser restaurado em outra [45]. Paranhos et al. desenvolveram um mecanismo baseado em replicação para aplicações de alto-desempenho no OurGrid [56][8].

Uma das desvantagem das abordagens estáticas é que elas não permitem ajustes de acordo com variações que ocorram no ambiente da grade. Para tentar resolver esse problema, alguns trabalhos propõem flexibilidade para o usuário na escolha sobre qual estratégia de TF aplicar na execução da sua aplicação, como é o caso do trabalho proposto por Souza et al. em [63], que desenvolveu um mecanismo flexível que permite aos usuários escolherem entre as técnicas de reinício, replicação e *checkpointing*, bem como utilizá-las de modo combinado. Outro exemplo é o trabalho de Hwang e Kesselman que propõem um *framework* baseado em *workflows* para definir, além do fluxo de execução de tarefas, as estratégias de tolerância a falhas que cada tarefa deve utilizar [33]. Nesta seção apresentaremos, de forma sucinta, esses dois trabalhos destacando suas contribuições no que diz respeito a tolerância a falhas de aplicações em grades computacionais.

6.1.1 Um Mecanismo Flexível de Tolerância a Falhas para o *Middleware* de Grade do InteGrade

Esse trabalho foi apresentado por Souza et al. [63] e consiste no desenvolvimento de um mecanismo implementado no *middleware* de grade do InteGrade que permite a customização de vários parâmetros e a combinação de diferentes técnicas utilizadas para o tratamento de falhas.

O InteGrade é um *middleware* de grade [23][14] cujo desenvolvimento envolve o esforço de várias instituições brasileiras e que visa o compartilhamento do poder computacional de estações de trabalho ociosas para execução de aplicações computacionalmente intensivas. A unidade arquitetural básica de uma grade que utiliza o *middleware* InteGrade é o aglomerado (*cluster*). Os aglomerados podem ser organizados em uma hierarquia, permitindo abranger um grande número de máquinas. Cada aglomerado contém um nó chamado *Cluster Manager*, que executa os componentes do InteGrade responsáveis por gerenciar os recursos computacionais mesmo e pela comunicação entre aglomerado. Os outros nós são chamados de estações

de trabalho, que podem ser máquinas dedicadas ou compartilhadas. Os principais componentes do InteGrade são:

- ***Application Submission and Control Tool* (ASCT)**: uma interface que permite que os usuários submetam e controlem a execução de aplicações;
- ***Application Repository* (AR)**: armazena as aplicações que podem ser executadas pela grade;
- ***Local Resource Manager* (LRM)**: componentes que executam nos nós do aglomerado instanciando as aplicações escalonadas para esse nó e coletando informações sobre o estado dos recursos como memória, CPU, disco e uso da rede;
- ***Global Resource Manager* (GRM)**: gerencia os recursos do aglomerado recebendo os dados coletados pelos LRMs e escalonando as tarefas para os nós de acordo com suas disponibilidades;
- ***Execution Manager* (EM)**: mantém as informações sobre o estado de execução de cada requisição e outros dados como o nó de execução, parâmetros de entrada e saída e tempos de submissão e conclusão.

Com o propósito de prover flexibilidade no uso de técnicas de tolerância a falhas para os usuários da grade, eles acrescentaram ao InteGrade a técnica de replicação (*replication*) e reestruturaram o *middleware* do InteGrade para permitir que os usuários da grade pudessem customizar parâmetros relacionados ao mecanismo de tratamento de falhas durante o processo de submissão das aplicações. Essas customizações consistem na escolha das técnicas de tolerância a falhas que devem ser empregadas na execução de suas aplicações e na configuração dos parâmetros relacionados às técnicas escolhidas. O mecanismo desenvolvido permite o uso combinado de algumas técnicas de forma que as opções de escolhas possíveis para os usuários são: reinício (sem *checkpoint* ou replicação), *checkpointing*, replicação (sem *checkpointing*) e replicação com *checkpointing*.

Para que esse nível de flexibilidade se tornasse possível no InteGrade, foram feitas modificações e desenvolvidos alguns componentes[1]:

1. **Execution Manager (EM)**: modificado para manter as informações sobre as coordenadas de recuperação das aplicações em caso de falhas;
2. **Checkpointing library (ckpLib)**: provê a funcionalidade de periodicamente gerar os *checkpoints* contendo o estado das aplicações;
3. **Autonomous Data Repositories (ADRs)**: um armazém estável distribuído que utiliza os recursos compartilhados da grade;
4. **Cluster Data Repository Manager (CDRM)**: gerencia a disponibilidade dos ADRs em um aglomerado e a controla localização dos *checkpoints* de cada aplicação;
5. **Application Replication Manager (ARM)**: instância e gerencia as réplicas de uma aplicação em execução.

6.1.2 GridWorkflow: Um framework para tratamento de falhas flexível para grades

Soonwook Hwang e Carl Kesselman [33] propuseram um *framework* baseado em *workflows* que permite ao usuário definir, de forma flexível, especificações para controlar, além do fluxo de execução de tarefas, as estratégias para recuperação de falhas que deve ser utilizado na execução de cada tarefa. Essa flexibilidade consiste em fornecer variadas formas de tratamento de falhas que podem ser especificadas pelo usuário, as quais podem ser escolhidas de acordo com os possíveis tipos de falhas a que estão sujeitas suas aplicações. Por exemplo, o usuário pode especificar que se não houver espaço em disco suficiente para a tarefa, ela deve ser cancelada e reiniciada em outro recurso ou deve ser executada uma outra implementação com menor desempenho que realize mesma tarefa, mas que consuma menos o espaço em disco.

[1]O técnica de replicação e a escolha flexível do mecanismo de tolerância a falhas ainda não encontra integrada a versão oficial do InteGrade disponível em `http://www.integrade.org.br/en/software`

Esse trabalho incorpora estratégias para o tratamento de falhas no nível da tarefa (reinício, replicação e *checkpointing*) e no nível do *workflow* (tarefa alternativa, redundância e tratamento de outras exceções definidas pelo usuário). Por exemplo, uma técnica do nível de *workflow* pode envolver duas diferentes implementações para uma determinada tarefa: uma mais rápida, porém mais propensa a falhas e outra mais lenta, contudo mais confiável. Nesse caso pode-se utilizar tanto a técnica da *tarefa alternativa* quanto a de *redundância*. Na técnica da tarefa alternativa, o mecanismo irá tentar executar primeiramente a mais rápida e, se ela falhar, tentará executar a mais confiável. Na técnica da redundância, ambas serão executadas simultaneamente, sendo considerados apenas os resultados da que terminar primeiro. O *framework* do GridWorkflow permite as seguintes configurações:

- Combinar o uso de diferentes técnicas de recuperação de falhas no nível da tarefa. Por exemplo, os usuários podem especificar que cada réplica pode ser reiniciada apenas adicionando a informação do número de reinícios, na especificação do *workflow* da tarefa.

- Combinar o uso de técnicas de recuperação de falhas no nível da tarefa e no nível do *workflow*. No exemplo citado acima (da tarefa alternativa ou da redundância), a execução da implementação mais rápida poderia ser mais tolerante a falhas se fosse aplicada a sua execução alguma das técnicas do nível da tarefa como reinício, replicação ou *checkpointing*.

- Permitir a alteração da estratégia de tolerância a falhas facilmente através da modificação da estrutura do *workflow*. Por exemplo, o usuário pode definir uma estrutura de *workflow* para a execução de sua tarefa e caso precise alterar a estratégia de tolerância a falhas, basta apenas alterar a estrutura das especificações do *workflow*, não sendo necessário recompilar, *linkeditar* ou mesmo testar a execução da aplicação novamente. Para que essa flexibilidade seja possível, a grade deve possuir uma grande variedade de mecanismos que forneçam suporte à detecção de falhas e às estratégias de tolerância a falhas abordadas.

