Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

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

Felipe Pontes Guimarães

Brasília
2010

Livros Grátis

http://www.livrosgratis.com.br

Milhares de livros grátis para download.

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

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

Felipe Pontes Guimarães

Dissertação apresentada como requisito parcial
para conclusão do Programa de Pós-Graduação em Informática

Orientadora
Prof.ª Dr.ª Alba Cristina Magalhães Alves de Melo

Brasília
2010

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Programa de Pós-Graduação em Informática

Coordenador: Prof. Dr. Maurício Ayala Rincón

Banca examinadora composta por:

Prof.ª Dr.ª Alba Cristina Magalhães Alves de Melo (Orientadora) — CIC/UnB
Prof. Dr. Alfredo Goldman vel Lejbman — IME/USP
Prof.ª Dr.ª Genaína Nunes Rodrigues — CIC/UnB

CIP — Catalogação Internacional na Publicação

Guimarães, Felipe Pontes.

Framework para Execução Adaptativa e Tolerante a Falhas de Workflows em Grid / Felipe Pontes Guimarães. Brasília : UnB, 2010.
91 p. : il. ; 29,5 cm.

Dissertação (Mestrado) — Universidade de Brasília, Brasília, 2010.

1. Grid, 2. Tolerância a falhas, 3. workflow

CDU 004.8

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro — Asa Norte
CEP 70910-900
Brasília–DF — Brasil

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

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

Felipe Pontes Guimarães

Dissertação apresentada como requisito parcial
para conclusão do Programa de Pós-Graduação em Informática

Prof.ª Dr.ª Alba Cristina Magalhães Alves de Melo (Orientadora)
CIC/UnB

Prof. Dr. Alfredo Goldman vel Lejbman
IME/USP

Prof.ª Dr.ª Genaína Nunes Rodrigues
CIC/UnB

Prof. Dr. Maurício Ayala Rincón
Coordenador do Programa de Pós-Graduação em Informática

Brasília, 14 de Outubro de 2010

Dedicatória

Dedico esta dissertação à minha mãe, que desde o primeiro dia me incentivou e me deu todo o suporte necessário para que este trabalho fosse realizado e à minha namorada Verônica, que me motiva, me alegra e está sempre ao meu lado em todas as minhas empreitadas.

Agradecimentos

Meus mais sinceros agradecimentos à Prof.ª Dr.ª Alba que me recebeu de braços abertos em um momento de necessidade e me orientou nos caminhos da pesquisa acadêmica; à Prof.ª Dr.ª Célia e ao meu caríssimo amigo Albert Il Pak que dividiu comigo a experiência do mestrado.

Resumo

A computação em Grid proporciona a seus usuários o compartilhamento de recursos autônomos e heterogêneos para solucionar problemas computacionais de grande complexidade. Em um Grid, os recursos possuem autonomia, logo podem entrar e sair do mesmo conforme suas necessidades. A computação em Grid é frequentemente usada para executar *workflows* científicos, que são uma rede de passos necessários à análise de grande volume de dados. Geralmente, a execução de *workflows* científicos é demorada, podendo levar vários minutos, várias horas ou mesmo dias. Ao se associar essas duas características - um ambiente dinâmico e *workflows* de longa duração - surge um problema: não há como se impedir que os recursos saiam do Grid durante a execução de tarefas de um *workflow*, causando assim um erro na execução. Não se pode, no entanto, permitir que tais erros inviabilizem a execução do *workflow*. Para contornar esse problema existem técnicas de tolerância a falhas, que procuram garantir que, mesmo em face de falhas na execução de algumas tarefas, o *workflow* como um todo será executado corretamente. Vários trabalhos lidam com técnicas de tolerância a falhas para *workflows* em Grid e várias técnicas diferentes já existem. No entanto, nenhuma das abordagens estudadas considera, em conjunto, as preferências do usuário e a situação atual do Grid. A presente dissertação de mestrado propõe e avalia um *framework* de execução adaptativa tolerante a falhas que permite ao usuário definir as regras pelas quais a seleção das técnicas de tolerância a falhas será realizada em tempo de execução e também permite a adição de novas técnicas de tolerância a falhas. Os resultados experimentais obtidos em um Grid com 5 máquinas mostram que o *framework* proposto de fato permite a definição de regras pelo usuário e a inclusão de novas técnicas de tolerância a falhas. Além disso, a sobrecarga no tempo de execução dos *workflows* foi baixo: cerca de 2%, na plataforma avaliada.

Palavras-chave: Grid, Tolerância a falhas, workflow

Abstract

Grid computing allows its users to share autonomous and heterogeneous resources to solve highly complex computational problems. It creates an extremely dynamic environment, in which the resources may enter or leave at any given moment according to their needs. One of the major uses of Grid computing is the execution of scientific workflows, a set of necessary steps for analyzing great amounts of data. The execution time of these workflows may vary from several minutes to days. Once we combine these two characteristics - a dynamic environment and long execution times - a problem arises, since there is no mechanism to prevent resources from leaving the Grid during the execution of a task belonging to a workflow, thus introducing an error in the execution. However, the ocurrence of these errors must not make unfeasible the workflow execution. To deal with this issue fault tolerance techniques have been proposed. They allow for correct workflow execution even when facing errors during a number of faults in intermediary tasks. Many published papers deal with fault tolerance techniques for workflow execution in the Grid, but none of the studied approaches consider both the user preferences and the current status of the Grid. The current Master's Thesis proposes and evaluates a framework that provides adaptive fault tolerant execution for workflows in Grids allowing the user to define the rules by which the fault tolerance techniques will be chosen at runtime. Moreover, it allows the addition of new fault tolerance techniques. The experimental results obtained from a 5-machine Grid show that the framework is able to allow the user to set the rules and add new fault tolerance techniques at the cost of a very low overhead in the execution time: around 2% in the evalution platform,

Keywords: Grid, fault tolerance, workflow

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

1.1 Motivação

Por volta de 1990, criou-se uma nova arquitetura de computação distribuída: a computação em grade ou *Grid computing* [34]. Esta arquitetura busca integrar recursos heterogêneos e compartilhados entre seus usuários de forma a possibilitar a resolução de problemas de grande complexidade.

Como a computação em Grid lida com recursos compartilhados, algumas características lhe são peculiares. Do ponto de vista dos recursos, o Grid possui recursos autônomos e heterogêneos que são compartilhados segundo políticas de controle bem definidas. Do ponto de vista das informações disponíveis, não há um conjunto predeterminado e constantemente atualizado. Além disso, as operações de submissão, término e controle da execução são mediadas pelo *middleware*. Um dos aspectos da computação em Grid que deve ser considerado é o seu grande dinamismo, com recursos frequentemente entrando e saindo do Grid por diversos motivos, como falha ou mudança da política local [15].

Diversas aplicações criadas para a computação em Grid são expressas através de *workflows* científicos [14], que são redes de passos analíticos que representam os passos necessários à análise de grandes volumes de dados [34]. Frequentemente os *workflows* científicos podem demorar horas ou mesmo dias para executar. De fato, em [14] é apresentado um exemplo de *workflow* cujo tempo de duração foi de 11 horas e 24 minutos.

Um dos desafios da execução de *workflows* em Grid é a alocação eficiente das tarefas aos recursos. O dinamismo do Grid faz com que realizar a alocação de todas as tarefas para os recursos disponíveis antes de iniciar a execução não seja uma boa opção. Ou seja, um recurso disponível no momento da decisão pode não estar disponível no momento da execução. Pode-se mesmo dizer que adiar ao máximo a seleção do recurso que irá realizar uma dada tarefa seja algo muito interessante [15].

Os *workflows* possuem entre seus requisitos um alto grau de confiabilidade e a execução destacada, *ie*, não é necessária uma conexão constante com o mecanismo de controle -. Não seria aceitável que, no exemplo citado, um erro após 11 horas de execução fizesse com que todo o trabalho realizado fosse desperdiçado.

No entanto, dadas as características do Grid e a rotineira longa duração dos *workflows*, a execução de *workflows* no Grid acaba frequentemente se deparando com falhas como a retirada de um recurso do Grid pelo administrador de tal recurso. E, se tratando do

ambiente de Grid, isto não deve ser encarado como um defeito mas como um desafio decorrente do compartilhamento de recursos entre diferentes entidades e/ou organizações.

Para amenizar o efeito de tais falhas, vários *middlewares* de Grid implementam técnicas de tolerância a falhas. Diz-se que um sistema é tolerante a falhas se o seu funcionamento não é afetado pela ocorrência de falhas em sua execução [41]. Para poder fornecer tal característica, é necessário que exista algum grau de redundância física, informações ou tempo.

Atualmente, já existem várias técnicas para implementar a redundância dentre as quais poderíamos citar o *retry* que consiste em tentar novamente a execução em caso de falhas (redundância de tempo), a replicação ativa que consiste em executar a mesma tarefa em diferentes recursos na expectativa de que pelo menos uma seja concluída com sucesso (redundância física) e a técnica de replicação passiva que salva pontos seguros da execução permitindo que, caso necessário, outro recurso assuma a execução daquela tarefa a partir daquele ponto (redundância de tempo). A seleção da técnica mais apropriada depende de vários fatores. Em [13], os autores apresentam uma comparação na qual uma dada técnica possuía um desempenho melhor ou pior dependendo de variáveis do ambiente.

Na literatura, foram propostos trabalhos que visam fornecer a capacidade de tolerância a falhas para *workflows* em Grid [3, 13, 26, 31, 40, 49, 51]. No entanto, em nenhum dos trabalhos avaliados a relação entre o desempenho da técnica de tolerância a falhas e variáveis do ambiente foi considerada. Além disso, nenhum dos trabalhos pesquisados permitia a inclusão de novas técnicas pelos usuários.

1.2 Principais contribuições

O objetivo da presente dissertação de mestrado é propor e avaliar um *framework* para a execução adaptativa e tolerante a falhas de *workflows* em Grid. Dentre as principais contribuições poderíamos citar:

Definição de regras para a seleção das técnicas de tolerância a falhas: Ao contrário do que existe nos trabalhos estudados [3, 13, 26, 31, 40, 49, 51], o *framework* proposto neste trabalho permite que o usuário defina regras que relacionem a escolha das técnicas de tolerância a falhas à situação do Grid. Desta forma, tais escolhas podem ser realizadas em tempo de execução, com informações atualizadas acerca dos recursos disponíveis e segundo regras definidas pelo usuário. A definição de regras difere do que foi apresentado em [26] por não atrelar a técnica de tolerância a falhas à tarefa antes do início da execução do *workflow* diretamente, mas sim definir as regras pelas quais esta seleção será realizada.

Inclusão de novas técnicas de tolerância a falhas: O usuário é capaz de incluir novas técnicas de tolerância a falhas conforme sua necessidade. Nenhum dos trabalhos estudados fornecia esta funcionalidade ao usuário.

1.3 Estrutura do Documento

O presente documento está organizado da seguinte maneira. O capítulo 2 apresenta os principais conceitos de computação em Grid. Ele descreve como se dá o mapeamento das tarefas para os recursos disponíveis, um histórico da evolução da arquitetura do Grid, apresenta a arquitetura do *middleware* Integrade utilizado no *framework* proposto. No capítulo 3, introduz-se os conceitos de tolerância a falhas, seus modelos, bem como discorre-se sobre o uso de redundância para tolerância a falhas. Além disso, o capítulo 3 também trata do processamento de erros e da recuperação do sistema. Em seguida, apresentam-se no capítulo 4 os requisitos básicos de um *workflow* e diversas abordagens para realizar o plano de execução deste. Depois, no capítulo 5, há uma revisão do estado da arte sobre execução tolerante a falhas de *workflows* em Grid. Nesse capítulo diversos trabalhos são apresentados e comparados. No capítulo 6 apresentamos o *framework* proposto nesta dissertação: suas características, arquitetura, módulos e um exemplo de execução. No capítulo 7 são apresentados e discutidos os resultados obtidos no uso do *framework*. Finalmente, o capítulo 8 apresenta a conclusão e propostas de trabalhos futuros.

Capítulo 2

Computação em Grid

O termo "Grid" foi criado por volta de 1990 para denotar, àquela época, o que era a proposta de uma nova arquitetura para computação distribuída [34]. Seus conceitos e tecnologias foram pensados de forma a permitir compartilhamento de recursos de vários tipos para favorecer a colaboração científica.

Atualmente, um Grid é definido como um sistema distribuído e escalável que possibilita o compartilhamento, seleção e agregação de recursos autônomos e heterogêneos pertencentes a diferentes domínios para a resolução de problemas de alta complexidade computacional em um contexto dinâmico e adaptável [7].

Ele provê serviços, sejam eles computacionais, de dados, de aplicação, de informação ou de conhecimento permitindo uma otimização da infra-estrutura, aumento da colaboração e alta disponibilidade.

Por, no entanto, se tratar de um compartilhamento de máquinas e recursos, políticas de controle se tornam extremamente necessárias [34]. Cada usuário ou grupo deve necessariamente definir regras claras definindo o que, quando, como e com quem serão compartilhados seus recursos. O conjunto de usuários que utilizam o Grid segundo essas políticas é chamado de Organização Virtual (*Virtual Organization* - VO) [1, 21]. Outra definição da VO é uma entidade abstrata que agrupa usuários, instituições e recursos em torno de um mesmo objetivo.

As VO's podem variar enormemente em termos de escopo, tamanho, duração, estrutura, distribuição, recursos sendo compartilhados, público-alvo, entre outros. Porém, a despeito dessa imensa diversidade, alguns requisitos e preocupações lhes são comuns.

1. Criação e manutenção de relacionamentos de compartilhamento que permitam expressar, por exemplo, relações cliente-servidor, *peer-to-peer* (P2P), ou mais complexas como o compartilhamento intermediado, entre outras;
2. Controles de alto nível definindo as regras de compartilhamento dos recursos;
3. Elementos básicos para descoberta, alocação, gestão de grande diversidade de recursos e;
4. Diversos modos de operação.

Mais recentemente, o Grid incorporou em sua arquitetura mais alguns conceitos importantes: orientação a serviços, integração e virtualização .

Serviços são entidades de software que interagem com as aplicações por meio de um modelo de comunicação fracamente acoplado baseado em mensagens [6]. De acordo com a sequência de mensagens trocadas, os serviços realizam operações. Esta definição apenas em função das mensagens trocadas garante flexibilidade no que tange a implementação e a localização dos mesmos.

Os serviços podem eles próprios realizar o atendimento das requisições, prover um serviço de virtualização, ou seja, uma interface única para múltiplos serviços equivalentes, ou mesmo prover a integração de diversos serviços de forma a prover a funcionalidade desejada.

Finalmente, para viabilizar tanto a virtualização de serviços quanto a integração, é necessário um mecanismo de descoberta, negociação, alocação, monitoramento e gestão dos recursos, sejam eles físicos ou serviços.

Tendo esta estrutura formada e disponível, o próximo passo naturalmente seria utilizá-la para a execução dos algoritmos desejados. Porém isto também não ocorre de maneira trivial. O cenário descrito nos leva a um novo desafio: o escalonamento de tarefas no Grid.