Um protótipo deste *framework*, denominado *Grid Workflow System* (GridWFS) e ilustrado na figura 6.1, foi implementado, cujos principais componentes são:

- **Uma Linguagem para Definição do Processamento de *Workflows* usando XML**, utilizada pelos usuários para a especificação do processamento do *workflow* na forma de um grafo acíclico dirigido (*Directed Acyclic Graph* - DAG);

- ***Workflow Engine***, um mecanismo que controla a execução do *workflow*. Este navega através das especificações do *workflow* submetendo as tarefas para nós específicos da grade e também monitora o estado de cada tarefa submetida;

- **Serviços de Execução do *Workflow***, um conjunto de serviços de diretórios que oferecem suporte ao *Workflow Engine* durante a execução do fluxo, armazenando informações sobre as aplicações, os dados e os recursos.

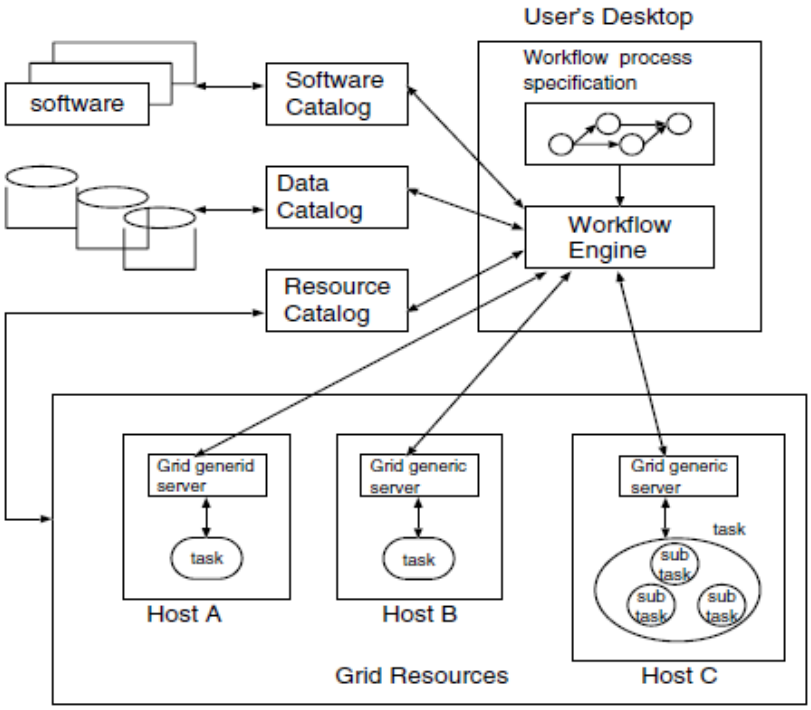

Figura 6.1: Arquitetura do Grid-WFS.

6.2 Trabalhos que Empregam Técnicas de Tolerância a Falhas de Forma Adaptativa

Embora a flexibilidade na escolha da técnica seja um avanço para tolerância a falhas na execução de aplicações em grades, existem situações em que o usuário não consegue prever as condições do ambiente de execução da grade no momento em que

ele submete suas tarefas. Também devido a grande escalabilidade e ao dinamismo usualmente observados em grades de computadores, a configuração de mecanismos flexíveis de tolerância a falhas por parte de administradores humanos é uma tarefa difícil de ser realizada e propensa a erros. Portanto, uma abordagem autonômica, na qual o próprio mecanismo decide o que fazer seria uma abordagem mais adequada a esse tipo de ambiente.

Dentro dessa linha de pesquisa, alguns trabalhos como [71], [25] e [5] apresentam soluções adaptativas para tomada de decisão. Nesta seção apresentaremos, de forma sucinta, esses trabalhos visando suas contribuições no que diz respeito a tolerância a falhas de aplicações em grades computacionais.

6.2.1 Uma abordagem adaptativa de tolerância a falhas no nível da tarefa para Grade

Uma proposta adaptativa apresentada por Wu et. al. em [71] decide qual técnica de tolerância a falhas deve ser utilizada pelo *middleware* da grade, tomando como base para a tomada de decisão o estado da tarefa. O mecanismo utiliza quatro técnicas básicas de tolerância a falhas: *retry*, *alternate resource*, *checkpointing* e replicação. As duas primeiras são variações da técnica de reinício, onde *retry* (que pode ser traduzido como *"nova tentativa"*) considera o reinício da tarefa no mesmo recurso em que ocorreu a falha, enquanto que *alternate resource* (que pode ser traduzido como *"recurso alternativo"*) considera o reinício da tarefa em um novo recurso, diferente do qual ela estava executando quando falhou. O modelo da abordagem desse trabalho é descrita na forma de um algoritmo que está ilustrado na figura 6.2.

O mecanismo realiza continuamente *checkpointing* durante toda execução da tarefa. Se a primeira falha ocorre, a tarefa é reiniciada a partir do último estado salvo no mesmo recurso. Nesse primeiro caso, considera-se que foi apenas uma falha transiente e que o reinício da tarefa é suficiente para solucionar o problema. Se uma segunda falha ocorrer, considera-se que é bastante provável que o recurso no qual a tarefa está executando não é estável e, portanto, a tarefa é reiniciada a partir do último *checkpoint* em um outro nó. Se a tarefa falhar novamente no novo recurso, uma nova consideração é feita, baseando-se em que o ambiente está sujeito a falhas e por isso devem ser

```
1  Todas as tarefas são inicialmente assinaladas com failure_rank = 3;
2  se a tarefa falhar então
3      se failure_rank = 3 então
4          Reinicia a tarefa no mesmo recurso a partir do último
           checkpoint;
5          failure_rank- -;
6      senão se failure_rank = 2 então
7          Busca um novo recurso, transfere a tarefa e o último
           checkpoint;
8          Reinicia a tarefa a partir do checkpoint transferido;
9          failure_rank- -;
10     senão se failure_rank = 1 então
11         Busca N recursos, transfere a tarefa e o último checkpoint
           simultaneamente para esses N recursos;
12         Reinicia as N replicas simultaneamente a partir dos
           checkpoints transferidos nos respectivos N recursos;
13     fim
14 fim
```

Figura 6.2: Algoritmo da abordagem adaptativa de Wu et. al.

instanciadas várias réplicas da tarefa, para serem executadas simultaneamente. Nesse caso o uso de replicação é combinado com o uso de *checkpointing*.

6.2.2 Um *Framework* para Execução Adaptativa e Tolerante a Falhas de *Workflows* em *Grid*

Este outro trabalho consiste na dissertação de mestrado de Felipe Guimarães [25], o qual apresenta um *framework* para execução adaptativa e tolerante a falhas de *workflows* que permite que usuários de grades computacionais possam definir dinamicamente a técnica de tolerância a falhas que deve ser usada durante a execução das tarefas de seus *workflows*. Para isso o usuário deve submeter, juntamente com a especificação do *workflow*, as regras que serão utilizadas na tomada de decisão, as quais

devem ser expressas na forma de árvores binárias. O *framework* também permite que o mecanismo substitua dinamicamente o mecanismo de tolerância a falhas selecionado por outro que satisfaça as condições de suas especificações. Além disso, o *framework* permite que o usuário desenvolva novas técnicas de tolerância a falhas e as adicione ao *framework*.

A arquitetura do *framework* desenvolvido nesse trabalho está ilustrada na figura 6.3 e apresenta os seguintes passos de execução: primeiro o usuário deve prover o *workflow*, os arquivos de entrada, um conjunto de regras para escolha da técnica de tolerância a falhas e um conjunto de arquivos de configuração (a), os quais são utilizados para instanciar coordenadores chamados *Fault-tolerant Execution Coordinators* (FTECs). Um coordenador é instanciadop ara cada técnica de tolerância a falhas que será utilizada durante a execução das tarefas do *workflow*. O *workflow* é recebido pelo *Workflow Submission and Control Tool* (WSCT), o qual irá transformá-lo em um conjunto de tarefas e relações de precedências (b). O *Workflow Manager* (WFM) verifica as tarefas com dependências satisfeitas, prontas para serem submetidas (c). Essas tarefas são então submetidas para o *Fault-Tolerance Manager* (FTM), responsável por controlar os aspectos relacionados a tolerância a falhas durante a execução das tarefas do fluxo.