Um dos aspectos que o ambiente de Grid tem que considerar é a gestão e o escalonamento de recursos. Tradicionalmente ela ocorre em ambientes onde o sistema de escalonamento possui o controle dos recursos, o que não é verdade em um ambiente de Grid. No Grid, o pano de fundo são recursos heterogêneos, com pouco ou nenhum grau de controle do gerenciador e com incontáveis diferenças nas políticas resultantes.

O escalonamento do Grid (*Grid Scheduling*) é definido em [34] como o processo de tomada de decisão envolvendo recursos de múltiplos domínios administrativos. Pode ser realizado em busca de uma ou mais máquinas onde se possa executar uma determinada tarefa (*job*). Um ponto que merece destaque é a diferenciação na abordagem do *local scheduler* da abordagem do *Grid Scheduler*. Enquanto o primeiro é "dono" dos recursos que ele utiliza, o *Grid Scheduler* não possui autonomia ou controle sobre os recursos utilizados, o que o leva a fazer escolhas buscando seu "*best effort*", submetendo então as tarefas aos recursos selecionados, sem ter conhecimento das tarefas que lhe serão submetidas posteriormente.

Outra diferença importante entre os *local schedulers* e os *Grid schedulers* é a disponibilidade de informações. Num contexto local assume-se que um conjunto de informações pré-definido esteja sempre disponível e correto. Já no ambiente de Grid esta premissa não é válida, pois cada recurso pode ter um grau diferente de detalhamento ou um período distinto de atualização e correção das informações ali presentes.

Além disso, o processo de obtenção das informações não necessariamente se dá diretamente junto ao recurso, mas intermediado por um Serviço de Informações do Grid (*Grid Information Service*, GIS). O GIS é o elemento responsável por recolher as informações dos recursos e disponibilizá-las aos Grid Schedulers. É a partir destas informações que se cria um escalonamento de tarefas no Grid.

Os *workflows* são compostos de diversas tarefas conectadas de acordo com suas dependências. Eles podem ser representados através de Grafos Direcionados Acíclicos (*Directed Acyclic Graphs* - DAG), dado que não contenham ciclos [50]. Nestes casos, os nós representam as tarefas e os arcos representam as dependências temporais entre as tarefas.

O DAG de um *workflow* pode representar as sequências, o paralelismo e, em um *workflow* estruturado por escolhas, as escolhas de tarefas. A sequência de tarefas diz

respeito à ordem e às precedências que devem ser respeitadas durante a execução. O paralelismo determina quais tarefas podem ser executadas de forma concorrente. Já as escolhas definem as alternativas de tarefas, onde um conjunto de uma ou mais tarefas pode ser utilizado em detrimento de outro [50].

2.1 Execução de Tarefas em Grid

O processo de execução de tarefas em Grid pode ser dividido em três fases [34]: (i) descoberta de recursos, que visa identificar um conjunto de recursos disponíveis para utilização; (ii) seleção de sistema, que consiste na recuperação de informações dinâmicas e estáticas sobre os recursos disponíveis encontrados e na escolha de um conjunto de recursos adequados à execução da tarefa; e (iii) a execução de tarefas propriamente ditas.

2.1.1 Descoberta de Recursos

A primeira fase é chamada de descoberta de recursos (*resource discovery*), que consiste no levantamento dos recursos que podem ser utilizados por um dado usuário e incorre na seleção de um conjunto de recursos que serão analisados mais apuradamente nas fases posteriores.

A descoberta de recursos se dá em 3 passos. O primeiro passo é o filtro por autorização. Isto se dá determinando exatamente em quais recursos o usuário tem direito de submissão de tarefas, condição primordial para viabilizar a utilização do recurso.

O segundo passo desta fase é a definição dos requisitos mínimos para execução da tarefa. Estes podem ser definidos em termos de atributos estáticos como sistema operacional, hardware ou arquitetura de processador e em termos de atributos dinâmicos como memória RAM disponível, nível de carga, grau de conectividade necessário ou espaço disponível em disco.

De posse portanto da listagem de recursos passíveis de utilização e dos requisitos mínimos para seleção, segue-se para o terceiro e último passo desta fase que consiste no filtro por requisitos mínimos, onde se descarta da listagem de candidatos à seleção todos os recursos que não atendam aos requisitos mínimos. Ao final deste passo, a saída da primeira fase da seleção de recursos será um conjunto menor de recursos que devam ser investigados mais cuidadosamente.

2.1.2 Seleção de Sistema

Segue-se então para a segunda fase do processo conhecida como Seleção de Sistema. Esta fase recebe portanto um conjunto de recursos aptos a executar a tarefa e seleciona dentro deste um recurso que irá efetivamente executá-la. Esta fase é, em geral, realizada em dois passos: levantamento de informações dinâmicas e seleção de sistema.

No primeiro passo, são obtidos requisitos mais detalhados e de caráter dinâmico sobre os recursos. Isto, contudo, depende de dois aspectos: quais informações estão disponíveis e como obtê-las. A relação das informações disponíveis e seu conteúdo são usualmente obtidas do GIS ou mesmo do *local scheduler*. refGlau32 Finalmente, no segundo passo, que é o próprio escalonamento de recursos do Grid, dá-se a escolha do sistema a ser usada considerando os dados recolhidos no passo anterior.

Finalmente, no segundo passo, ocorre o escalonamento de recursos do Grid. Este passo consiste em fazer um mapeamento de tarefas a um grupo de recursos disponíveis, possivelmente pertinentes a diferentes domínios administrativos, que serão responsáveis por executá-las [16]. Para realizar esse mapeamento utilizam-se os dados recolhidos no passo anterior.

Uma observação importante: para que o Grid possa atingir um de seus objetivos - agregar diferentes recursos e prover aos usuários serviços não-triviais - torna-se uma parte fundamental do sistema um escalonador eficiente [16] porém este é um problema NP-Completo e as soluções providas são *ad hoc* e específicas de domínio [17].

2.1.3 Execução da Tarefa

Tendo uma tarefa a ser realizada em um recurso já especificado, segue-se então para a fase de execução da tarefa. Nesta fase, existe um passo preliminar, de cárater opcional, conhecido por Reserva Avançada. Este passo é extremamente dependente do recurso em particular,que definirá o grau de dificuldade para realização da reserva, se ela é feita de forma automática ou manual, se há e qual é o período de validade da reserva e a penalidade aplicada caso esta não seja respeitada.

Em seguida, é realizado o passo mais importante do processo, a submissão da tarefa. Apesar de ter uma aparência simplista, este passo possui nuances e complicações decorrentes da falta de padronização para a submissão, onde alguns sistemas como o *Globus Resource Allocation Manager* (Globus GRAM) [18] encapsulam o escalonamento local mas mantendo uma severa dependência com relação aos parâmetros locais.

Prosseguimos portanto para as tarefas de preparação, que podem envolver *setup*, utilização da reserva ou quaisquer ações prévias necessárias para o correto funcionamento da tarefa a ser executada.

Durante todo o período de execução das tarefas, outro passo que é realizado é o monitoramento do progresso da execução. Com base neste monitoramento, ações preventivas ou corretivas podem ser tomadas, como por exemplo o reescalonamento de uma tarefa.

Finalmente, o passo seguinte do processo de execução de tarefas no Grid é o encerramento da mesma, quando o usuário é notificado do término, por vezes utilizando scripts e/ou e-mail. A este passo segue-se o passo final do processo que trata das tarefas de limpeza, recuperando arquivos daquele recurso, removendo configurações temporárias, entre outros.

2.2 Arquitetura do Grid

O principal objetivo de um Grid é o compartilhamento coordenado de recursos entre diversas organizações virtuais. Esses recursos podem estar espalhados por uma ampla área geográfica, possuir proprietários, plataformas, sistemas, linguagens de programação, políticas de acesso e segurança diferenciadas, entre outros.

Para viabilizar esta interoperabilidade definem-se regras que acabam por sua vez definindo a arquitetura. Geralmente, são definidos mecanismos de negociação e de interação e a estrutura da informação pela qual as informações são trocadas, além de estabelecidas e gerenciadas as relações de compartilhamento;

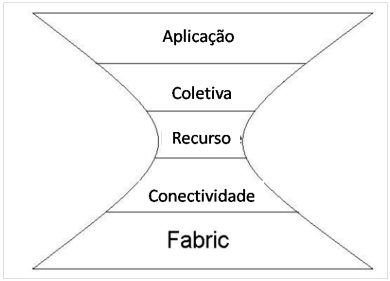

Figura 2.1: O modelo da ampulheta para a arquitetura do Grid [21]

Uma das primeiras arquiteturas propostas para Grid foi a arquitetura de protocolos baseada no modelo da ampulheta [21].

2.2.1 Modelo da Ampulheta

O modelo da ampulheta (ou arquitetura de protocolos) possui a configuração ilustrada na Figura 2.1.

A camada *Fabric* provê os componentes que serão compartilhados pelos usuários do Grid, tais como recursos computacionais, de armazenamento e recursos de rede. Os componentes desta camada são responsáveis pela implementação de operações específicas, dependentes de recursos específicos. Os componentes dessa camada podem ser dos seguintes tipos: computacionais, de armazenamento, de rede, de código ou catálogos.

A camada *Conectividade* é a camada na qual são definidos os protocolos de autenticação/segurança e de comunicação, necessários para operações de rede no Grid. Os protocolos de comunicação devem prover suporte a operações de transporte, operações de roteamento e de identificação de endereços através de nomes.

Quanto à autenticação/segurança, é importante que os mecanismos baseiem-se em padrões existentes e atuem de forma uniforme conforme as características de *Single sign-on* [21], delegação, integração com várias soluções de segurança locais e de relações de confiança entre usuários. Os mecanismos de segurança do Grid devem prover flexibilidade à proteção de comunicações e devem permitir que os interessados tenham controle sobre as decisões de autorização.

A camada *Recurso* define protocolos, APIs e SDKs para a negociação segura, inicialização, controle e monitoração das operações de compartilhamento de recursos individuais por meio de chamadas a funções definidas na camada *Fabric* para acessar e controlar recursos locais. Os protocolos desta camada se preocupam unicamente com os recursos individuais, ignorando o estado global do sistema distribuído. Podemos distinguir dentro da camada *Recurso* dois tipos fundamentais de protocolos: Protocolos de informação, que descrevem o estado e a estrutura de recursos, e os de gerência, que negociam o acesso a recursos compartilhados. A escassez de protocolos é consequência da restrição do "gargalo da ampulheta".

Diferente da camada *Recurso*, que se preocupa com as operações em recursos individuais, a camada *Coletiva* concerne a definição de protocolos, APIs e SDKs que estão ligados a recursos de natureza global, capturando interações entre conjuntos de recursos. Nesta camada, a restrição do gargalo da ampulheta não mais se aplica. Há, nesta camada,

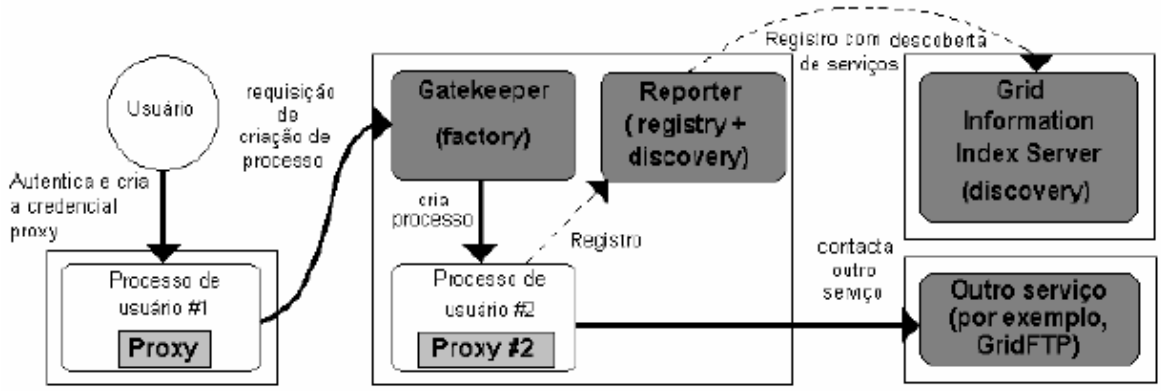

Figura 2.2: Mecanismos da camada Resource no GlobusToolkit 2 [20]

uma ampla variedade de protocolos e serviços que podem ser implementados, variando de genéricos a protocolos definidos por usuários.

Por fim, a camada *Aplicação* é o topo da arquitetura e inclui as aplicações de usuário, construídas de acordo com e utilizando os serviços das demais camadas.

Implementação - Globus Toolkit 2 A arquitetura de protocolos foi implementada pelo Globus Toolkit 2 (GT2), conforme consta em [19]. O GT2 é constituído de um conjunto de serviços e bibliotecas de arquitetura e código aberto que dá suporte ao Grid e a suas aplicações.

Todas as camadas descritas anteriormente da arquitetura da ampulheta são, em algum nível, suportadas pelo GT2. A camada *Fabric*, mais diretamente relacionada ao hardware, não é implementada diretamente pelo GT2, porém este fornece componentes projetados para facilitar a interconexão com protocolos da camada Recurso. Inclui mecanismos para a descoberta do estado e estrutura de recursos e para o empacotamento das informações descobertas de forma a facilitar a implementação de protocolos de níveis superiores [19].

A camada Conectividade é definida com base em uma estrutura de segurança baseada em chave pública chamada *Grid Security Infrastructure* (GSI), que provê meios para que aplicações invoquem operações de autenticação utilizando APIs de alto-nível, ou seja, sem realizar operações de protocolo diretamente. Esta estrutura garante o suporte à autenticação única (*single sign-on*), delegação e proteção às comunicações [19].

Em relação à camada Recurso, o GT2 utiliza o protocolo *Grid Resource Allocation and Management* (GRAM) para a criação segura e confiável de computações remotas [19]. Sua implementação do GRAM utiliza um processo *gatekeeper* para a criação de computações remotas, um gerente de tarefas para gerenciá-las e o GRAM reporter para a monitoração e publicação de informações sobre identidade e estado das computações locais.

O GT2 utiliza também uma implementação do *Monitoring and Discovery Service* (MDS-2), o qual provê meios para a descoberta e o acesso a informações de configuração e status de serviços [19]. Tal implementação provê dois componentes principais: um registro local, responsável por gerenciar a coleta e publicação de informações em um local em particular, e um registro coletivo, responsável por dar suporte a consultas de informações partindo de múltiplas localidades.