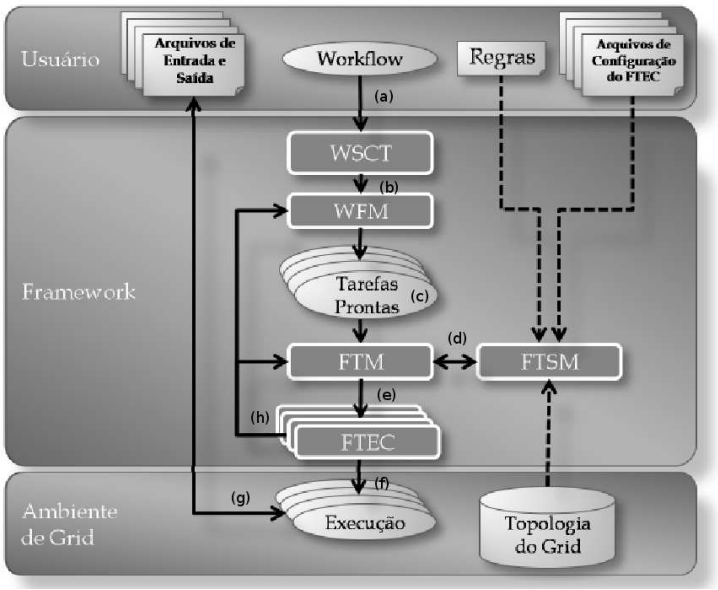

Figura 6.3: Arquitetura do *Framework* proposto por [25].

O FTM consulta o *Fault-Tolerance Selection Mechanism* (FTSM) sobre qual técnica é a mais apropriada para ser utilizada com aquele conjunto de tarefas naquele

dado momento. O FTSM utiliza as regras definidas pelo usuário para escolher e enviar o identificador do FTEC que deve ser usado (d). Após isso, o FTM fará uma invocação ao Servidor FTEC para cada tarefa que irá executar, o qual irá criar uma nova *thread* FTEC para coordenar a execução daquela tarefa específica (e). A *thread* FTEC criada irá interagir com o *middleware* da grade para executar a tarefa e sua tolerância a falhas (f). O FTEC obtém os arquivos de entrada, requisita e monitora a execução da tarefa na grade (g). Quando a tarefa completa a sua execução, o WFM verifica se há novas tarefas prontas para execução (h), e assim o processo continua até que não haja mais tarefas para executar.

A arquitetura com múltiplas *threads* dos FTECs está representada na figura 6.4. Cada FTEC é um servidor que implementa uma técnica de tolerância a falhas e cada *thread* é responsável por uma tarefa. Para implementar novas técnicas de tolerância a falhas o usuário deve criar uma nova classe que herde da classe padrão FTEC. Depois disso, o FTM estará pronto para invocar o método `startFtecThread` que irá criar novas threads com os novos FTECs. Para que a nova técnica seja selecionada o usuário deve definir regras que utilizem a nova técnica.

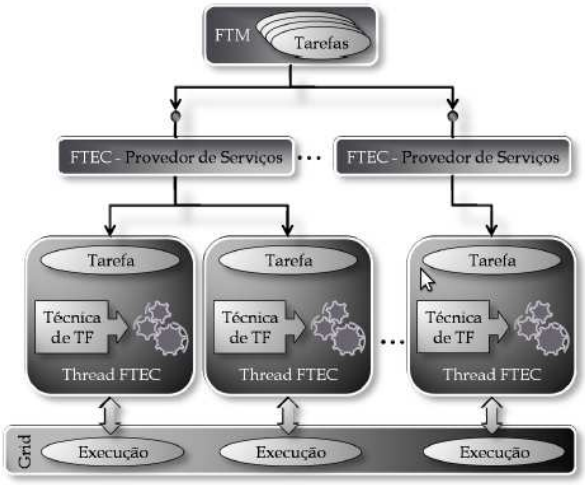

Figura 6.4: A arquitetura com múltiplas *threads* dos FTECs.

O mecanismo de seleção FTSM utiliza uma especificação baseada em árvore binária para realizar a escolha do FTEC mais apropriado. A figura 6.5 exemplifica uma árvore binária de decisão. Através dessa estrutura, o algoritmo de decisão irá percorrer a árvore no sentido da raiz para as folhas, de modo que cada nó intermediário representa uma condição que verifica informações dinâmicas do ambiente para escolha sobre qual será o próximo nó a ser seguido. Esse processo se repete até que o algoritmo

alcance os nós-folhas da árvore. Cada nó-folha determina um mecanismo FTEC específico a ser utilizado.

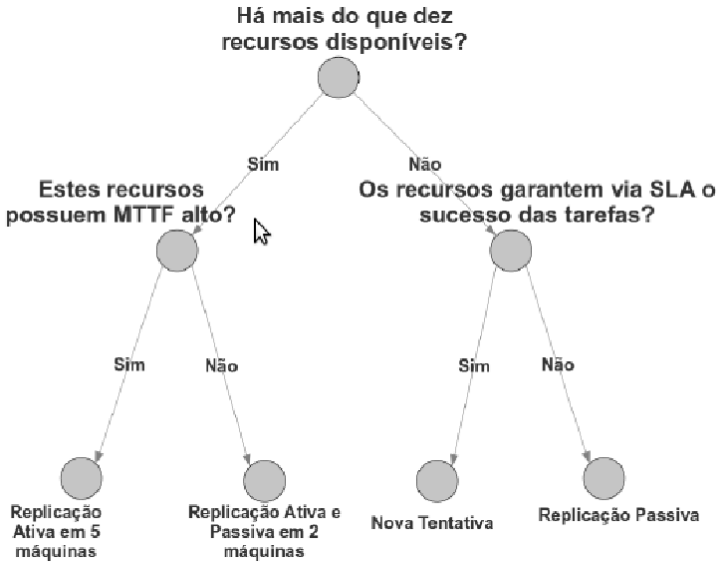

Figura 6.5: Exemplo simplificado de árvore de decisão binária.

6.2.3 *Checkpointing* e Replicação de Tarefas Adaptativos: em Direção às Grades Eficientes Tolerantes a Falhas

Maria Chtepen et al. apresentam em [5] algumas heurísticas adaptativas para os mecanismos de *checkpointing*, replicação e uma abordagem combinada deles, com o objetivo de melhorar a utilização de recursos e reduzir o tempo de execução de aplicações em grades computacionais. Nesse trabalho, foi levado em consideração um modelo de grade conforme ilustrado na figura 6.6.

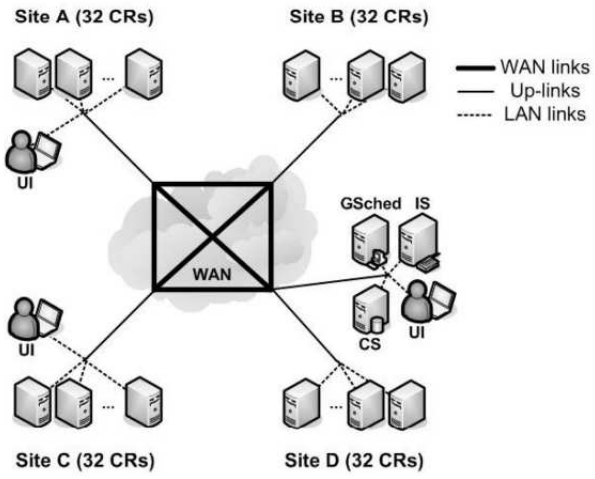

Figura 6.6: Modelo de grade utilizado em [5].