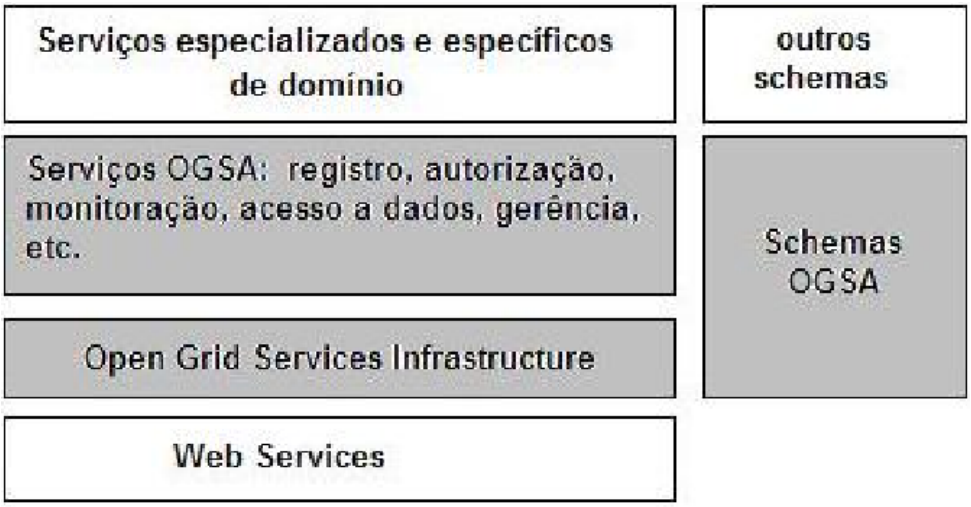

Figura 2.3: Arquitetura OGSA e seus principais componentes [19]

Na camada Recurso, como ilustra a Figura 2.2, inicialmente o usuário se autentica e é criada uma credencial para o processo de usuário #1, o qual requisita, utilizando a credencial criada e o protocolo GRAM, a criação de um processo remoto. A requisição é processada pelo gatekeeper do GRAM e um novo processo é criado, o processo de usuário #2, com novas credenciais. Propriedades relativas ao processo criado são registradas pelo MDS-2.

2.3 Open Grid Services Architecture (OGSA)

A OGSA é considerada uma evolução da arquitetura de protocolos e visa o suporte à criação, manutenção e aplicação dos serviços mantidos pelas organizações virtuais [20, 22]. É orientada a serviços, ou seja, todos os recursos disponíveis no Grid são representados como serviços, sendo estes definidos como entidades que provêem determinadas funcionalidades aos clientes através de trocas de mensagens. É uma arquitetura construída sobre *Web Services*, sendo independentes de modelo e de linguagem de programação, assim como de sistema de software. A OGSA e seus principais componentes podem ser visualizados na Figura 2.3.

A OGSA define o conceito fundamental de *Grid Service*, um *web service* que implementa interfaces e comportamentos padrão que permitem que os serviços sejam criados, destruídos e tenham estados bem definidos. Tais interfaces e comportamentos são definidos pela *Open Grid Services Infrastructure* (OGSI).

Logo acima da OGSI, tem-se na arquitetura um conjunto de serviços providos pela OGSA. Tais funcionalidades são combinadas de forma a se construir interfaces de mais alto nível para que se possa prover novas funcionalidades não suportadas diretamente pela OGSI [19]. Um dos componentes mais importantes para a definição de um *web service* na OGSA é a *Web Service Definition Language* (WSDL) [8]. É fundamental que no Grid haja suporte à descoberta dinâmica e composição de serviços em ambientes heterogêneos, o que necessita mecanismos para o registro e descoberta de definições de interfaces, ao que a WSDL dá suporte separando as definições de sua implementação concreta [19]. Uma

definição de serviço WSDL é um documento XML que descreve interfaces de serviço para acesso a Web services, contendo também as mensagens trocadas no acesso ao serviço, dentre outras informações.

A OGSA não possui como objetivo apresentar detalhes de implementação, tais como linguagem de programação, ferramentas de implementação ou ambiente de execução. Isso torna possível que sejam especificadas interações entre serviços de maneira que sejam totalmente independentes do ambiente em que os serviços estão hospedados.

A camada mais baixa da OGSA (OGSI) sofria de incompatibilidade parcial com muitas ferramentas de manipulação de Web Services. Por essa razão, a mesma foi substituída pela WSRF [11], descrita na Seção 2.4, gerando a arquitetura OGSA+WSRF.

2.4 WS-Resource Framework (WSRF)

O *WS-Resource Framework* (WSRF) é tido também como uma evolução da OGSI que possui como objetivos principais a criação, endereçamento, inspeção e gerência de recursos com estados bem definidos [11]. Codifica a relação entre Web services e os recursos utilizando metadados como o endereço do Web service e outros associados a este como a descrição e propriedades do serviço.

A composição entre um recurso e um Web service é denominada WS-Resource. O WSRF permite a declaração, criação, acesso, monitoramento e destruição de WS-Resources através de mecanismos convencionais de Web Services, e é descrito por cinco especificações normativas [12]. São elas:

- **WS-ResourceLifetime**: Consiste em mecanismos para a destruição de WS- Resources, incluindo trocas de mensagens que permitem que um requerente execute a destruição tanto de forma imediata quanto de forma agendada;
- **WS-ResourceProperties**: Consiste na definição de um WS-Resource, além de mecanismos para a recuperação, mudança e remoção de propriedades;
- **WS-RenewableReferences**: Consiste em informações associadas a uma referência *endpoint* contendo dados necessários para a recuperação de uma nova versão da referência caso ela se torne inválida;
- **WS-ServiceGroup**: Consiste em uma interface para coleções heterogêneas de *Web Services* e;
- **WS-BaseFaults**: Um tipo XML básico para ser utilizado quando for necessário aos *Web Services* retornar falhas em uma troca de mensagens.

2.5 Globus Toolkit 4

O Globus Toolkit 4 (GT4) é uma implementação da arquiteura OGSA+WSRF. Faz uso extensivo de Web services para a definição de suas interfaces e para a estruturação de seus componentes [18]. É composto por um conjunto de serviços, conteineres para serviços desenvolvidos por usuários (suportando Java, C e Python) além de bibliotecas para chamadas de métodos do GT4 e de serviços de usuário.

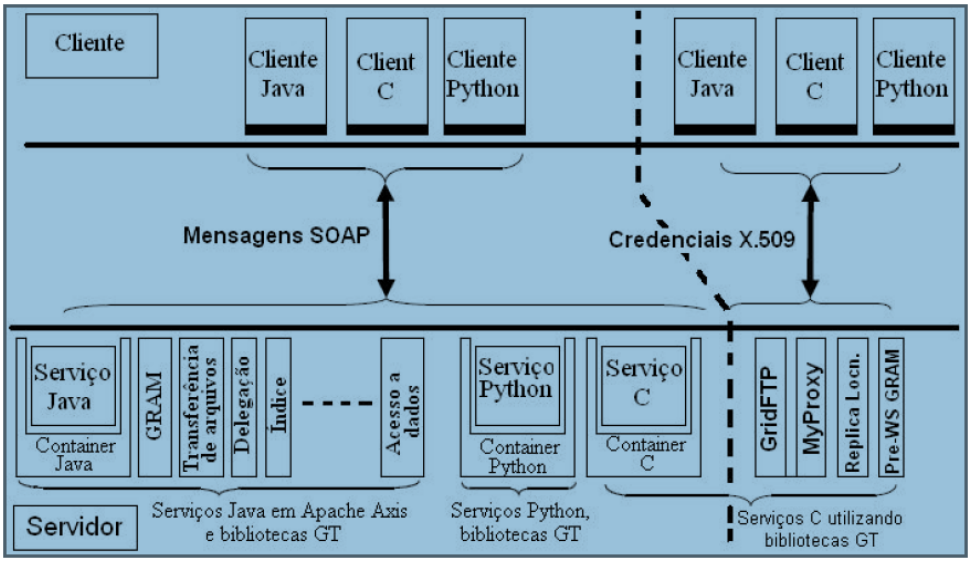

Figura 2.4: Arquitetura do GT4 [18]

O conjunto de serviços contidos no GT4 auxiliam na infraestrutura [18], gerência de execução, acesso e movimentação de dados, gerência de credenciais, monitoração e descoberta de recursos e serviços, entre outros. Para a gerência de execução, o GT4 possui uma implementação do *Globus Resource Allocation and Management* (GRAM), o qual fornece uma interface Web services para a iniciação, monitoração e a gerência de computações arbitrárias em recursos remotos, e que permite a um cliente expressar, entre outras coisas, a quantidade de recursos desejada, o executável e seus parâmetros. Para a descoberta e monitoração de recursos e serviços, provê mecanismos padronizados para a associação de arquivos de propriedades baseados em XML a entidades da rede e para o acesso a tais propriedades. Provê ainda serviços para a coleta e acesso de informações recentes sobre o estado de diversos serviços registrados. A arquitetura do GT4 é ilustrada na Figura 2.4.

2.6 Integrade

O Integrade foi proposto em 2004 [24]. É um projeto de várias universidades que visa construir um *middleware* de Grid orientado a objetos para disponibilizar o poder de computação não utilizado por estações de trabalho aos demais usuários do Grid.

Seu objetivo principal é permitir que organizações utilizem a infraestrutura de computação já existente para executar tarefas complexas, dispensando-as da necessidade de adquirir *hardware* adicional. Além disso, os usuários que compartilham seus recursos devem ter seu nível de qualidade de serviço preservados pelo *middleware*.

O Integrade estrutura o Grid em *clusters*, possuindo de 1 a aproximadamente 100 máquinas cada um, que podem ser máquinas compartilhadas ou servidores dedicados [24]. Os *clusters* fazem parte de uma hierarquia de forma a permitir que um único Grid Integrade englobe um número muito grande de máquinas. Um exemplo desta hierarquia de *clusters* é ilustrado na Figura 2.5.

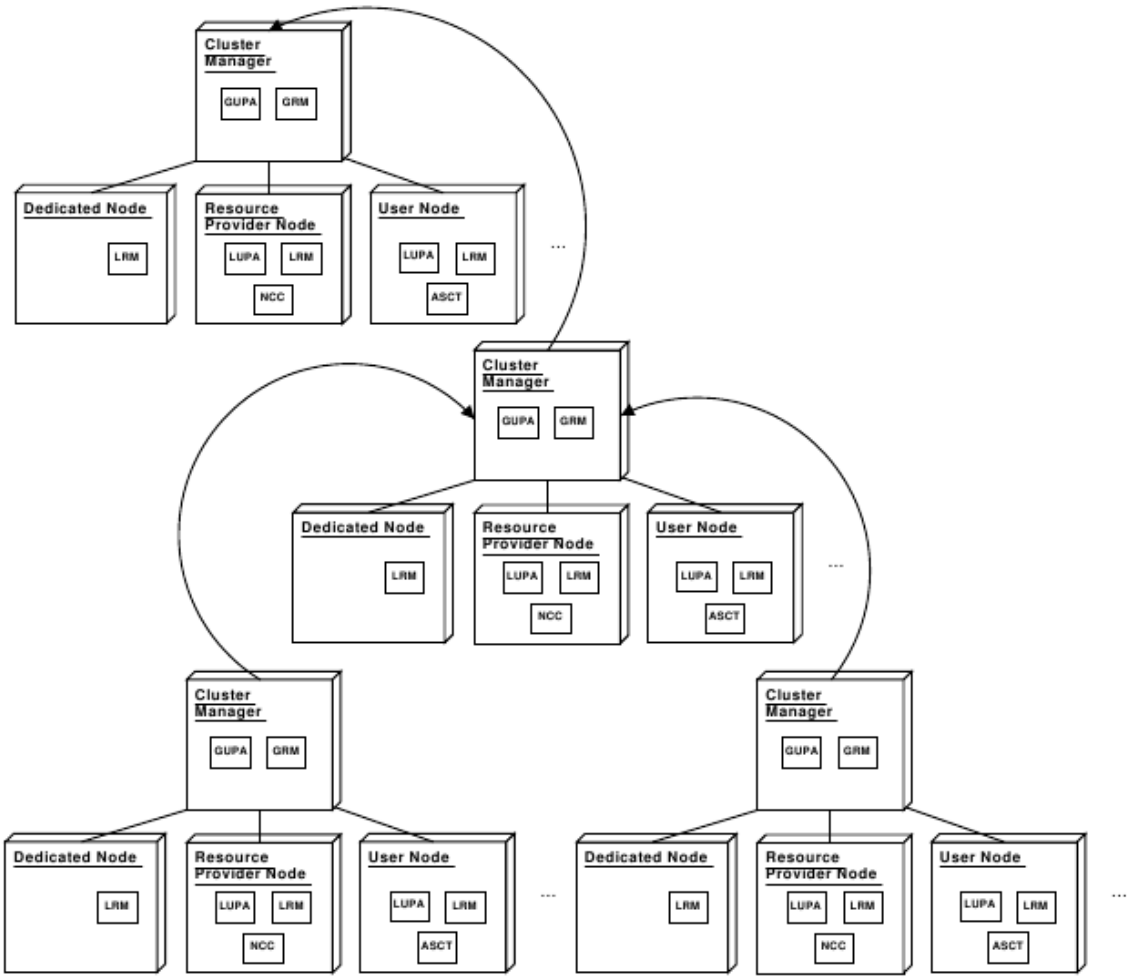

Figura 2.5: Hierarquia de Clusters do Integrade [24]

A arquitetura atual do Integrade está ilustrada na Figura 2.6 [35, 43] e consta de diversos módulos. Estes módulos podem ser divididos conforme o nodo que o executa. A arquitetura do Integrade é executada em dois contextos distintos: o gerenciador de recursos do Grid e os provedores de recursos.

No nodo do Gerenciador de Recursos do Grid (*Grid Resource Manager* - GRM), existem os seguintes módulos:

AR `Application Repository`: Um repositório para as aplicações, dados e meta-dados do Grid.

ARSM `Application Repository Security Manager`: responsável pelo gerenciamento de segurança no Grid *cluster*, fornecendo suporte para assinaturas digitais, criptografia de dados, autenticação e autorizações.

CDRM `Cluster Data Repository Manager`: Gerencia os repositórios de checkpoint de seu *cluster*. Além disso, distribui fragmentos codificados de *checkpoints* entre os demais repositórios de *checkpoint* disponíveis.

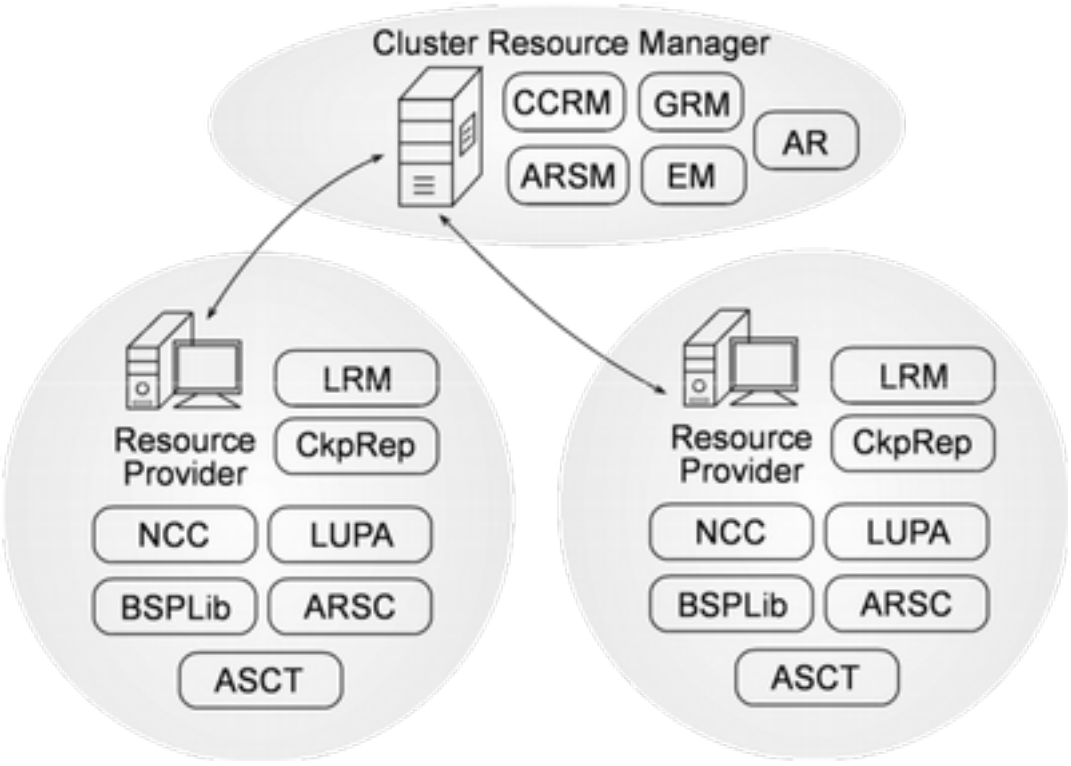

Figura 2.6: A arquitetura atual do Intergrade - [35]

EM `Execution Manager`: Mantém os dados de execução, como *checkpoints* salvos e nós onde a aplicação está rodando. Também coordena a reinicialização e migração de aplicações.