Esse modelo consiste em uma grade composta por sites (S) dispersos geograficamente, agregando 128 recursos computacionais (CR) e um conjunto de serviços gerais de grade que inclui: uma interface para submissão de tarefas de usuários (UI), um escalonador (GSched), um serviço de informações sobre os recursos (IS) e um servidor de *checkpoint* (CS). Os sites estão espalhados pela rede WAN, enquanto que os recursos de um mesmo site são interconectados por uma LAN. O modelo considera que todos os serviços de gerenciamento são protegidos contra falhas e apenas os CRs são instáveis.

Nesse trabalho, são apresentadas algumas heurísticas, dentre as quais destacamos três: (1) uma que busca ajustar os intervalos entre os *checkpoints* para cada tarefa de acordo com o MTBF do recurso em que está executando; (2) outra que busca limitar o uso de replicação de acordo com a carga do sistema; (3) e uma que combina o uso das duas heurísticas anteriores. Essas heurísticas serão descritas abaixo:

Mean Failure Dependent Checkpointing (`MeanFailureCP`)

A heurística `MeanFailureCP` modifica a frequência do *checkpointing* dinamicamente para lidar com intervalos que não adequados ao ambiente de falhas da grade. O algoritmo proposto nesse trabalho modifica o intervalo baseado na informação sobre o tempo estimado para o restante da execução da tarefa e a média do tempo entre as falhas do recurso em que a tarefa está executando. O funcionamento dessa heurística está exemplificado na figura 6.7.

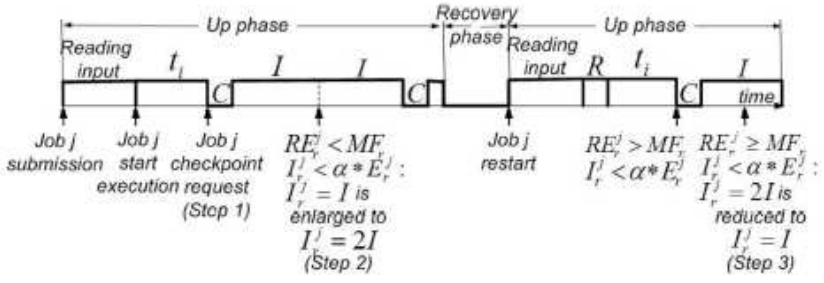

Figura 6.7: Exemplo de funcionamento do `MeanFailureCP`.

O *checkpointing* de cada tarefa j começa com um intervalo predefinido (Passo 1) e cada vez que o *checkpointing* é executado, o intervalo é adaptado de acordo com o tempo estimado para o término da tarefa j no recurso r (RE_r^j) e o tempo médio

entre falhas do recurso (MF_r) em que está executando. O algoritmo abaixo expressa as sentenças que formam as condições para a tomada de decisão:

```
1 se RE_r^j < MF_r and I_j < α × Ej então
2 |   // diminui a frequência do checkpointing
3 |   // através do crescimento do intervalo (Passo 2);
4 |   I_jnew ← I_jold + I_j;
5 senão
6 |   // aumenta a frequência do checkpointing
7 |   // através da redução do intervalo (Passo 3);
8 |   I_jnew ← I_jold − I_j;
9 fim
```

Figura 6.8: Algoritmo de comparação da abordagem `MeanFailureCP`.

Duas condições definem se o intervalo do *checkpointing* será aumentado, diminuindo a frequência com que será feito (Passo 2): a primeira inequação prevê que o recurso r seja suficientemente estável para que seja concluída a execução da tarefa j, enquanto que a segunda previne que o intervalo do *checkpointing* I_j cresça indefinidamente. $\alpha < 1$ limita o intervalo I_j para que seja menor que o tempo de execução total da tarefa E_j. Por outro lado, quando essas inequações não são satisfeitas, o intervalo do *checkpointing* é reduzido, aumentando a frequência com que será feito (Passo 3).

Failure-Dependent Replication (`FailureDependentRep`)

Nessa heurística, o algoritmo leva em consideração a carga do sistema e, de acordo com a mesma, atrasa ou reduz a replicação nas horas de pico. Um exemplo de uma operação com essa heurística está representado na figura 6.9, cujos parâmetros e valores iniciais são: número mínimo de réplicas ($Rep_{min} = 1$), número máximo de réplicas ($Rep_{max} = 2$) e o limite mínimo de CPUs livres para que se utilize replicação ($CL = 2$). No exemplo, o sistema possui dois recursos com duas CPUs cada.

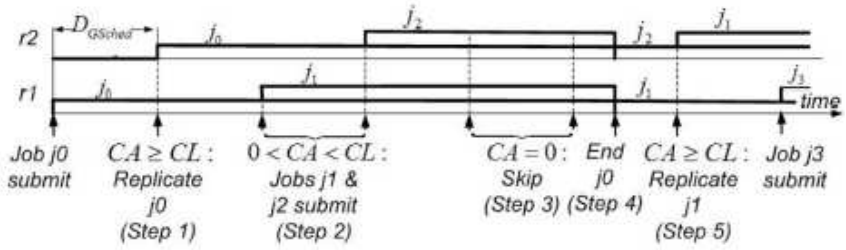

Figura 6.9: Exemplo de funcionamento do `FailureDependentRep`.

Em cada iteração, o escalonador da grade deve escalonar as tarefas para os recursos disponíveis. Nessa abordagem, essas tarefas podem ser de novas submissões de usuários ou novas tentativas após uma falha de recurso. Para realizar essa operação, o (`GSched`) consulta o serviço de informações (IS) a fim de obter o estado do sistema. Baseado nessas informações, CA e CL são comparados, onde CA é o número de CPUs ativas prontas para executar novas tarefas. E CL (*CPU limit*) especifica qual é a quantidade mínima de CPUs ativas livres que deve haver na grade para que se possa usar a replicação. Ou seja, caso a quantidade de CPUs ativas livres seja inferior a esse limite, não será utilizado replicação. O resultado dessa comparação determina a escolha da próxima tarefa a ser escalonada:

1 **se** $CA \geq CL$ **então**
2 | Seleciona a tarefa j que chegou primeiro e o número de réplicas
3 | menor que Rep_{max} (Passo 1);
4 **senão se** $0 < CA < CL$ **então**
5 | Seleciona a tarefa j que chegou primeiro e o número de réplicas
6 | menor que Rep_{min} (Passo 2);
7 **senão se** $CA = 0$ **então**
8 | Não realiza nenhum escalonamento
9 | e espera a próxima iteração (Passo 3);
10 **fim**

Figura 6.10: Algoritmo de comparação da abordagem `FailureDependentRep`.

Quando uma das tarefas duplicadas termina, a outra é automaticamente cancelada(Passo 4). Se a carga do sistema diminui antes que a uma tarefa termine de executar, novas réplicas dessa tarefa são automaticamente escalonadas (Passo 5).

Adaptive Checkpoint and Replication-Based Fault Tolerance (`CombinedFT`)

Nessa heurística é utilizada uma abordagem baseada em replicação e *checkpointing* adaptativos que troca dinamicamente as duas técnicas de acordo com informações sobre a carga do sistema obtidas em tempo de execução. Um exemplo da heurística `CombinedFT` é apresentado na figura 6.11.

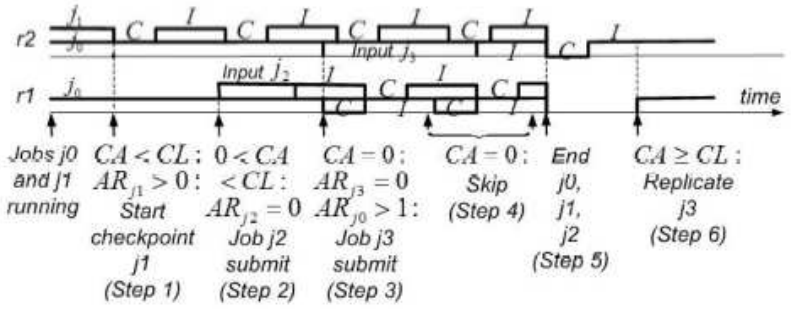

Figura 6.11: Exemplo de funcionamento do `CombinedFT`.

Nesse exemplo, o Rep_{min} e Rep_{max} são iniciados respectivamente com 1 e 2; e CL é igual a duas CPUs. Quando a disponibilidade de CPU é baixa $(CA < CL)$, o algoritmo é colocado em modo *checkpointing*. Nesse modo, se for necessário, o algoritmo pode cancelar as réplicas ativas de tarefa j (AR_j) fazendo o *checkpointing* da réplica remanescente. Ao processar o escalonamento da próxima tarefa j, as seguintes situações podem ocorrer:

```
1  se AR_j > 0 então
2      Inicia o checkpointing da réplica mais avançada
3      e cancela a execução das outras réplicas (Passo 1);
4  senão se AR_j = 0 and CA > 0 então
5      Escalona a tarefa j para executar em um dos recursos
6      livres (Step 2);
7  senão se AR_j = 0 and CA = 0 and ∃_i : AR_i > 1 então
8      Escolhe aleatoriamente uma tarefa i,
9      inicia o checkpointing da sua réplica mais avançada
10     e cancela as outras réplicas;
11     Submete a tarefa j para o melhor recurso liberado (Passo 3);
12 senão se AR_j = 0 and CA = 0 and ¬∃_i : AR_i > 1 então
13     Não realiza nenhum escalonamento
14     e espera a próxima iteração (Passo 3);
15
```

Figura 6.12: Algoritmo de comparação da abordagem `CombinedFT`.

Na primeira condição ($AR_j > 0$), o número de réplicas ativas da tarefa j é maior que zero, ou seja, se já existem outras réplicas dessa tarefa, deve ser feito o *checkpointing* da réplica mais avançada e cancelar as demais (Passo 1). Na segunda condição, quando a tarefa não possui nenhuma replica ativa ($AR_j = 0$) e existem CPUs prontas para executar novas tarefas ($CA > 0$), deve-se escalonar a tarefa j para uma dessas CPUs (Passo 2). Na terceira condição, quando a tarefa não possui nenhuma replica ativa ($AR_j = 0$) e não existem CPUs prontas para executar novas tarefas ($CA = 0$), mas existe alguma tarefa i que possui mais de uma replica ativa ($\exists_i : AR_i > 1$), deve-se escolher aleatoriamente uma dessas tarefas, fazer o *checkpointing* de sua réplica mais avançada e cancelar as demais a fim de que sejam liberadas CPUs para executar a nova tarefa j (Passo 3). Na última condição, quando nenhuma condição é satisfeita, ou seja, não existe nenhuma réplica ativa da tarefa, não existe CPU livre, e nem existe nenhuma tarefa executando com mais de uma réplica, a tarefa j terá de esperar para ser escalonada na próxima iteração.

O sistema volta para o modo replicação quando a carga do sistema diminui ($CA \geq CL$) (Passo 5). No modo replicação, todas as réplicas das tarefas são submetidas para os recursos livres, do mesmo modo como funciona a heurística *FailureDependentRep*. Se uma tarefa j que estava executando anteriormente em modo *checkpointing*, passa para o modo de replicação e uma de suas réplicas termina com sucesso, o *checkpoint* de j é descartado (Passo 6).

6.3 Comparação

Nesta seção apresentamos uma análise comparativa entre as estratégias de tolerância falhas para execução de aplicações em grades que foram descritas neste capítulo, incluindo também o nossa abordagem autonômica. Tomamos como base para comparação os seguintes critérios:

1. **Flexibilidade**: define se a abordagem fornece flexibilidade para que os usuários possam fazer configurações de parâmetros que definam como o mecanismo irá atuar para prover a tolerância a falhas de suas aplicações;

2. **Combinação de Técnicas**: é a capacidade para utilização de técnicas de tolerância a falhas de forma combinada: replicação com reinício, replicação com checkpointing, redundância com reinício, etc.;

3. **Classes de Aplicações**: determina as classes de aplicações que são levadas em consideração pelo mecanismo: regulares, paramétricas (BoT), paralelas fortemente acopladas e *workflows*;

4. **Auto-Tolerante a Falhas**: provê a tolerância a falhas do próprio mecanismo, ou seja, se a abordagem se preocupa em tratar falhas que podem ocorrer nos componentes que fazem parte do mecanismo;

5. **Extensibilidade de Técnicas**: permite que o usuário crie suas próprias técnicas de tolerância a falhas;

6. **Adaptação Dinâmica**: utilização de técnicas adaptativas que permite a reconfiguração dinâmica da estratégia de tolerância a falhas, sem que haja necessidade de qualquer intervenção humana;

7. **Estratégia de Decisão Incorporada**: incorpora conhecimento suficiente para fazer automaticamente a substituição das técnicas de tolerância a falhas de modo eficiente, em resposta às mudanças percebidas no contexto do ambiente de execução;

8. **MAPE-K**: mecanismo baseado no modelo arquitetural MAPE-K, largamente adotado em softwares autonômicos;

A tabela 6.1 apresenta o resultado da comparação efetuada. O primeiro critério abordado é a flexibilidade. Alguns modelos como o de Souza et al. e o de Hwang e Kesselman fazem uso desse critério concentrando todo o poder de decisão no usuário, enquanto que o modelo proposto por Guimarães permite que o usuário defina regras que possam tomar essa decisão antes que uma determinada tarefa do *workflow* comece a executar. Nossa abordagem oferece a opção de usuários configurarem vários parâmetros, como os intervalos utilizados para a tomada de decisão sobre a quantidade de réplicas a ser utilizada ou quando deve haver uma troca na técnica de tolerância a falhas.

O modelo proposto nesta dissertação, bem como os demais trabalhos apresentados, permitem de alguma forma combinar diferentes técnicas de tolerância a falhas. A maioria dos modelos utilizam duas ou mais técnicas ao mesmo tempo, exceto o de Chtepen et. al. que mesmo quando combina as heurísticas, trabalha em apenas um dos dois modos de sua abordagem: no modo replicação ou no modo *checkpointing*.

Nossa abordagem autonômica fornece suporte ao tratamento de falhas de duas classes de aplicações: regulares e paramétricas (BoT). Sendo que o tratamento mais adequado às aplicações do tipo *bag-of-task* durante o cancelamento de réplicas da abordagem é uma das contribuições deste trabalho. Os trabalhos de Souza et al., Wu et al. e Chtepen et. al. fornecem suporte apenas para aplicações regulares, enquanto que o de Hwang e Kesselman e o de Guimarães oferece esse suporte para *workflows*.

A fim de simplificar o estudo que nos levaria à concepção do modelo proposto neste trabalho, não nos preocupamos com falhas no mecanismo que irá implementar nosso modelo. Entretanto, esta é uma possibilidade real que devemos levar em consideração em trabalhos futuros. Dentre os principais trabalhos relacionados comparados, apenas o mecanismo proposto por Hwang e Kesselman levou em consideração esse critério. Para isso, eles utilizaram técnicas baseadas em

checkpointing. O mecanismo salva a informação sobre quais tarefas já concluíram, tão logo ele seja notificado disso. Quando o sistema é reiniciado, após uma falha, ele continua o processamento do *workflow* executando apenas as tarefas que não haviam concluído antes da falha.

O *framework* proposto por Guimarães considerou a capacidade da extensibilidade de técnicas, permitindo aos usuários desenvolverem as suas próprias técnicas de tolerância a falhas. Hwang e Kesselman apenas permitem que o usuário possa definir outros tratamentos de falhas que atuem no nível do *workflow*.