GRM `Global Resource Manager`: Gerencia os recursos computacionais em um *cluster* de máquinas conectados por uma rede local (LAN) contendo tipicamente de uma a cem máquinas. Provê os mecanismos de escalonamento baseados no conhecimento adquirido sobre a disponibilidade dinâmica de recursos e os requisitos da aplicação.

Já no Provedor de Recursos (*Resource Provider*), existem os seguintes módulos:

ARSC `Application Repository Security Client`: Auxilia os componentes locais do Integrade a interagir com componentes remotos de forma segura.

ASCT `Application Submission and Control Tool`: Interface de usuário para submissão e controle da execução de aplicações no Grid.

BSPLib `BSP Library`: Biblioteca implementando a API da Oxford BSPLib, que permite a execução de aplicações BSP no Integrade.

CkpRep `Checkpoint Repository`: Armazena dados de *checkpoint* de forma distribuída. Cada provedor de recursos que rode um LRM é um nó potencial para hospedar um repositório de checkpoints.

LRM `Local Resource Manager`: gerencia os recursos em uma máquina específica e roda aplicações submetidas pela ASCT.

LUPA `Local Usage Pattern Analyser`: Obtém informações sobre padrões de uso de recursos em uma máquina específica e procura fazer previsões sobre a utilização futura dos recursos baseado em técnicas de aprendizado de máquina (Este componente ainda está em desenvolvimento)

NCC `Node Control Center`: Interface de usuário que define o grau de recursos locais que estarão disponíveis para utilização em aplicações de Grid (Este componente ainda está em desenvolvimento).

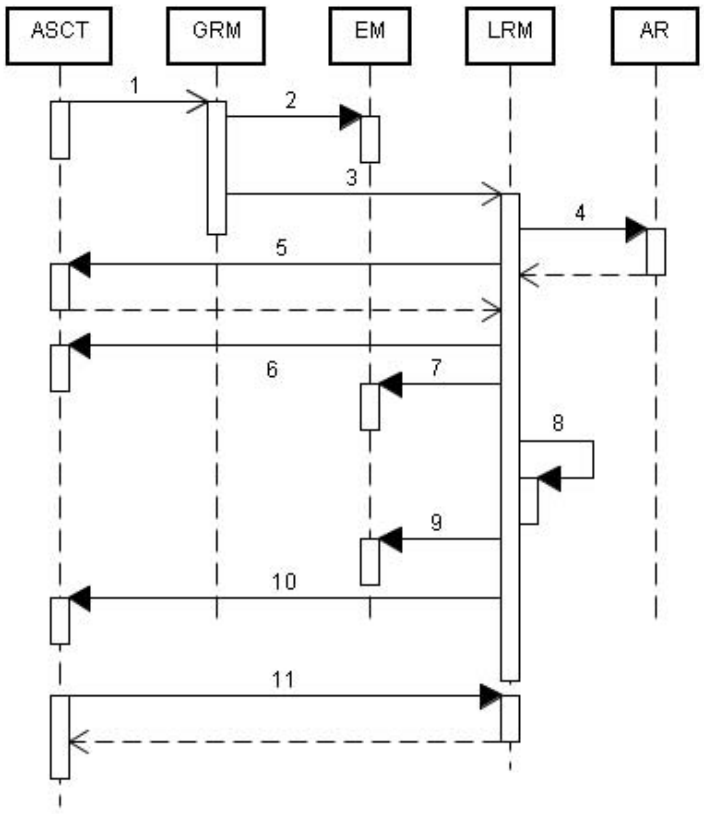

Figura 2.7: Interações entre módulos do Integrade [13]

O relacionamento entre módulos pode ser ilustrado pela Figura 2.7. Como representado na imagem, uma tarefa que deva ser executada pelo Grid é primeiramente submetida pelo usuário por meio do ASCT, repassada ao GRM (1) que decidirá em que cluster ela deverá ser executada por meio de seus algoritmos de escalonamento. O GRM notifica então o EM (2) que a aplicação foi agendada e encaminha ao LRM (3) os dados de submissão do nó selecionado. O LRM obtém os binários da aplicação do AR (4), requisita, se necessário, os arquivos de entrada para o ASCT (5), enquanto concorda com a solicitação (6). Logo antes de iniciar a execução, o LRM notifica o EM da inicialização do aplicativo (7) e inicia o monitoramento do progresso da execução (8), enquanto aguarda sua conclusão. Neste ponto, o LRM notifica o EM (9) e o ASCT (10) o término da aplicação. Finalmente, o ASCT pode baixar os resultados (arquivos de saída) da aplicação executada (11) [13].

Capítulo 3

Tolerância a falhas

3.1 Conceitos Básicos

Um sistema tolerante a falhas é aquele cujo correto funcionamento não é afetado pela ocorrência de falhas em seus componentes [41]. Uma falha se caracteriza como a discrepância entre o comportamento esperado e o comportamento observado do sistema. Uma falha (consequência do problema) é decorrente de um erro (manifestação do problema), que por sua vez é causado por uma falta (causa do problema) no sistema [39].

Na classificação dada por [46], falhas podem ser *Transientes*, se ocorrem apenas uma vez; *Intermitentes*, se ocorrem esporadicamente ou; *Permanentes*, se indisponibilizam todo o sistema até que sejam sanadas.

Algumas métricas importantes de tolerância a falhas são o tempo para reparo da falha, que representa a manutenibilidade e o tempo médio entre as falhas, representando a disponibilidade e confiabilidade. Outros conceitos importantes são [46]:

- *Safety*: Propriedade de um sistema na qual todas as possíveis computações resultem em uma configuração do escopo de configurações válidas, ao que se denomina `invariante`.
- *Liveness*: Propriedade de um sistema no qual o resultado ou comportamento desejado seja certamente percebido em algum momento futuro.

Conforme [46], para que um sistema distribuído comporte-se adequadamente ele deve satisfazer as condições de *safety* e de *liveness*. No entanto, isto não necessariamente se manterá em caso de falhas.

Suponha que um programa A consiga manter ambas propriedades, ainda que falhas de uma certa classe F ocorram. Neste caso diz-se que A está mascarando as falhas da classe F . Esta é a abordagem mais estrita e mais almejada, porém também a mais custosa. Se por outro lado, um programa B não garantir nenhuma das propriedades, então não há como se falar em tolerância a falhas. Fora dos dois extremos há ainda duas possibilidades. Se apenas a propriedade de *Safety* puder ser garantida, o programa é chamado de *fail-safe*. Finalmente, caso o programa seja elaborado de forma a continuar (*liveness*) mesmo que falhas de uma dada classe o levem a um estado onde a propriedade de *safety* não possa ser garantida, ele será denominado como *nonmasking* [23].

3.2 Modelo de Falhas

As duas categorizações básicas para a classificação de falhas são a origem ou o tipo de falha [27]. Na primeira classificação, as falhas podem se originar de falhas físicas no hardware oriundo de fatores externos ou internos (radiação, temperatura, desgaste, curto-circuito). Podem ser originadas também de falhas no projeto, decorrente de equívoco, geralmente humano, na concepção e implementação do sistema. Também podem ser originadas em falhas de interação, ou seja problemas na interface entre componentes ou no middleware. Já na classificação por tipo existem falhas:

- Por Omissão: O sistema é incapaz de atender às demandas e não produz saída para alguns valores de entrada. Também é conhecida como *fail-stop*
- Por Desempenho: O sistema produz a resposta adequada porém fora do intervalo de tempo previsto. Uma falha por omissão pode ser considerada também uma falha por desempenho.
- Por Valor (ou resposta): O sistema produz uma resposta incorreta para uma ou mais requisições. Podem ser subdividas em falhas de valor e de transição de estados.
- Arbitrárias (ou bizantinas): Ocorrem quando um componente do sistema deliberadamente omite ou acrescenta etapas ao processamento e em geral não podem ser detectados trivialmente pela existência ou não de uma resposta à requisição.

3.3 Redundância

A redundância de um sistema é a técnica mais utilizada para tolerância a falhas e pode ser classificada de forma geral como [46]:

- Redundância de Informação ou Valor: Permite verificar a integridade de dados.
- Redundância de Tempo: Possibilita que uma mesma tarefa seja executada repetidas vezes até que se alcance o resultado esperado.
- Redundância Física ou de Espaço: Adicionam-se componentes além dos necessários de forma a permitir que a falha de até um certo conjunto de elementos não afete o resultado esperado. Para isto pode-se utilizar a replicação ativa ou *backup* primário.

3.3.1 Replicação ativa

Na replicação ativa, um dado componente de software é processado por distintas réplicas, as quais operam conforme uma máquina de estados. Partindo do pressuposto de que as entradas de todas as réplicas são idênticas (Consistência da entrada), com processamento determinístico e passando pelas mesmas transições de estados (Determinismo do grupo de réplicas) então todas as saídas serão iguais, exceto possivelmente daquelas que enfrentaram algum erro durante o processamento. Para eliminá-las, as saídas são repassadas a um elemento votante que define por maioria o resultado mais provável. Para mascarar também os erros dos elementos votantes, cada componente possui um elemento votante distinto (redundância física). Esta metodologia permite inclusive mascarar falhas bizantinas de

até f dentre um total de *2f+1* processadores [44]. Por outro lado, se apenas falhas do tipo *fail-stop* são consideradas, os elementos votantes tornam-se dispensáveis pois a primeira saída já será considerada como a saída correta. Desta forma consegue-se mascarar falhas em até f de um total de *f+1* componentes. Porém, além das premissas de Consistência da entrada e do Determinismo do grupo de réplicas é necessária também a Coordenação de réplicas que significa que todas as réplicas recebem todas as requisições de seus clientes (*Acordo ou consenso*) e que as processam exatamente na mesma ordem (*Ordenação*) [44]. Todos estes pré-requisitos determinam a consistência dentro do grupo de réplicas e devem ser garantidos por meio de protocolos específicos.

3.3.2 Backup primário ou Replicação Passiva

No modelo de replicação passiva existem duas classes de réplicas: uma réplica denominada primária, responsável pelo processamento propriamente dito e um conjunto de uma ou mais réplicas denominadas "*backups*" ou "*slaves*", que assumem o papel de réplica primária em caso de falha da mesma, garantindo a continuidade.

A réplica primária é a única a receber, processar e responder às requisições dos clientes. As demais permanecem inertes exceto por atualizar seus estados internos sempre que a réplica primária emitir um checkpoint, o que ocorre em intervalos regulares. Estes checkpoints permitem às réplicas reiniciar o processamento já realizado pela réplica primária até o envio do checkpoint em si, ou seja, até o último estado certamente consistente do sistema em caso de falha da réplica primária.

Este sistema parte do pressuposto de que todas as saídas geradas por uma réplica são corretas e portanto não é capaz de lidar com falhas bizantinas, apenas com falhas do tipo *fail-stop* até o limite de f falhas para um conjunto de $f + 1$ réplicas.

Como principais vantagens em relação à replicação ativa, o modelo de backup primário apresenta um esquema de funcionamento mais simples devido à centralização do processamento em uma réplica e não em todo um grupo evitando por exemplo o problema da ordenação de mensagens e o menor número de componentes necessários, apenas um servidor primário e um servidor de backup. Por outro lado, a impossibilidade de tratar falhas bizantinas e a possível complexidade e o tempo necessário para transformar uma réplica de backup em uma réplica primária são pontos que prejudicam esta abordagem.

3.4 Fases do processamento de erros

Em um sistema tolerante a falhas, definem-se 3 fases para o processamento de erros, se possível antes mesmo que estes incorram em falhas. Estas fases são [29]:

1. Detecção, compensação e correção do erro;
2. Confinamento e Avaliação do Dano e;
3. Recuperação do sistema.

3.4.1 Detecção, compensação e correção do erro

Existem basicamente dois tipos de erros que podem ser detectados: erros no domínio do tempo e erros no domínio do valor. Por sua vez, estes erros podem se dar tanto em

hardware quanto em software. Assim, foram desenvolvidos mecanismos específicos para lidar com tais erros, como é explicado a seguir.

Mecanismos para tolerância a falhas em Hardware Para a detecção de erros em hardware do domínio de valor existem mecanismos baseados na teoria da codificação, onde se acrescentam alguns bits de forma a gerar redundância na informação permitindo detectar e/ou corrigir erros. Para isso, podem ser utilizados Códigos de Hamming [27], bits de paridade, códigos de redundância cíclica, "*m-out-of-n*" ou aritméticos. No entanto, estes sistemas só conseguem identificar erros na manutenção da informação durante sua transmissão ou armazenamento.

Para identificar erros na transformação da informação é necessário algum grau de redundância física com vários componentes realizando a mesma tarefa de forma a compararem-se os resultados finais. Neste âmbito a técnica mais comum é a Redundância Modular Tripla, ou TMR [27], onde o processador é triplicado e o resultado é avaliado por um elemento votante. Esta técnica é suficiente para mascarar erros em um processador.

Para a detecção de erros no domínio do tempo, utilizam-se temporizadores ou *timeouts* [36] que definem o tempo máximo aceitável para responder a uma requisição. Caso a resposta não ocorra dentro daquele intervalo de tempo, o sistema sinaliza uma falha.

Mecanismos para tolerância a falhas em Software Os mecanismos para tolerância a falhas em software são em muitos aspectos similares aos de hardware. Os erros no domínio do tempo também são identificados por meio de *timeouts*. Contudo, um *timeout* não significa necessariamente um erro pois pode ser decorrente de sobrecarga do componente. Um sistema de administração registra os *timeouts* de maneira a buscar aliviar a carga do componente antes de declará-lo definitavemente como indisponível por repetidas notificações de erro. Outro complicador neste aspecto é que muitos sistemas distribuídos são assíncronos de forma a ser impossível definir um tempo limite para respostas [36].

Já os erros no domínio de valor são identificados por meio de diferentes réplicas de um componente de software. Para isto, elas devem permanecer consistentes, ou seja, deve haver algum método para garantir a equivalência das mensagens. Deverá haver também um esquema de votação majoritária para escolher, dentre os valores gerados pelas diferentes réplicas, aquele que será considerado correto.

3.4.2 Confinamento e Avaliação do Erro

Para evitar-se que um erro seja propagado devem existir mecanismos que o confinem ao ponto onde ocorreu. Neste sentido, existem duas abordagens:

- os componentes verificam suas entrada ou;
- os componentens verificam suas saídas.

Para a verificação dos dados na entrada deve haver algum módulo de detecção/correção de erros anterior ao componente, cuja função seja checar e permitir a entrada apenas de informações corretas.

Já na segunda abordagem existe um módulo de controle de erros na interface de saída que, ao encontrar um erro, é capaz de corrigi-lo em tempo hábil ou desabilitar a saída impedindo portanto uma propagação do erro para outros componentes.

Uma terceira opção, voltada para sistemas sem recursos especiais para o confinamento do erro baseia-se na assertiva de que o erro se propaga por meio da comunicação entre os módulos e que, uma vez que se controlem as comunicações, pode-se avaliar a extensão dos danos após a identificação de um erro pelo exame das mesmas.

3.4.3 Recuperação do sistema

Apesar do confinamento de erro preservar os demais componentes ele não é suficiente para mascarar e reparar o erro em si. Para tal, existem técnicas que retornam o sistema a um estado válido, o que é conhecido como recuperação de erro [27].

Uma maneira comum de se realizar a recuperação de erro é através dos checkpoints. O processo é retrocedido (*rollback*) até o último estado válido (*checkpoint* mais recente) por meio da restauração dos valores dos registradores e da memória para então reiniciar a execução a partir daquele ponto por processadores suplentes (*spares*). Este mecanismo é conhecido como recuperação de erro para trás ou *backward error recovery* [48].

Outra técnica existente é a recuperação de erro para frente, ou *forward error recovery*, [48] que não requer conhecimento de estados prévios nem retrocede o sistema mas livra o estado atual de erros por meio de ações corretivas. A maior dificuldade deste método é a necessidade de se conhecer as ações corretivas necessárias para cada erro em particular, o que a torna sensivelmente menos geral que a abordagem anterior [27].

3.5 Fases relacionadas ao tratamento da falta

Até agora, tratou-se do processamento de erros que ocorreram e foram de alguma forma mascarados ou confinados. Porém existem abordagens que lidam com a falha em si, antes que erros se manifestem ou de forma a prevenir que eles se repitam, estas se denominam tratamento da falta.

Faltas temporárias podem ser resolvidas por uma nova tentativa ou reinicialização do sistema, uma vez que ela não estará mais presente. Isto não ocorre em faltas permanentes, que precisam ser retiradas do sistema.

Para o tratamento de faltas permanentes, as seguintes fases são observadas: diagnóstico, apassivação e reconfiguração [39].

O diagnóstico visa identificar os componentes que apresentam falhas e causaram os erros. A partir desta listagem, a apassivação procura evitar que estes erros se repitam e para isso ela utiliza-se de reconfiguração para que os componentes identificados sejam excluídos do sistema, o que pode incorrer em uma menor capacidade de tolerância a falhas. Para corrigir tal deficiência, a reconfiguração trata de restaurar o nível de redundância do sistema de forma que futuros erros possam ser também contornados.

Capítulo 4

Workflows

Workflows científicos são redes de passos analíticos que representam a computação necessária à análise de grandes volumes de dados [34]. Eles podem envolver acessos a bases de dados, análises e mineração dos mesmos e muitos outros passos inclusive tarefas com grande esforço computacional em *clusters* de computadores de alto desempenho [32]. Os *workflows* científicos podem ser inclusive de grande escala não apenas em termos computacionais como também em número de tarefas, tempo total de execução entre outros [34].

Em geral, *workflows* possuem características que lhe são peculiares. Eles podem ser *data-intensive*, *compute-intensive*, *analysis-intensive*, *visualization-intensive*, entre outros. De acordo com o grupo de usuários almejado, alguns aspectos ou peculiaridades técnicas são ressaltados ou suprimidos. Isto é realizado definindo-se vários níveis de complexidade para utilização de acordo com o perfil do usuário.

Um exemplo de *workflow* científico é apresentado na Figura 4.1. Nele estão ilustrados os passos necessários para a identificação e caracterização de promotores eucarióticos na bioinformática [32]. A partir de dados de *micro-arrays* utilizam-se algoritmos de análise para levantar genes com padrões de perfil de expressão genética similares. Com isso deduz-se serem co-regulados como parte de um caminho bioquímico. Dadas as identificações dos genes, as sequências são buscadas em um banco de dados e repassadas à uma ferramenta que localiza sequências similares. Nos passos subsequentes, o *binding* dos fatores de transcrição e promotores são identificados para criar um modelo de promotor que possa ser refinado iterativamente [32].

4.1 Requisitos de *Workflows* Científicos

Os *workflows* científicos possuem uma série de requisitos. Alguns dos mais comuns estão listados abaixo [32]:

1. **Acesso transparente a recursos e serviços:** Os acessos a recursos e serviços utilizados devem ser independentes da implementação interna dos mesmos. Isto é fornecido pelos Web Services.
2. **Design de *workflows* por meio de composição de serviços e reuso:** Em um *workflow* o usuário deve ser capaz de agrupar vários serviços de baixo nível em um único serviço de mais alto nível.

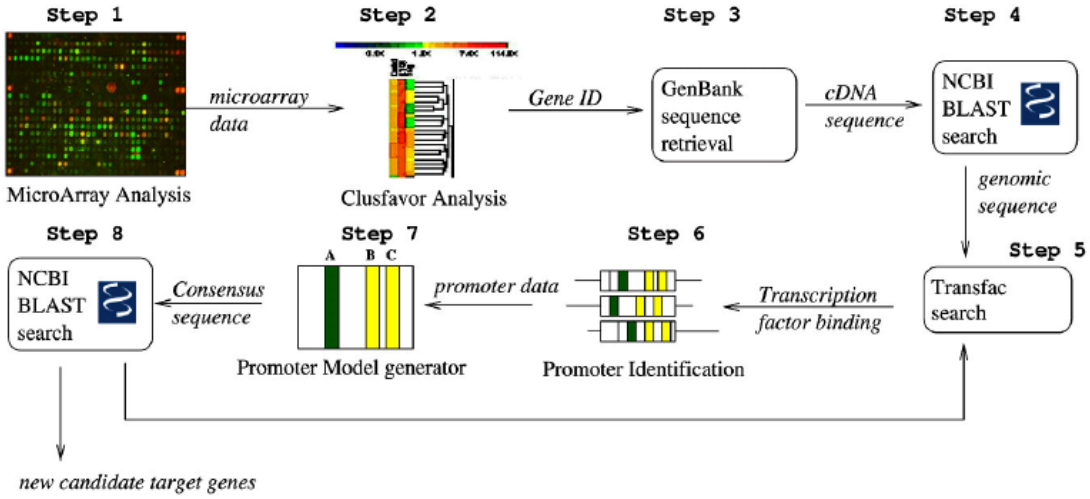

Figura 4.1: *Workflow* para identificação de promotores (PIW) - Alto Nível [32]

3. **Escalabilidade:** Para suportar o aumento escalável do volume de dados e computação necessários para alguns *workflows*, são necessárias interfaces adequadas para os componentes do *middleware*.
4. **Execução destacada:** *workflows* devem permitir que o mecanismo de controle rode em segundo plano em uma máquina remota sem a necesidade de conexão constante à aplicação do usuário.
5. **Confiabilidade e tolerância a falhas:** O *workflow* deve permitir que planos de contingência possam ser especificados para o caso de erros.
6. **Interação com usuário:** Alguns *workflows* podem permitir que o usuário pause o processo e inspecione resultados intermediários.
7. ***Re-runs* inteligentes:** Caso um usuário resolva rodar novamente o processo apenas com algumas diferenças de parâmetros, o *Workflow* não deverá reiniciar mas sim refazer apenas os passos alterados.
8. **Provenança de dados:** Um sistema de *workflows* científicos deve gerar um log dos passos realizados garantindo a possibilidade de reprodução dos experimentos e resultados obtidos.

Os *workflows* científicos não devem ser confundidos com *workflows* de Negócios [32]. Apesar da aparente similaridade, *workflows* de negócio tendem a possuir seu foco no fluxo de controle e em eventos enquanto os *workflows* científicos concentram-se no fluxo de dados. Esta diferenciação reflete inclusive nos formalismos que os suportam. Outra peculiaridade a ser ressaltada é o relacionamento próximo entre as abordagens baseadas em fluxo de dados e linguagens funcionais, inclusive variantes não estritas como Haskell [32].

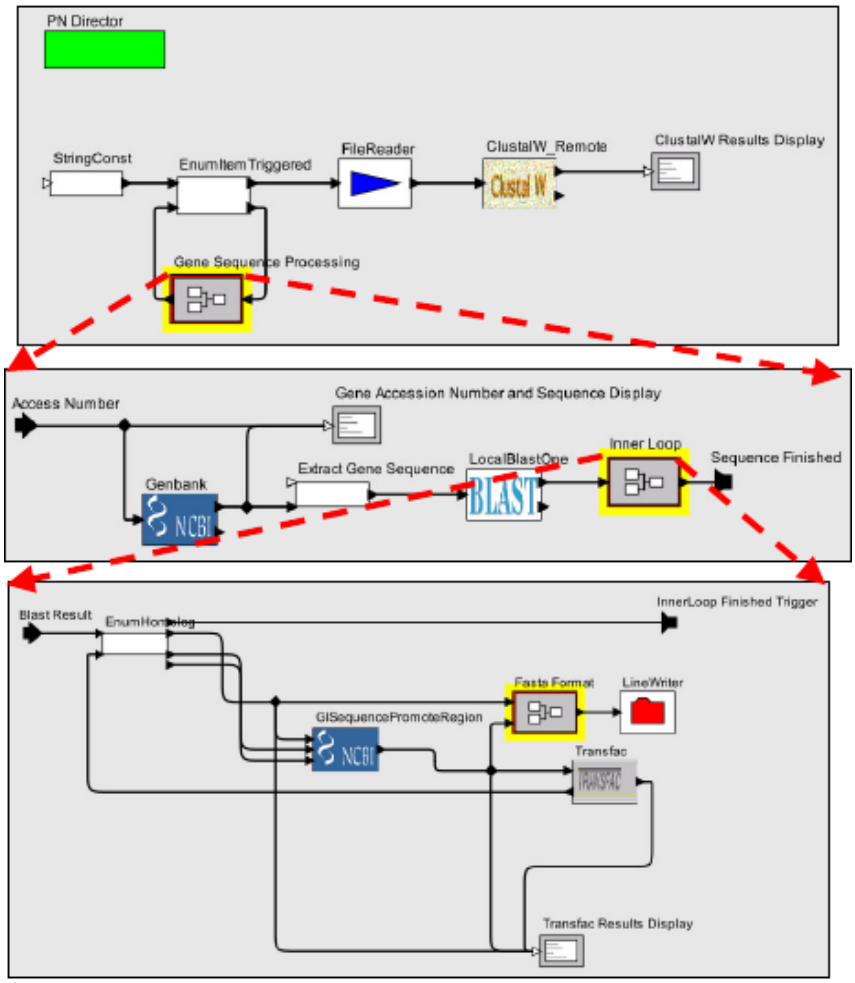

Figura 4.2: *Workflow* PIW - Detalhamento das tarefas [32]

4.2 Plano de Execução de *Workflows*

Para a definição de *workflows* existem serviços como os geradores de *workflow* ou simplesmente planejadores. Eles constroem os *workflows* que serão posteriormente realizados pelos executores, outra classe de serviços que realiza um *workflow* em um conjunto de recursos de um Grid [34]. Esta interação é crucial para a obtenção dos resultados esperados da execução.

Para a criação dos *workflows*, os planejadores devem decidir quando e onde executar cada tarefa do *workflow*. Para embasar a decisão, podem levar em consideração algumas métricas, por exemplo o tempo total de execução ou a probabilidade de sucesso na execução de uma tarefa. Decisões tomadas pelos planejadores são consideradas globais. Por outro lado, os planejadores podem deixar brechas no planejamento para que o executor tome decisões *just-in-time*. Essas decisões são denominadas locais.

Considere, por exemplo, o *workflow* PIW descrito anteriormente. O *workflow* ilustrado na Figura 4.2 descreve o mesmo fluxo porém já com o detalhamento das tarefas envolvidas. Para a sua correta execução são necessários tanto o planejador quanto os executores. O planejador, tendo a visão macro, pode tomar algumas decisões importantes como priorização de tarefas com maior criticidade fornecendo-lhes os recursos mais eficientes. Por

outro lado, aos executores cabe selecionar dentro das opções a ele fornecidas pelo planejador as melhores escolhas para a execução da tarefa específica a ele delegada. No intuito de evitar conflitos de interesses e de priorizar as tarefas mais críticas gera-se um plano de execução.

Para realizar a criação deste plano de execução existem várias abordagens. Algumas delas são [34]:

full-plan-ahead O planejador cria um *workflow* completo e detalhado baseado nas informações disponíveis no momento do planejamento para posteriormente iniciar a execução.

in-time local scheduling O planejador deixa muitas escolhas em aberto e permite que os executores tomem tais decisões sem uma visão geral do *workflow* porém baseando-se em dados possivelmente mais atualizados.

in-time global scheduling O planejador cria um *workflow* simples e o fornece ao executor que lhe informa quando estiver pronto para agendar uma tarefa. Neste momento o planejador toma uma decisão utilizando informações globais.

Por suas peculiaridades, nenhuma das soluções é ideal em qualquer caso. Uma abordagem *full-plan-ahead* seria interessante em um ambiente de alta confiabilidade mas estaria propensa a muitos erros em um ambiente de rápida transformação. Uma abordagem *in-time local scheduling* pode optar por escolhas erradas por não levar em consideração todo o conjunto do *workflow*. E uma solução *in-time global scheduling* pode levar a um controle demasiadamente minucioso resultando em sobrecargas de comunicação e computação em razão do replanejamento [34].

Capítulo 5

Execução Tolerante a Falhas de *Workflows* em Grid

Para a execução tolerante a falhas de *workflows* em grid, várias abordagens já foram propostas, algumas das quais são avaliadas neste capítulo. As diversas abordagens existentes foram classificadas por Yu e Buyya [50]. Os elementos de tolerância a falhas de um sistema de *workflow* em Grid são classificados conforme a taxonomia descrita na Figura 5.1.

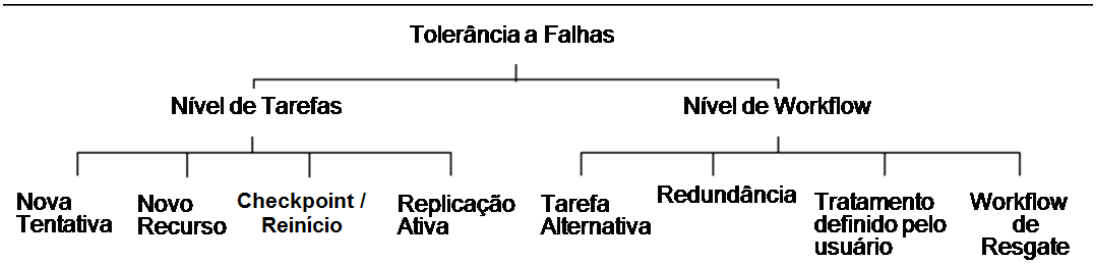

Figura 5.1: Taxonomia de tolerância a falhas [50]

Basicamente, dividem-se os elementos de tolerância a falhas em dois nichos, seguindo o que foi feito por Hwang e Kesselman [26]: tolerância a falhas em nível de tarefas e em nível de *workflow*.