Quanto ao critério da adaptação dinâmica, o modelo apresentado por Wu et al. utiliza como regra para a tomada de decisão apenas suposições baseadas no número de ocorrência de falhas que uma tarefa sofre. No modelo de Guimarães, a escolha ocorre momentos antes da execução da próxima tarefa do *workflow*. Chtepen et al. utiliza uma espécie de escalonador *batch*, cujos ciclos de escalonamento são utilizados para realizar o monitoramento e adaptação. Em nossa abordagem, a tomada de decisão é realizada de acordo com os ciclos do componente autonômico que se inicia com o monitoramento. Alguns monitores utilizados realizam sua função de acordo com a necessidade da adaptação que tem que ser realizada, por exemplo o monitoramento da ocupação dos recursos é feita de forma periódica, podendo o usuário definir essa periodicidade, enquanto que o monitoramento da taxa de falhas da grade é feita quando o analisador quer diminuir a quantidade de réplicas. Ainda existem outros monitores que atuam quando certos eventos ocorrem. Essa estrutura permite que se tenha mais agilidade na tomada de decisão.

Com relação à estratégia de decisão estar incorporada ao mecanismo de tolerância a falhas, em nosso trabalho apresentamos várias heurísticas que compõem a base de conhecimento de nossa abordagem. Essas heurísticas especificam as regras que são baseadas nas informações monitoradas do ambiente como a taxa de falhas e o percentual de ocupação dos recursos. Essas regras estão incorporadas no nosso modelo e são utilizadas pelos monitores e analisadores para a tomada de decisões quanto às adaptações que devem ser realizadas. As heurísticas apresentadas em nosso trabalho foram inspiradas no trabalho de Chtepen et al., o qual também incorpora diversas outras heurísticas. Wu et. al. apresentou em seu trabalho uma árvore de decisão baseada no número de falhas das tarefas, a qual representa a base de conhecimento utilizada para tomada de decisão do mecanismo e está incorporada ao

modelo proposto. Embora o modelo proposto por Guimarães seja baseado em regras, estas devem ser fornecidas pelos usuários. As abordagens de Souza et al. e o de Hwang e Kesselman não apresentam nenhuma base de conhecimento.

Nosso trabalho foi baseado no modelo arquitetural MAPE-K que consiste em uma estrutura convencionada para o desenvolvimento de sistemas computacionais autonômicos. Essa arquitetura permitiu que o gerenciamento autonômico do nosso modelo fosse feito com o menor acoplamento possível em relação ao modelo de grade implementado no AGST. Além disso, essa estrutura permitiu que o ciclo autonômico de monitoramento, análise, planejamento e execução de nossa estratégia pudesse ser realizado de forma independente dos demais componentes da grade, o que torna o mecanismo mais ágil na percepção das mudanças de contexto e na tomada de decisão. Os demais trabalhos relacionados utilizaram arquiteturas desenvolvidas de forma individual.

Critérios	Trabalhos de Pesquisa					
	Souza et. al. [63]	**Hwang e Kesselman [33]**	**Wu et. al. [71]**	**Guimarães [25]**	**Chtepen et. al. [5]**	**Nossa abordagem**
Flexibilidade	sim	sim		sim	sim	
Combinação de Técnicas	sim	sim	sim	sim		sim
Classes de Aplicações	Regulares	*Workflows*	Regulares	*Workflows*	Regulares	Regulares e Paramétricas (BoT)
Auto-Tolerante a Falhas		sim				
Extensibilidade de Técnicas				sim		
Adaptação Dinâmica				sim	sim	sim
Estratégia de Decisão Incorporada			sim		sim	sim
MAPE-K						sim

Tabela 6.1: Quadro resumo dos principais trabalhos relacionados.

7 Conclusões e Trabalhos Futuros

Fornecer tolerância a falhas em grades computacionais oportunistas é uma tarefa desafiadora devido a diversas características presentes nestes ambientes computacionais, tais como a alta heterogeneidade e escalabilidade de recursos distribuídos, a autonomia de domínios administrativos e o dinamismo de recursos, tais como variações imprevisíveis na taxa de chegada de aplicações e de falhas de recursos. Estas características justificam a importância de investigar abordagens adaptativas de tolerância a falhas para as aplicações de grade.

Este trabalho apresentou uma estratégia autonômica que realize de forma eficiente a tolerância a falhas durante a execução das tarefas submetidas por usuários de grades oportunistas, tratando de forma mais adequada aplicações do tipo *bag-of-tasks*. A estratégia proposta foi modelada tendo como base uma das arquiteturas mais utilizadas no desenvolvimento de sistemas autonômicos: o modelo MAPE-K, que permite realizar as atividades de monitoramento, análise, planejamento e executar ações de reconfiguração. A criação desse modelo foi cuidadosamente projetado para levar em consideração o dinamismo de grades oportunistas, permitindo realizar dois níveis de adaptações, permitindo reconfigurações paramétricas nas estratégias de tolerância a falhas utilizadas e adaptações estruturais no mecanismo, substituindo completamente uma técnica por outra.

As principais contribuições deste trabalho foram:

- A definição de uma estratégia autonômica de tolerância a falhas para execução de aplicações em grades oportunistas capaz de obter dinamicamente informações sobre contexto de execução para efetuar a tomada de decisão e realizar adaptações paramétricas e estruturais, a fim de melhorar a eficiência das técnicas empregadas.

- Após a troca da técnica de replicação por checkpointing, pode ser necessário cancelar réplicas. O cancelamento de réplicas pode afetar drasticamente o sucesso na conclusão de aplicações como as *bag-of-tasks*. Em nossa estratégia

nos preocupamos em fornecer suporte a aplicações do tipo *bag-of-tasks*, que compreende a classe de aplicações mais utilizada em *desktop grids*.

- Acrescentamos à abordagem o tratamento do efeito ping-pong, causado por oscilações em torno dos pontos da tomada de decisão. Isso foi feito determinando-se margens de tolerância que evitam que abordagem faça reconfigurações inadequadas e sofra perda de performance.

- Foi desenvolvido um modelo da abordagem usando a ferramenta de simulação AGST, de modo que pudesse representar e permitir avaliar o comportamento de um gerenciador autonômico segundo a arquitetura MAPE-K, capaz de realizar as atividades de monitoramento, análise, planejamento e execução de ações de reconfiguração.

- Foram realizadas simulações em diferentes cenários que representassem possíveis condições do ambiente de execução de *desktop grids* oportunistas e que pudessem avaliar o comportamento da estratégia autonômica proposta em comparação às estratégias estáticas de tolerância a falhas para execução de aplicações nesses ambientes, utilizando-se como métricas o tempo médio de conclusão das aplicações e número de conclusões com sucesso das aplicações.

7.1 Trabalhos Futuros

A partir deste trabalho inicial, identificamos diversas possibilidades de trabalhos futuros que poderiam ser desenvolvidos, tais como:

- Neste trabalho utilizamos, para a detecção de eventos de monitoramento e para tomada de decisão no processo de análise e planejamento, representações do conhecimento baseadas em Regras. Entretanto, poderiam ser aplicadas outras técnicas que, tais como Funções de Utilidades e Aprendizagem por Reforço que foram descritas no capítulo 2 ou ainda outras técnicas que podem ser encontradas na literatura, tais como técnicas Bayesianas [26] e técnicas híbridas [68];

- Como a Universidade Federal do Maranhão faz parte de um projeto multi-institucional de grades de computadores (InteGrade), um dos trabalhos futuros consiste em implementar o modelo proposto, no *middleware* do InteGrade;

- Nosso trabalho, não utilizou nenhuma técnica de tolerância a falhas do nível de *workflow*. Entretanto, seria muito interessante que essas técnicas pudessem ser abordados no mecanismo de forma autonômica, levando em consideração dados de contexto do ambiente de execução da grade.