Para mascarar os efeitos de falha na execução de tarefas no *workflow*, os mecanismos existentes são: realizar uma nova tentativa no mesmo ou em outro recurso, reiniciar a execução a partir do último ponto de consistência do sistema (*checkpoint*) ou do ínicio ou, finalmente, utilizar a replicação ativa.

Já as técnicas para tolerância a falhas de *workflow* visam manipular a estrutura do *workflow* para lidar com condições errôneas. Nesse sentido, após uma falha pode-se tentar uma tarefa ou *workflow* alternativo porém com função equivalente, executar o mesmo *workflow* em vários recursos ou definir um tratamento específico para uma certa falha no *workflow*. Por fim, a abordagem de *Rescue Workflow*, atualmente implementada pelo sistema Condor DAGMan [45], continua a execução das tarefas sem erros enquanto for possível e gera um relatório para posterior resubmissão [50].

A utilização de métodos de tolerância a falhas leva o sistema como um todo a uma eficiência e eficácia superior. Muitos dos *workflows* executados no Laboratório Nacional de Los Alamos (LANL), por exemplo, possuem prazos bem definidos e precisam ser executa-

dos com máxima probabilidade de sucesso [28], como por exemplo o *Linked Environments for Atmospheric Discovery* (LEAD) [28], cuja função de previsão do tempo possui limites muito restritos.

Após a implementação de um Serviço de Tolerância a Falhas e Recuperação, o laboratório LANL obteve ganhos muito significativos na execução completa e bem-sucedida de aplicações e de *workflows* rodando em um TeraGrid [42], representados nas Figuras 5.2 e 5.3 respectivamente.

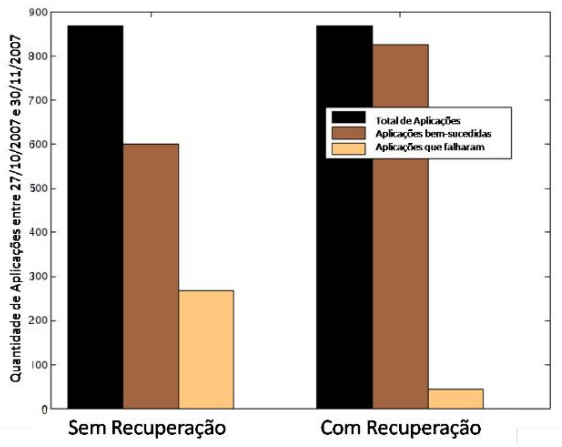

Figura 5.2: Estatísticas de recuperação de aplicações em *workflows* LEAD executando no TeraGrid [28]

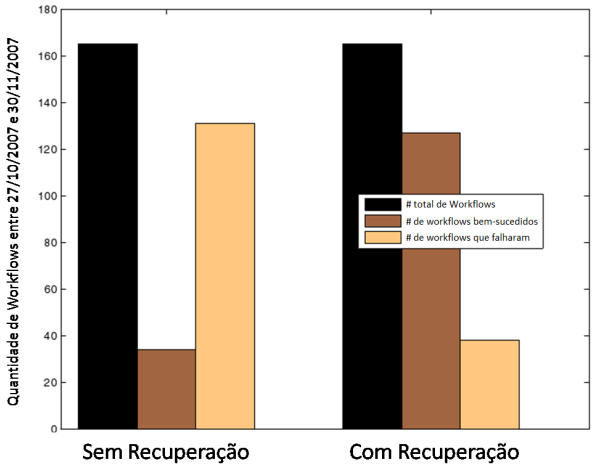

Figura 5.3: Estatísticas de recuperação de *workflows* LEAD executando no TeraGrid [28]

Apesar dos benefícios descritos, o uso de mecanismos de tolerância a falhas ainda não foi devidamente incorporado aos sistemas de Grid. Em [38], foi realizado um estudo comparando 10 sistemas distintos de *workflow* em Grid e os mecanismos implementados por cada um deles.

Neste estudo, ele mostra que os erros em níveis de abstração mais tradicionais (hardware, *Middleware* e *workflow*) são em geral detectados (Figura 5.4) porém a sua recuperação ainda é complicada considerando-se que menos de 40% dos erros são passíveis de

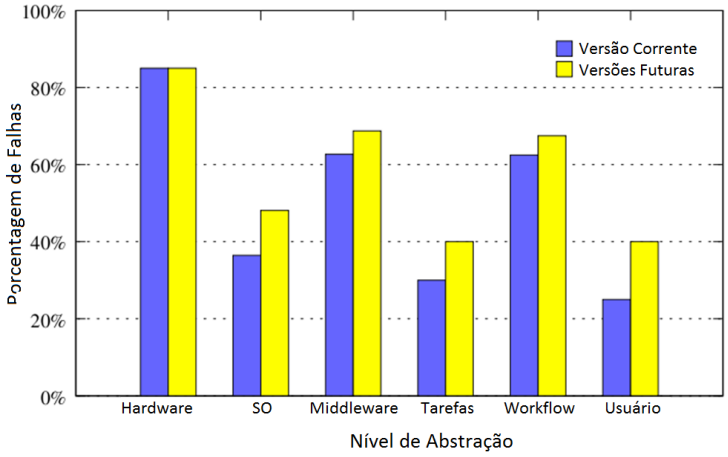

Figura 5.4: Média de detecção de falhas [38]

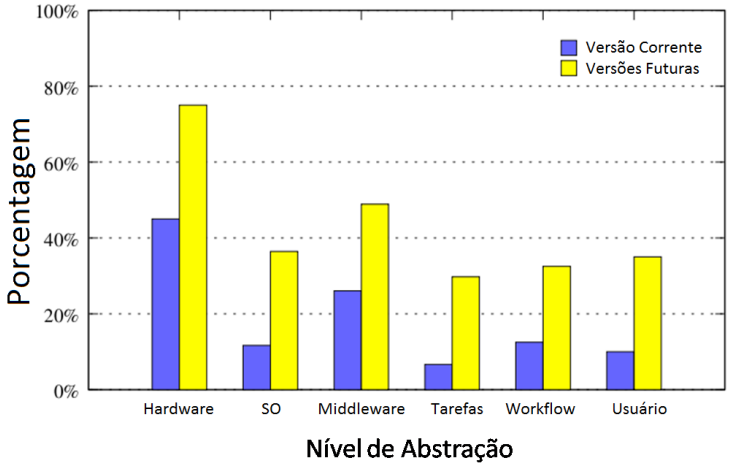

Figura 5.5: Média de recuperação de falhas [38]

recuperação (Figura 5.5). Por sua vez, a prevenção ainda pode ser considerada praticamente inexistente com porcentagens inferiores a 10% em todos os tipos de falha avaliados (Figura 5.6).

Baseado nos resultados descritos em [28] e [38] podemos chegar à conclusão de que, apesar de ser um aspecto de amplo interesse prático, com resultados efetivos e significativos, a tolerância e prevenção de erros em Grid ainda se conserva em um estado inicial e necessita de maiores desenvolvimentos na expectativa de obterem-se os ganhos esperados.

A seguir, serão apresentados e discutidos alguns trabalhos sobre tolerância a falhas de *workflows* em *Grid*.

5.1 Grid Worflow System - Grid-WFS [26]

Um *framework* para o tratamento de falhas em ambiente de Grid é proposto em [26]. Os autores argumentam que cada aplicação pode possuir uma semântica de falhas própria, ou seja, sistema de notificação e manejo de falhas específicas àquela tarefa ou aplicação. Neste sentido, diversos tipos de falhas, específicos ou não, deverão ser tratadas de diversas

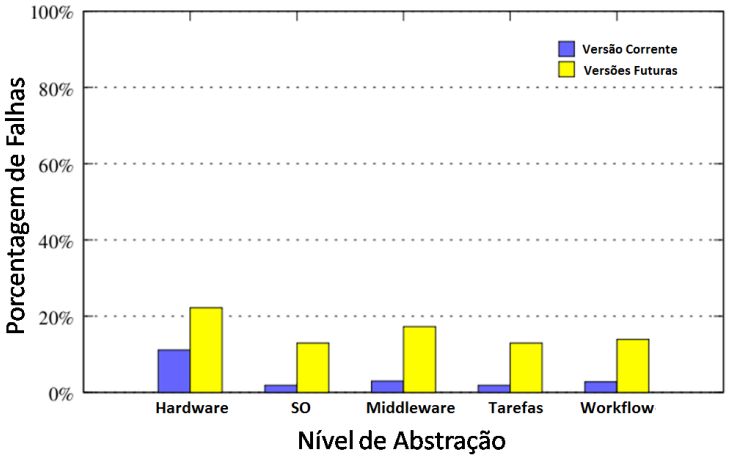

Figura 5.6: Média de prevenção de falhas [38]

maneiras dependendo da semântica de execução tanto da tarefa quanto da aplicação como um todo.

Desta forma, colocam-se como requisitos para o *framework* proposto o suporte a diversas estratégias de manejo de falhas, a separação das políticas de manejo do código das aplicações e o suporte a mecanismos de tolerância a falhas definidos pelo usuário. Com estes requisitos em mente, foi desenvolvido o protótipo de um Sistema de *Workflow* para Grids (Grid-WFS) consistindo de três componentes principais:

1. Uma Linguagem de Definição de Processo do *Workflow* (XML WPDL) que permite a especificação do *workflow* em um DAG;
2. Um *Workflow Engine* que controla e acompanha a execução das tarefas;
3. Serviços para prover a infraestrutura de intermediação de recursos (*resource brokering*).

A abordagem proposta baseia-se no conceito de *heartbeat* e notificações de evento para permitir a identificação de falhas. No recebimento do *heartbeat*, as aplicações interpretam a mensagem recebida para definir o estado das suas tarefas. O *framework* utiliza estruturas de *workflow* nas quais integram-se técnicas de tolerância a falhas em nível de tarefa e de *workflow*. As técnicas em nível de tarefa visam mascarar as falhas das tarefas sem que o fluxo da estrutura de *workflow* seja afetado. Já as técnicas em nível de *workflow* podem, por exemplo, alterar o fluxo de execução para lidar com exceções definidas pelo usuário assim como com falhas que não puderam ser mascaradas pelas técnicas em nível de tarefa.

Além disto, o *framework* é flexível o suficiente para combinar diferentes técnicas em nível de tarefa ou mesmo combiná-las com técnicas em nível de *workflow* e modificar as políticas de manejo de falhas de forma fácil e incremental, alterando a estrutura de *workflow*.

Destes três componentes, o de maior relevância é a *Workflow Engine*, implementado como uma aplicação *stand-alone* sobre a plataforma Globus Toolkit 2.0. É ela a responsável por receber o DAG do *workflow* e, a partir destes dados, definir as tarefas prontas para a execução de acordo com suas dependências. Em seguida, ela submete as tarefas para execução por meio do protocolo Globus GRAM. Durante a execução, ela monitora

as tarefas e determina seu estado final (completo, falho ou exceções) usando o mecanismo de detecção de falhas definido em [25]. Em seguida se reinicia o processo definindo-se as próximas tarefas de acordo com as respectivas dependências.

5.2 Workflows QoS-Aware e com tolerância a Falhas [3]

Bian et al. [3] descrevem uma composição para *workflows* baseadas em Qualidade de Serviços (QoS). Para isto trabalham com os conceitos de restrições, no que diz respeito a parâmetros específicos mínimos, e de preferências do usuário, que acrescentam um peso aos atributos de QoS antes de serem processados pelo mecanismo de mapeamento do *workflow*.

Os usuários, antes da execução, podem expressar seus requisitos de QoS de duas formas: (i) Limites impostos, ou seja, requisitos mínimos do recurso; e (ii) preferências, que são expressas na forma de grau de importância nos requisitos, aumentando a probabilidade de que um recurso com aquelas características seja designado para a execução daquela tarefa.

Para implementar este sistema, utiliza-se o `Web Services - Power Grid` (WS-PG). O WS-PG possui uma estratégia de mapeamento de tarefas que utiliza informações registradas para designar uma tarefa para o nó mais apropriado, por meio de cálculos que levam em consideração as restrições e preferências associadas a cada tarefa.

Para lidar com o problema de tolerância a falhas, considera-se o fato de que na arquitetura do *PowerGrid* [15] cada nó exerce um papel (*role*) e que o sistema como um todo falharia no caso de falha no Coordenador. Assim, criam-se nós de redundância ou *mirror sites* que assumam este papel no caso de falhas. No caso de falhas em um Recurso WS, as tarefas são simplesmente reagendadas em outros servidores disponíveis. Finalmente, o *Grid Daemon* é o responsável por gerenciar o estado dos componentes. Para isso, ele periodicamente se comunica com eles e armazena seu estado no banco de dados de forma que, ao ser solicitado um recurso novo, exclua-se da lista os indisponíveis.

Para a avaliação dos resultados, Bian et al. [3] conduziu experimentos utilizando *PowerGrid* capacitado para *workflows* e o próprio *Power Service* como aplicação de teste. Na comparação entre uma composição de *workflow* capacitada para QoS e uma sem esta capacitação utilizaram até 140 instâncias de *workflow* e obtiveram um aumento no tempo de execução desprezíveis com até 60 instâncias. Porém com as 140 instâncias a sobrecarga já representava de 10 a 15% do tempo de execução. Em outra etapa de avaliações, compararam composições com e sem capacidade de tolerância a falhas e conseguiram aumentar a taxa de *workflows* executados com sucesso em torno de 5% ao utilizarem a tolerância a falhas.

5.3 Combinação de Técnicas de Tolerância a Falhas e de Escalonamento para *Workflows* em Grid [51]

Este trabalho busca estudar a efetividade das técnicas de tolerância a falhas com métodos de escalonamento para *workflow* existentes [51]. Desta forma, possui foco na avaliação do desempenho, custo e efetividade de métodos de tolerância a falhas, especificamente

a replicação passiva e ativa, quando associados a diferentes formas de escalonamento do *workflow*. Para isto, avaliam estes atributos sob diversos modelos de confiabilidade, precisão da predição de falhas e tipos de aplicação do *workflow*.

Para o escalonamento, foram utilizados dois algoritmos como base. O primeiro é o HEFT [47] que designa para cada tarefa, em ordem de prioridade, o recurso que a completaria mais cedo. O segundo algoritmo é o algoritmo de duplicação DSH [30] que combina idéias de escalonamento de listas com duplicatas para reduzir a duração total do *workflow*. Ele calcula para cada tarefa, novamente em ordem de prioridade, o momento de início da tarefa sem duplicar nenhuma de suas predecessoras. Em seguida, aloca duplicatas das tarefas pai até que não seja mais viável ou que isto não mais melhore o horário de início da tarefa seguinte.

Estes algoritmos não possuem tolerância a falhas. Por isso acrescentou-se ao HEFT e ao DSH uma restrição de confiabilidade e um mecanismo de replicação ativa. No HEFT, no caso de todas as tarefas filho estarem em um mesmo recurso, este mecanismo é invocado e localizará um outro recurso para a tarefa pai de forma a satisfazer a restrição de confiabilidade. No DSH o processo é parecido. A tarefa pai é duplicada e então o algoritmo de replicação ativa é acionado para alcançar uma confiabilidade superior.

Além da replicação ativa, também se estipulou uma estratégia de replicação passiva. Optou-se por uma estratégia *lightweight*, que salva apenas a localização atual dos dados intermediários. Em caso de falhas a tarefa é migrada para outro recurso e reiniciada a partir do último checkpoint válido.

Através de experimentações em uma simulação de Grid multi-cluster, totalizando 9 clusters de máquinas com processadores equivalentes distintos foram realizadas medições de desempenho variando de acordo com o algoritmo de escalonamento e técnica(s) de tolerância a falhas.

Nos experimentos mostrou-se que a probabilidade de sucesso aumenta consideravelmente (entre 7 e 16%) utilizando uma das técnicas de tolerância a falhas e até 23% quando se utiliza uma combinação das duas. No entanto, este aumento da probabilidade de sucesso vem agregado a um aumento no tempo total de execução de 5 a 10% e também a utilização de recursos aumenta entre 50% , utilizando a replicação passiva, e 150% quando utiliza-se a replicação ativa.

Como resultados finais, os autores concluem que técnicas de tolerância a falhas são benéficas no sentido de aumentar a confiabilidade da execução de *workflows* em até 200%, quando utilizando recursos com baixa confiabilidade, sem afetar o desempenho mais do que 10%.

5.4 Escalonamento Robusto de *Workflows* Científicos [49]

Segundo Wang et al. [49], as pesquisas atuais se preocupam com o *trade-off* entre o custo e o tempo de execução de uma tarefa assumindo que todos os participantes do Grid sejam igualmente idôneos e confiáveis. No entanto, os provedores de serviço são entidades autônomas que podem se comportar de maneira desonesta quando isto lhes parecer vantajoso. Por isso propõem agregar este conceito de confiança ao ambiente de

Grid e introduzem um modelo que incorpora a confiança, um indicador da probabilidade de que o agente que fornece o serviço irá honrar seus compromissos.

Para gerenciar o nível de confiança, Wang et al. [49] optaram por um gerenciador centralizado onde a confiança de cada agente pode ser obtida por meio de uma *query* junto ao gerenciador, que classifica os agentes de acordo com o retorno fornecido pelos usuários daquele serviço.

Em sua implementação, utilizam o conceito de Coordenador, entidade responsável pelo gerenciamento da execução que realiza as tarefas de avaliação, escalonamento e monitoramento das tarefas. O Coordenador recebe um grafo que define um *workflow* e o decompõe em um conjunto de planos de execução expressos como Grafos Acíclicos Direcionados (DAG). Em seguida, avalia cada um dos planos possíveis para selecionar aquele mais promissor no que diz respeito à robustez para servir de entrada para o próximo estágio. Na etapa de escalonamento, designa-se para cada tarefa um agente de serviços de forma a encontrar um planejamento adequado visando maximizar a robustez, porém observando as restrições de prazo e custo estipuladas. No decorrer da execução o Coordenador acompanha o processo e pode tomar a decisão de realizar um novo escalonamento ou uma nova avaliação dos planos viáveis.

5.5 Recuperação de Erros para *Workflows* com SLA [40]

Um Acordo de Nível de Serviços (*Service Level Agreement* ou SLA) permite realizar o gerenciamento das expectativas, a regulamentação do uso de recursos e para a especificação dos custos [40]. A utilização do SLA garantiria a Qualidade do Serviço (QoS) desejada e pré-estabelecida. O objetivo é, portanto, buscar um algoritmo que consiga um remapeamento rápido como resposta a uma falha do tipo *fail-stop* de um ou mais recursos.

O SLA contudo é posto à prova em casos de erros, que possuem a real capacidade de retardar a execução das tarefas e do *workflow*. Por isso, os sistemas baseados em SLAs devem possuir algum grau de preparação para lidar ou eliminar seus efeitos. Mais especificamente, este trabalho se concentra nas falhas decorrentes da desconexão de um ou mais recursos do Grid em um dado instante. Quando estas falhas ocorrem, todas as tarefas que estivessem rodando ou aguardando por algum desses recursos são consideradas falhas. Além disto, muitas tarefas aguardando execução em outros recursos podem ficar impossibilitadas de serem executadas por indisponibilidade dos dados de entrada, sob o risco de comprometer a integridade da execução. Mais ainda, o cancelamento da execução do *workflow* resultaria em multa previamente estipulada no SLA.

Para sanar este problema, [40] define um mecanismo de recuperação de erros que determina todos os *workflows* afetados bem como suas respectivas tarefas que necessitarão de um novo recurso, cria novos *workflows* e determina a prioridade de remapeamento e, finalmente, utiliza o algoritmo proposto (w-Tabu) para realizar o mapeamento para novos recursos.

Para efeitos de comparação, os autores compararam seu algoritmo com os algoritmos minmin [10], maxmin [10], *suffer*, GRASP [4], w-DCP [33]. No que diz respeito ao tempo

de execução, o algoritmo w-Tabu revelou-se realmente rápido, conseguindo um tempo máximo de 13 segundos para um *workflow* com 21 tarefas.

5.6 Previsão de Falhas com Tolerância a Falhas Adaptativa [31]

Em [31] se propõe uma nova abordagem para a replicação passiva, chamada de FT-Pro. Na expectativa de reduzir o número de checkpoints coordenados necessários em um sistema de Grid, utiliza-se a predição de falhas para adaptar a estratégia de tolerância a falhas conforme as previsões. Para isso, utiliza um algoritmo baseado em custo para selecionar uma dentre as diversas atitudes preventivas possíveis, de acordo com a acurácia da previsão. Desta forma, visa-se reduzir o número de checkpoints necessários e evitar falhas para, em última instância, minimizar o tempo de execução de aplicações paralelas.

Para isto, o FT-Pro possui uma abordagem cooperativa onde os programadores podem indicar pontos de decisão de forma que o FT-Pro possa selecionar, em tempo de execução, a ação preventiva mais adequada para aquele momento. As ações que o FT-Pro pode selecionar são:

1. **Migração de tarefas:** Após um checkpoint coordenado, as tarefas que estejam em nós suspeitos, ou seja, cujas previsões indiquem alta probabilidade de falhas, são migradas para recursos mais confiáveis.
2. **Checkpoint:** Todos os processos pausam a execução e conduzem um checkpoint coordenado.
3. **Nada a fazer:** Caso haja uma baixa probabilidade de falhas na configuração atual, o FT-Pro pode optar por não interromper a execução para realizar um checkpoint.

Este método consegue utilizar um algoritmo de avaliação baseado em custos para integrar migração proativa de tarefas e replicação passiva e realizar a predição de falhas em cada nó. Vale ressaltar que esta abordagem, ao invés de objetivar a otimização da disponibilidade do sistema, concentra-se na diminuição do tempo de execução, ainda que em detrimento de eventuais falhas. Para efeito de comparação, utilizam o sistema com previsão de falhas e um sistema de replicação passiva periódica, sob uma ampla gama de acurácias das predições, tamanhos de problemas e parâmetros de sistema.

Para a avaliação da melhora da eficiência resultante da abordagem proposta, foi utilizado um log de erros de um supercomputador real no NCSA (*National Center for Supercomputing Applications*). Na comparação entre o resultado da abordagem tradicional de replicação passiva com checkpoints periódicos e a abordagem do FT-Pro foi observado um ganho entre 13.6% e 14.3%. Os autores atribuem este ganho de desempenho ao fato de falhas serem essencialmente raras e da probabilidade de duas falhas simultâneas ser baixa.

5.7 Mecanismo Flexível para Tolerância a Falhas no Integrade [13]

Este trabalho dá continuidade ao que foi realizado em [43] e o expande de forma a prover mais técnicas de tolerância a falhas, uma vez que o Integrade só possuía implementada a replicação passiva a nível de aplicação e dependente de alterações no código da aplicação por meio de um pré-compilador. De Souza et al. [13] acrescentam ao Integrade a técnica de *retry* e replicação ativa, inclusive permitindo a customização de parâmetros relacionados ao mecanismo de tolerância a falhas. Permite ao usuário escolher se será utilizado ou não o sistema de replicação passiva em conjunto com a ativa, o intervalo entre os *checkpoints* e quantidade de réplicas a serem criadas.

Para permitir esse grau de controle pelo usuário, alguns elementos foram acrescentados à arquitetura descrita em 2.6. São eles:

1. **ckpLib:** Uma biblioteca de checkpoint, ckpLib, que provê a funcionalidade para criar checkpoints esporádicos;
2. **ADRs:** Os Repositórios Autônomos de Dados (ADRs) que residem em máquinas que compartilham seus recursos com o Grid e provê um armazenamento estável;
3. **CDRM:** É um Gerenciador de Repositórios de Dados do Cluster, que gerencia os ADRs disponíveis e a localização dos dados dos checkpoints;
4. **ARMs:** Gerenciador de Réplicas de Aplicações, que instancia as diversas réplicas e aguarda até que uma se encerre com sucesso para encerrar às demais e liberar os recursos.

Para efeito de comparação e avaliação de resultados, os autores criaram 4 cenários distintos e simularam a ocorrência de falhas. Em cada um dos cenários, variava-se o tempo médio para a falha (*mean time to failure* ou MTTF) e comparava-se o resultado de acordo com a técnica de tolerância: retry (`Rt`), replicação passiva (`Ck`), ativa (`Rp`) ou ativa com checkpoint (`RpCk`). Dos resultados observados, a técnica de `RT` foi sempre a mais lenta dentre as opções avaliadas. No entanto, o desempenho das demais técnicas variava amplamente. Em um dos cenários propostos, possuindo um MTTF de 20 segundos, a `Rp` demorava 100% a mais, quando comparada à `RpCk`. No entanto, no mesmo cenário porém com um MTTF de 120 ou de 180 tanto a `Rp` quanto a `RpCk` possuíam praticamente o mesmo desempenho. Em outro cenário, a `Rp` conseguia concluir um *workflow* 22% mais rápido que a `RpCk`. De fato, considerando os dados obtidos, nenhuma das técnicas (`Ck`, `Rp` ou `RpCk`) era sempre superior ou sempre inferior às demais e a escolha adequada dependia das características de cada aplicação ou Grid.

5.7.1 Relacionamentos entre os módulos

Como descrito em 5.7, os relacionamentos do Integrade (ver Seção 2.6, Figura 2.7) foram adaptados de forma a incorporar a possibilidade de realizar a replicação passiva, ativa ou a combinação das duas no Integrade. Nesta arquitetura os relacionamentos foram reestruturados conforme a Figura 5.7.

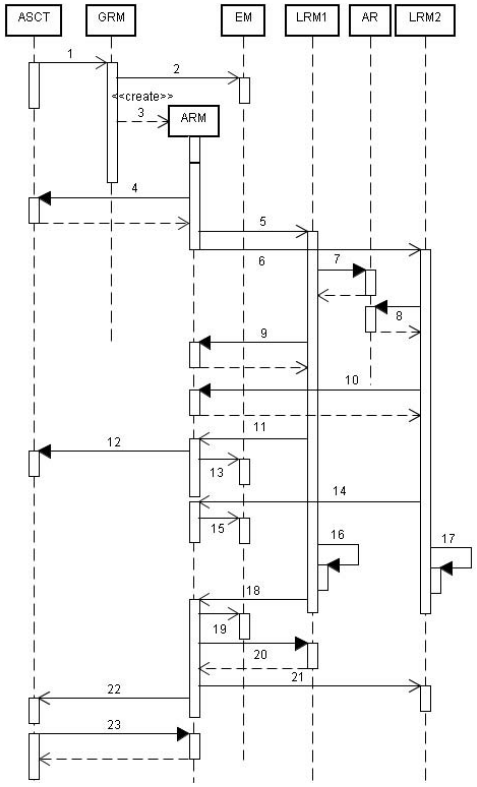

Figura 5.7: Reestruturação das interações entre os módulos de forma a permitir a Replicação Ativa [13]

Nesta nova arquitetura, as interações entre os módulos também foram modificadas. Inicialmente o usuário submete um pedido para executar uma aplicação e informa a quantidade de réplicas que deverão ser geradas. O pedido é encaminhado ao GRM (1) que escalona a execução para os recursos disponíveis, notificando o EM que a aplicação foi escalonada com sucesso (2). O GRM instancia um novo ARM e repassa o pedido de execução juntamente com os nós designados (3). O ARM obtém os arquivos de entrada, caso existam, do ASCT (4) e encaminha o pedido de execução para o LRM rodando em cada nós designado (5 e 6), gerando assim as réplicas da aplicação. Neste exemplo, apenas duas cópias são criadas. Cada LRM obtém os binários de um Repositório de Aplicação (7 e 8), solicita os arquivos de entrada para o ARM (9 e 10), e responde com uma notificação que a execução foi aceita (11 e 14). Assim que algum LRM confirme a aceitação do pedido, o ARM notifica o ASCT (12). Também notifica o EM sobre cada cópia da aplicação (13 e 15). Cada LRM inicia e monitora o progresso da aplicação em seu nó(16 e 17), notificando o ARM no momento da conclusão. O ARM repassa esta informação ao EM (19) e obtém os resultados (arquivos de saída) (20).

Caso haja alguma falha, como uma máquina ser desligada, durante a transferência dos resultados, o ARM considera a computação como não concluída e aguarda o término de outra réplica. Se nenhuma falha ocorre, o ARM solicita o encerramento das demais réplicas (21) e notifica o ASCT do sucesso da execução (22). Finalmente, o ASCT pode obter os resultados finais junto ao ARM (23).

5.8 Tabela Comparativa

A Tabela 5.1 apresenta um quadro comparativo das abordagens discutidas neste capítulo. Dos trabalhos apresentados, o que se observou foi que a maioria [51, 3, 49, 40, 31, 13] utiliza apenas técnicas previamente definidas e selecionadas, sendo a mais comum a replicação ativa utilizada em [51, 3, 26, 13]. Apenas [26] permite o tratamento de exceções definidas pelo usuário.

Além disso, os trabalhos [51, 3, 49] utilizam um esquema fixo para definir a técnica de tolerância a falhas a ser utilizada em qualquer circunstância; [40, 31] utilizam um algoritmo próprio que realiza a seleção dinâmica em tempo de execução; e [26, 13] permitem ao usuário fazer a seleção da técnica apropriada antes da submissão da tarefa ou do *workflow*. Nenhum dos trabalhos estudados consegue tanto buscar o atendimento das expectativas do usuário quanto a seleção dinâmica das técnicas de tolerância a falhas em tempo de execução. Em todos os trabalhos estudados, uma dessas características era sempre negligenciada.

Com relação aos *middlewares* utilizados, o Globus [51, 26, 40] e a simulação [49, 31] foram os mais utilizados.

Outro aspecto que devemos considerar é a necessidade de alterações no código da aplicação. Três trabalhos [40, 31, 13] apresentavam a necessidade de que o código fosse especialmente desenvolvido para aquele fim.

Finalmente, alguns trabalhos procuravam tratar de aspectos que não são considerados na maior parte dos demais trabalhos: [3, 13] visavam oferecer uma arquitetura orientada a serviços (SOA); [40, 3] tratavam da qualidade dos serviços (QoS); [49] introduz o conceito de que nem todos os recursos são necessariamente confiáveis e [13] traz o objetivo de possibilitar uma computação autonômica.