- A estratégia proposta foi projetada para funcionar de modo centralizado, contudo essa estrutura é bastante vulnerável, sendo um único ponto de falhas. Poderiam ser desenvolvido um gerenciador global que funcionasse de forma descentralizada com componentes se comunicam usando abordagens *pear-to-pear*;

- A estratégia autonômica proposta não leva em consideração a tolerância a falhas do próprio mecanismo. Uma possível abordagem seria, além de incluir componentes descentralizados, que esses componentes fossem capazes de detectar a falha dos outros e realizar sua recuperação.

- Realizar novas simulações da abordagem em outros cenários que utilizem *traces* de carga de trabalho e em cenários que possam permitir comparar a estratégia proposta neste trabalho com outras abordagens autonômicas;

Referências Bibliográficas

[1] S. Agarwala, Y. Chen, D. Milojicic, and K. Schwan. QMON: QoS- and Utility-Aware Monitoring in Enterprise Systems. In *Autonomic Computing, 2006. ICAC '06. IEEE International Conference on*, pages 124 – 133, Dublin, Ireland, June 2006.

[2] A. AuYoung, L. Rit, S. Wiener, and J. Wilkes. Service Contracts and Aggregate Atility Functions. *High-Performance Distributed Computing, International Symposium on*, 0:119–131, 2006.

[3] B. Bennett, E. Davis, T. Kunau, and W. Wren. Beowulf Parallel Processing for Dynamic Load-Balancing. In *Aerospace Conference Proceedings, 2000 IEEE*, volume 4, pages 389 –395, Big Sky, MT , USA, March 2000.

[4] R. Buyya and M. Murshed. Gridsim: A Toolkit for the Modeling and Simulation of Distributed Resource Management and Scheduling for Grid Computing. *Concurrency and Computation: Practice and Experience*, volume 14:pages 1175–1220, November 2002.

[5] M. Chtepen, F. H. A. Claeys, B. Dhoedt, F. De Turck, P. Demeester, and P. A. Vanrolleghem. Adaptive Task Checkpointing and Replication: Toward Efficient Fault-Tolerant Grids. *IEEE Trans. Parallel Distrib. Syst.*, 20:180–190, February 2009.

[6] M. Chtepen, B. Dhoedt, F. H. A. Claeys, F. D. Turck, P. Demeester, and P. A. Vanrolleghem. Dynamic Scheduling of Computationally Intensive Applications on Unreliable Infrastructures. 2006.

[7] L. Chwif and A. C. Medina. *Modelagem e Simulação de Eventos Discretos: Teoria & Aplicações*. Ed. dos Autores, São Paulo, Brasil, 2 edition, 2007.

[8] W. Cirne, F. Brasileiro, D. Paranhos, L. F. W. Góes, and W. Voorsluys. On the Efficacy, Efficiency and Emergent Behavior of Task Replication in Large Distributed Systems. *Parallel Computing*, 33:213–234, April 2007.

[9] D. Conde. Análise de Padrões de Uso em Grades Computacionais. Master's thesis, Universidade de São Paulo, São Paulo, SP – Brasil, 2008.

[10] Condor. *Online Manual of Condor Version 7.4.4*. University of Wisconsin-Madison, http://www.cs.wisc.edu/condor/manual/v7.4, Janeiro 2004.

[11] I. Corp. An architectural blueprint for autonomic computing. 2003.

[12] G. Cunha Filho. OGST (Opportunistic Grid Simulation Tool): uma ferramenta de simulação para avaliação de estratégias de escalonamento de aplicações em grades oportunistas. Master's thesis, Universidade Federal do Maranhão, São Luís, MA – Brasil, 2009.

[13] G. Cunha Filho and F. J. da Silva e Silva. Ogst: An Opportunistic Grid Simulation Tool. In *LAGrid 2008: 2nd International Workshop Latin American Grid*, Campo Grande, Mato Grosso do Sul, Agosto 2008.

[14] F. J. da Silva e Silva, F. Kon, A. Goldman, M. Finger, de Raphael Y. de Camargo, F. C. Filho, and F. M. Costa. Application execution management on the InteGrade opportunistic grid middleware. *J. Parallel Distrib. Comput.*, 70:573–583, May 2010.

[15] R. Y. de Camargo. *Armazenamento Distribuído de Dados e Checkpointing de Aplicações Paralelas em Grades Oportunistas*. PhD thesis, Universiadade de São Paulo, São Paulo, SP – Brasil, Agosto 2007.

[16] B. de Tácio Pereira Gomes, G. C. Filho, I. R. Campos, J. F. Gonçalves, and F. J. da Silva e Silva. Scheduling Strategies Evaluation for Opportunistic Grids. *Computing Systems, Symposium on*, 0, Outubro 2010.

[17] B. de Tácio Pereira Gomesa e Silva and F. J. da Silva e Silva. Agst - Autonomic Grid Simulation Tool. A simulator of autonomic functions based on the MAPE-K model. In *Proceedings of 1st International Conference on Simulation and Modeling Methodologies, Technologies and Applications (SIMULTECH)*, Noordwijkerhout, The Netherlands, Julho 2011.

[18] J. Dowling, R. Cunningham, E. Curran, and V. Cahill. Building Autonomic Systems Using Collaborative Reinforcement Learning. *Knowledge Engineering Review*, 21:231–238, September 2006.

[19] S. C. e Renato Cerqueira. *Computação Autônoma: Conceitos, Infra-estruturas e Soluções em Sistemas Distribuídos*. SBRC, 2009.

[20] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. *ACM Comput. Surv.*, 34:375–408, September 2002.

[21] E. Fenson and R. Howard. Reinforcement learning for autonomic network repair. In *Proceedings of the First International Conference on Autonomic Computing*, pages 284–285, Washington, DC, USA, 2004. IEEE Computer Society.

[22] F. J. S. S. Gilberto Cunha Filho. Ogst: An Opportunistic Grid Simulation Tool. In *Proceedings of 2009 The Latin American Grid (LAGrid)*, LAGrid '08, pages 398–407, Washington, DC, USA, 2008. IEEE Computer Society.

[23] A. Goldchleger, F. Kon, A. Goldman, M. Finger, and G. C. Bezerra. Integrade Object-Oriented Grid middleware Leveraging the Idle Computing Power of Desktop Machines: Research Articles. *Concurr. Comput.: Pract. Exper.*, 16:449–459, April 2004.

[24] A. Goodloe and L. Pike. Monitoring distributed real-time systems: A survey and future directions. Technical Report NASA/CR-2010-216724, NASA Langley Research Center, July 2010. Available at `http://ntrs.nasa.gov/search.jsp?R=278742&id=3&as=false&or=false&qs=Ns%3DArchiveName%257c0%26N%3D4294643047`.

[25] F. P. Guimarães. Framework para Execução Adaptativa e Tolerante a Falhas de workflows em grid. Master's thesis, Universidade de Brasília, Brasília, DF – Brasil, Outubro 2010.

[26] H. Guo. A Bayesian Approach for Automatic Algorithm Selection, 2003.

[27] S. Hallsteinsen. Madam - theory of adaptation. Technical report, SINTEF ICT, 2010.

[28] S. Hariri, B. Khargharia, H. Chen, J. Yang, Y. Zhang, M. Parashar, and H. Liu. The Autonomic Computing Paradigm. *Cluster Computing*, 9:5–17, Janeiro 2006.

[29] F. Howell and R. McNab. Simjava: A Discrete Event Simulation Library for Java. *International Conference on WebBased Modeling and Simulation*, 30:51–56, 1998.

[30] M. C. Huebscher and J. A. McCann. A Survey of Autonomic Computing - Degrees, Models, and Applications. *ACM Computing Surveys*, 40:7:1–7:28, August 2008.

[31] M. C. Huebscher and J. A. McCann. A survey of autonomic computing degrees, models, and applications. *ACM Comput. Surv.*, 40:7:1–7:28, August 2008.

[32] M. C. Huebscher, J. A. McCann, and A. Hoskins. Context as autonomic intelligence in a ubiquitous computing environment. *Int. J. Internet Protoc. Technol.*, 2:30–39, December 2007.

[33] S. Hwang and C. Kesselman. Gridworkflow: A flexible failure handling framework for the grid. In *Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing*, HPDC '03, pages 126–, Washington, DC, USA, 2003. IEEE Computer Society.

[34] IBM. *An architectural blueprint for autonomic computing*, 2003.

[35] A. Iosup, H. Li, M. Jan, S. Anoep, C. Dumitrescu, L. Wolters, and D. H. J. Epema. The Grid Workloads Archive. *Future Gener. Comput. Syst.*, 24:672–686, July 2008.

[36] P. Jalote. *Fault Tolerance in Distributed Systems*. Prentice Hall, April 1994.

[37] T. Kelly and T. Kelly. Utility-directed allocation. In *First Workshop on Algorithms and Architectures for Self-Managing Systems*, San Diego, CA, USA, Junho 2003.

[38] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36:41–50, Janeiro 2003.

[39] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. marc Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP*. SpringerVerlag, 1997.

[40] G. Kiczales and J. D. Rivieres. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, USA, 1991.

[41] D. Klusáček, L. Matyska, and H. Rudová. Alea: Grid Scheduling Simulation Environment. In *Proceedings of the 7th international conference on Parallel processing*

and applied mathematics, PPAM'07, pages 1029–1038, Berlin, Heidelberg, 2008. Springer-Verlag.

[42] D. Kondo, B. Javadi, A. Iosup, and D. Epema. The Failure Trace Archive: Enabling Comparative Analysis of Failures in Diverse Distributed Systems. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID '10, pages 398–407, Washington, DC, USA, 2010. IEEE Computer Society.

[43] K. Kurowski, J. Nabrzyski, A. Oleksiak, and J. Weglarz. Grid scheduling simulations with gssim. In *Proceedings of the 13th International Conference on Parallel and Distributed Systems - Volume 02*, pages 1–8, Washington, DC, USA, 2007. IEEE Computer Society.

[44] M. Léger, T. Ledoux, and T. Coupaye. Reliable dynamic reconfigurations in the fractal component model. In *Proceedings of the 6th international workshop on Adaptive and reflective middleware: held at the ACM/IFIP/USENIX International Middleware Conference*, ARM '07, pages 3:1–3:6, New York, NY, USA, 2007. ACM.

[45] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny. Checkpoint and migration of UNIX processes in the Condor distributed processing system. Technical Report UW-CS-TR-1346, University of Wisconsin - Madison Computer Sciences Department, April 1997.

[46] Y. Liu. Grid scheduling. Technical report, Department of Computer Science, University of Iowa, 2004. http://www.cs.uiowa.edu/ yanliu/QE/QEreview.pdf, Acessado em: 22/01/2009.

[47] U. Lublin and D. G. Feitelson. The Workload on Parallel Supercomputers: Modeling the Characteristics of Rigid Jobs. *J. Parallel Distrib. Comput.*, 63:pages 1105–1122, November 2003.

[48] P. Maes. Concepts and experiments in computational reflection. In *Conference proceedings on Object-oriented programming systems, languages and applications*, OOPSLA '87, pages 147–155, New York, NY, USA, 1987. ACM.

[49] M. Mansouri-Samani. *Monitoring of Distributed Systems*. PhD thesis, University of London. Imperial College of Science, Technology and Medicine, December 1995.

[50] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. C. Cheng. Composing adaptive software. *Computer*, 37:56–64, July 2004.

[51] R. Murch. *Autonomic Computing*. IBM Press/Prentice-Hall, 1nd edition, 2004.

[52] M. A. S. Netto and R. Buyya. Offer-based Scheduling of Deadline-Constrained Bag-of-Tasks Applications for Utility Computing Systems. In *Proceedings of the International Heterogeneity in Computing Workshop (HCW), in conjunction with the 23rd IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[53] M. A. S. Netto and R. Buyya. Coordinated Rescheduling of Bag-of-Tasks for Executions on Multiple Resource Providers. *Concurrency and Computation: Practice and Experience*, 2011.

[54] N. D. Palma, D. P. Laumay, and L. Bellissard. Ensuring dynamic reconfiguration consistency. In *In 6th International Workshop on Component-Oriented Programming (WCOP 2001), ECOOP related Workshop*, pages 18–24, 2001.

[55] J. Pan. Software Testing. *Carnegie Mellon University*, 1999.

[56] D. Paranhos, W. Cirne, and F. Brasileiro. Trading Cycles for Information: Using Replication to Schedule Bag-of-Tasks Applications on Computational Grids. In *Proceedings of the Euro-Par 2003: International Conference on Parallel and Distributed Computing*, Klagenfurt, Austria, August 2003. IEEE Computer Society.

[57] M. Parashar and S. Hariri. Autonomic computing: An overview. In *Unconventional Programming Paradigms*, pages 247–259. Springer Verlag, 2005.

[58] M. Parashar, H. Liu, Z. Li, V. Matossian, C. Schmidt, G. Zhang, and S. Hariri. Automate: Enabling Autonomic Applications on the Grid. *Cluster Computing*, 9:161–174, April 2006.

[59] J. Pruyne and M. Livny. Managing Checkpoints for Parallel Programs. In *Workshop on Job Scheduling Strategies for Parallel Processing (IPPS '96)*, Honolulu, HI, April 1996.

[60] C. Roblee and G. Cybenko. Implementing large-scale autonomic server monitoring using process query systems. In *Proceedings of the Second International*

Conference on Automatic Computing, pages 123–133, Washington, DC, USA, 2005. IEEE Computer Society.

[61] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ, 2nd edition edition, 2003.

[62] S. S. Sathya and K. S. Babu. Survey of Fault Tolerant Techniques for Grid. *Computer Science Review*, 4:101–120, 2010.

[63] S. A. d. Sousa, F. J. d. S. e. Silva, and R. F. Lopes. A Flexible Fault-Tolerance Mechanism for the Integrade Grid Middleware. In *Proceedings of the Third International Conference on Networking and Services*, pages 26–, Washington, DC, USA, 2007.

[64] C. SOUZA and C. MAZIERO. Intercessão em tempo de implantação uma abordagem reflexiva para a plataforma j2ee. *XV Simpósio Brasileiro de Engenharia de Software*, 1:286–301, Outubro 2001.

[65] J. E. Souza and F. Castor. Um Detector de Defeitos Cumulativo Baseado em uma Abordagem Difusa. In *Anais do 28o Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBPC 2010)*, Gramado, Brazil, 2010.

[66] W. Stallings. *Wireless Communications and Networks*. Prentice Hall Professional Technical Reference, 1st edition, 2001.

[67] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[68] G. Tesauro, N. K. Jong, R. Das, and M. N. Bennani. A hybrid reinforcement learning approach to autonomic resource allocation. In *Proceedings of the 2006 IEEE International Conference on Autonomic Computing*, pages 65–73, Washington, DC, USA, 2006. IEEE Computer Society.

[69] R. J. Walker, E. L. A. Baniassad, and G. C. Murphy. An initial assessment of aspect-oriented programming. Technical report, Vancouver, BC, Canada, Canada, 1998.

[70] S. Whiteson and P. Stone. Evolutionary function approximation for reinforcement learning. *J. Mach. Learn. Res.*, 7:877–917, December 2006.

[71] Y. Wu, Y. Yuan, G. Yang, and W. Zheng. An Adaptive Task-Level Fault-Tolerant Approach to Grid. *J. Supercomput.*, 51:97–114, Fevereiro 2010.

[72] J. Yu and R. Buyya. A Taxonomy of Workflow Management Systems for Grid Computing. *Journal of Grid Computing*, pages 171–200, 2005.

[73] J. Zhang and R. J. Figueiredo. Autonomic feature selection for application classification. In *Proceedings of the 2006 IEEE International Conference on Autonomic Computing*, pages 43–52, Washington, DC, USA, 2006. IEEE Computer Society.