Assim, podemos perceber das linhas gerais delineadas na Tabela 5.1 que, a nosso conhecimento, as abordagens atuais não permitem ao usuário uma flexibilidade suficiente para definir a seleção das técnicas de tolerância a falhas por meio de regras e em tempo de execução. Em sua maioria, os usuários ficam restritos a um único mecanismo de seleção de técnicas de tolerância a falhas [40, 31] ou mesmo a uma técnica em particular [49, 40], ainda que este mecanismo ou esta técnica não sejam ideais para atender às especifidades de sua aplicação. Mesmo *frameworks* mais flexíveis como o citado em [26, 13] ainda demandam do usuário que faça manualmente a definição das técnicas para cada uma das tarefas de cada *workflow*. Esta necessidade acaba por desconsiderar alterações na topografia do Grid no decorrer da execução. Isto impede o reuso de políticas em diferentes *workflows*.

Mais ainda, as técnicas de tolerância a falhas implementadas são, em geral, restritas a um conjunto considerado suficiente por seus respectivos autores e não permitem a inclusão de novas técnicas pelos usuários. Isto impossibilita a expansão do modelo com novas técnicas e impede os usuários de, por exemplo, incluir para uma certa aplicação uma técnica de tolerância a falhas que lhe seja específica e muito eficiente.

Tabela 5.1: Tabela Comparativa entre as abordagens avaliadas

Nome	Técnicas de Tolerância a Falhas	Método de Seleção	Middleware	Modificação do Código	SOA	QoS	Confiança dos Recursos	Visa a Computação autonômica
[51]	Replicação Ativa, Passiva e de todo o DAG	Fixo	TeraGrid (Globus Toolkit e Condor-G)	Não		Não	Todos idôneos	Não
[3]	Replicação Ativa e Nova tentativa em outro recurso	Fixo, dependente do tipo de recurso	PowerGrid	Não	Sim	Sim	Todos idôneos	Não
[26]	Replicação Ativa, Passiva e de tempo, tarefa alternativa e exceções definidas pelo usuário	XML WPDL definido pelo usuário	Globus Toolkit 2.0	Não	Não	Não	Todos idôneos	Não
[49]	Nova tentativa em outro recurso	Fixo	Simulação	Não	Não	Não	Gerenciado por entidade externa ao recurso	Não
[40]	Nova tentativa em outro recurso	Algoritmo próprio	Globus Toolkit 3.2	Sim	Não	Sim	Todos idôneos	Não
[31]	Nova tentativa em outro recurso, Replicação Passiva	Avaliação baseada no Custo	Simulação	Sim	Não	Não	Todos idôneos	Não
[13]	Nova Tentativa, Replicação Passiva e/ou Ativa	Definido pelo usuário	Integrade	Sim	Sim	Não	Todos idôneos	sim

Capítulo 6

Framework para Flexibilização da Escolha de Tolerância a Falhas

6.1 Decisões de Projeto

Como foi definido no Capítulo 4, *workflows* são sequências de tarefas com dependências temporais, frequentemente demandando um longo período de execução. Por sua vez, o Grid possui variações na disponibilidade de recursos e é propenso a falhas [28]. Assim, as técnicas utilizadas para contornar as falhas encontradas não devem ser rígidas ou definidas com base em uma dada topologia do Grid [25], dada a impossibilidade de prever as alterações na disponibilidade dos recursos do Grid.

Por outro lado, os mecanismos estudados de tolerância a falhas em Grid não são flexíveis a ponto de permitir ao usuário definir o método utilizado para a seleção dos mecanismos de tolerância a falhas ou acrescentar novas técnicas de tolerância a falhas às já existentes.

Dos trabalhos explicitados na Tabela 5.1 nenhum prevê a inclusão de novas técnicas de tolerância a falhas e, em geral, o método de seleção do mecanismo de tolerância a falhas ou é definido de forma única [51, 3, 49, 31, 40] ou é definido de forma rígida pelo usuário antes do início da execução do *workflow* [25, 13].

É precisamente neste nicho que nosso trabalho se enquadra. No presente capítulo, apresentamos um *framework* desenvolvido sobre a infraestrutura do middleware de Grid Integrade (Seção 2.6), com interfaces bem definidas, que se propõe a permitir ao usuário definir um conjunto de regras para a seleção das técnicas no momento da execução e a adição de novas técnicas, inclusive aquelas específicas a uma dada aplicação.

As regras para a seleção da técnica de tolerância a falhas e seus parâmetros são definidas pelo usuário para cada tarefa individualmente. A técnica de tolerância a falhas e a parametrização escolhida dependerão portanto das regras do usuário e da topologia corrente do Grid em tempo de execução. Permitindo aos usuários definir as regras para seleção de técnicas, damos a eles a oportunidade de se aproveitar das peculiaridades de cada tarefa ou *workflow* e da topologia dinâmica do Grid.

Outra vantagem potencial será permitir que sejam consideradas as expectativas dos usuários e as características dos recursos no processo de escolha. Por exemplo, um usuário pode optar por designar uma tarefa crítica a um recurso de alta confiabilidade com replicação passiva de forma a garantir um prazo previamente acordado ou cumprimento de

acordo de nível de serviço. Outro usuário poderia optar por designar a mesma tarefa a diversos recursos de baixa confiabilidade em replicação ativa, na esperança de obter ganhos de desempenho.

A inclusão de novas técnicas deve ocorrer de forma simples. Com isso, o usuário deverá ter liberdade para implementar formas diferentes de contornar erros, novas técnicas mais eficientes ou tratamentos específico para tipos de erros anteriormente não previstos. Para permitir tal liberdade de implementação, optamos por criar uma arquitetura distribuída, onde a comunicação entre os diversos módulos do *framework* se desse através de invocação remota de métodos - RMI [5]. Assim, cada técnica de tolerância a falhas terá um maior controle sobre sua implementação, podendo utilizar métodos variados para alcançar seus objetivos.

Além disso, optou-se por encapsular todos os parâmetros dos módulos responsáveis pela execução das tarefas em um arquivo texto. Com isso, não existe uma predefinição da quantidade ou tipos de parâmetros para cada nova implementação destes módulos. Neste sentido, optamos também pela não-definição do formato deste arquivo. Assim, cada implementação pode definir livremente o método mais conveniente para expressar seus parâmetros. Desde linguagens como XML ou YAML até parâmetros de tamanho fixo e ordem específica.

Outro aspecto importante do framework é que, da forma como está estruturado, o *framework* se propõe a lidar com técnicas de tolerância a falhas em nível de tarefas (ver 5.1).

6.2 Visão Geral

De forma geral, o *framework* proposto nesta dissertação busca permitir aos usuários definirem, para um *workflow* qualquer, as regras através das quais o mecanismo de tolerância a falhas de cada tarefa será escolhido em tempo de execução.

O usuário fornece como entrada um *workflow* e suas tarefas, definidos por meio de um arquivo, os arquivos de entrada para o *workflow* e um terceiro arquivo onde é especificada as regras segundo as quais a técnica de tolerância a falhas será selecionada, considerando os possíveis cenários para a topologia do Grid e disponibilidade de recursos no momento da execução.

O *framework*, de posse destas informações, irá submeter as tarefas ao Grid quando suas dependências já estiverem satisfeitas. Contudo, no momento da submissão ele não irá simplesmente submeter as tarefas. O *framework* irá solicitar ao Grid as informações sobre os recursos disponíveis e, em seguida, aplicar as regras definidas pelo usuário. Com base nessas regras, o *framework* finalmente decide a técnica mais apropriada de tolerância a falhas, cria uma instância do mecanismo que a implementa e só então submete a tarefa ao Grid.

Com isto, as técnicas de tolerância a falhas não precisam estar definidas antes do início da execução do *workflow* e podem se ajustar às mudanças no Grid que ocorram ao longo da execução. Além disso, o processo de seleção não está preso a um critério predefinido e possivelmente não adequado às características e peculiaridades do *workflow* em questão. Ao invés disso, permite-se que este critério seja definido de acordo com o *workflow* sendo executado pelo usuário. Para isto, o usuário deve definir as regras que

nortearão o *framework* na seleção da técnica de tolerância a falhas mais apropriada dada a topologia corrente do Grid.

6.3 Arquitetura do Framework

O *framework* proposto nesta dissertação é composto de diversos módulos, cada qual com uma função bem definida. Estes módulos estão ilustrados na Figura 6.1. Primeiramente, o usuário define um *workflow* e suas tarefas em um arquivo. Além deste arquivo, o usuário define outro arquivo que contém as regras que irão reger o processo de seleção das técnicas de tolerância a falhas e os arquivos de entrada do seu *workflow*.

O procedimento de submissão, execução e controle dos *workflows* estão ilustrados na Figura 6.2. Inicialmente o usuário submete os arquivos de entrada e o arquivo com as regras ao *framework* por meio da ferramenta WSCT (*Workflow Submission Control Tool*). Essa ferramenta converte o arquivo descritor do *workflow* em um objeto Java contendo suas informações e o repassa ao WFM (*Workflow Manager*). Este é o encarregado de controlar a evolução do *workflow*, avaliando cada *workflow*, definindo o conjunto de tarefas prontas para serem executadas e, com base nessa análise, submetendo as tarefas prontas ao FTM (*Fault-Tolerance Manager*). O FTM solicita ao FTSM (*Fault-Tolerance Selection Mechanism*) o nome do FTEC (*Fault-Tolerant Execution Coordinator*) que deverá ser utilizado. O FTSM cruza os dados dos recursos do Grid atualmente disponíveis com as regras definidas pelo usuário e informa o identificador do FTEC escolhido. O FTM então solicita ao servidor daquele FTEC a criação de uma *thread* para acompanhar o processo de execução da tarefa. Toda tarefa executada no *framework* é acompanhada por uma *thread* do FTEC que lhe foi designado. Uma vez que a tarefa esteja concluída, a thread do FTEC que a executa notifica o fato ao FTM e ao WFM que, respectivamente, dão a tarefa como concluída e avaliam se há mais alguma tarefa cujas dependências estejam satisfeitas. O processo se repete até que todas as tarefas tenham sido concluídas.

Como o ambiente de Grid é, por definição distribuído, toda a comunicação entre os módulos do framework foi realizada através do RMI. Ainda que nesta primeira versão todos os módulos rodem na máquina central junto ao GRM (*Global Resource Manager*), utilizando uma comunicação entre módulos realizada através de invocação remota de métodos a transição em uma versão futura para um ambiente distribuído torna-se razoavelmente simples.

O *framework* para seleção flexível de técnicas de tolerância a falhas se baseia em dois aspectos básicos da execução em Grid: o controle da topologia corrente por um gerenciador de recursos e a capacidade de submissão de tarefas para execução. Tais funcionalidades são oferecidas pela maior parte dos middlewares de Grid. Por exemplo, no Integrade (ver Seção 2.6) estes pilares correspondem, respectivamente, aos módulos GRM (*Global Resource Manager*) e ASCT (*Application Submission and Control Tool*). A partir das informações acerca da topologia corrente, o *framework* é capaz de selecionar o FTEC apropriado que realizará a submissão e o acompanhamento das tarefas por intermédio do ASCT.

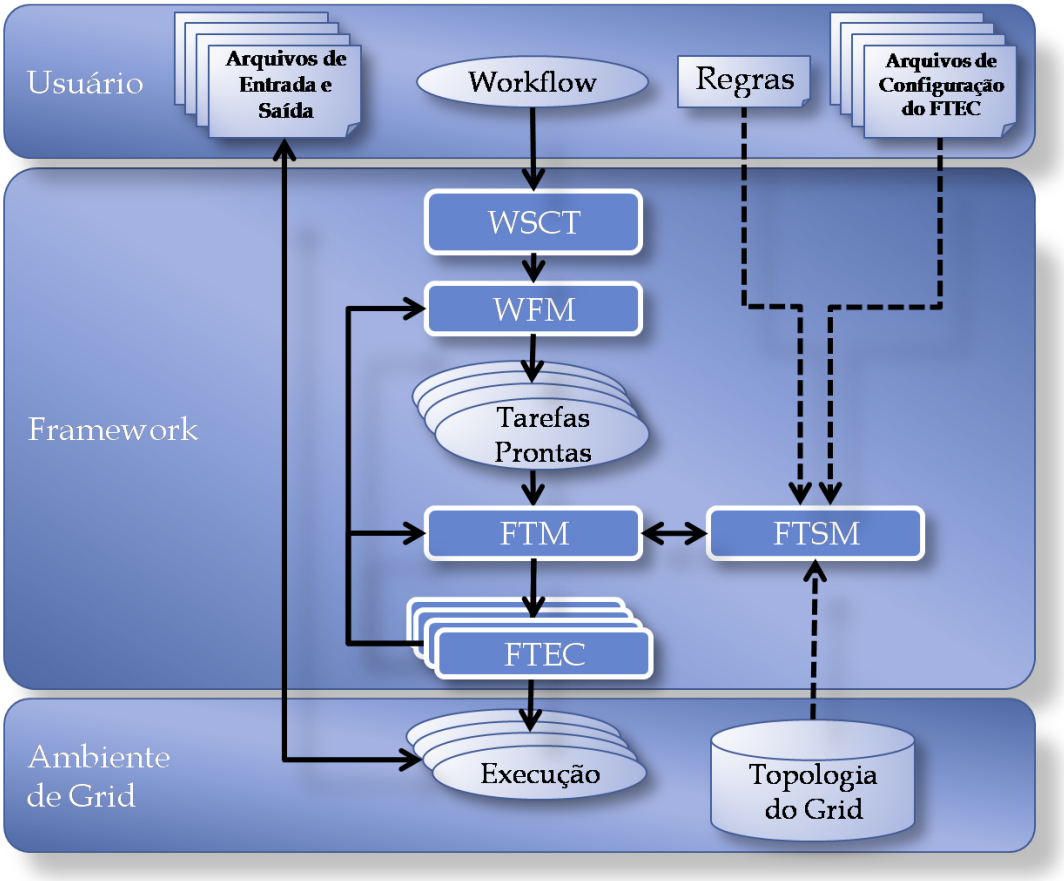

Figura 6.1: Arquitetura do *framework*

6.4 Módulos do *Framework*

Para a obtenção da flexibilidade desejada, foram criados diversos módulos divididos em duas categorias: os que dizem respeito à gestão do *framework* e são executados no mesmo nodo que o Gerenciador Global de Recursos (GRM) do Grid e os que se propõem a ser utilizados pelos clientes e são executados nos nodos provedores de recursos. Uma descrição detalhada de cada módulo será apresentada nas seções subsequentes.

6.4.1 Workflow Submission & Control Tool - WSCT

O *Workflow Submission & Control Tool - WSCT* permite ao usuário submeter um *workflow* ao *framework* e obter seus resultados ao final da execução.

Para isto, o usuário preenche um documento no formato YAML [2] que descreve o *workflow*. O formato YAML foi escolhido por ser simples e já ser o formato de arquivo para vários arquivos de configuração para o Deployer do Integrade [35]. Para cada tarefa o usuário informa um registro YAML no formato descrito na Figura 6.3.

Cada registro YAML é iniciado pela sequência de caracteres ---. Com isto, indica-se ao parser tratar-se de um novo registro. Após o marcador de início do registro, cada campo é identificado pelo nome seguido do caractere " : " e o valor que a ele deve ser atribuído.

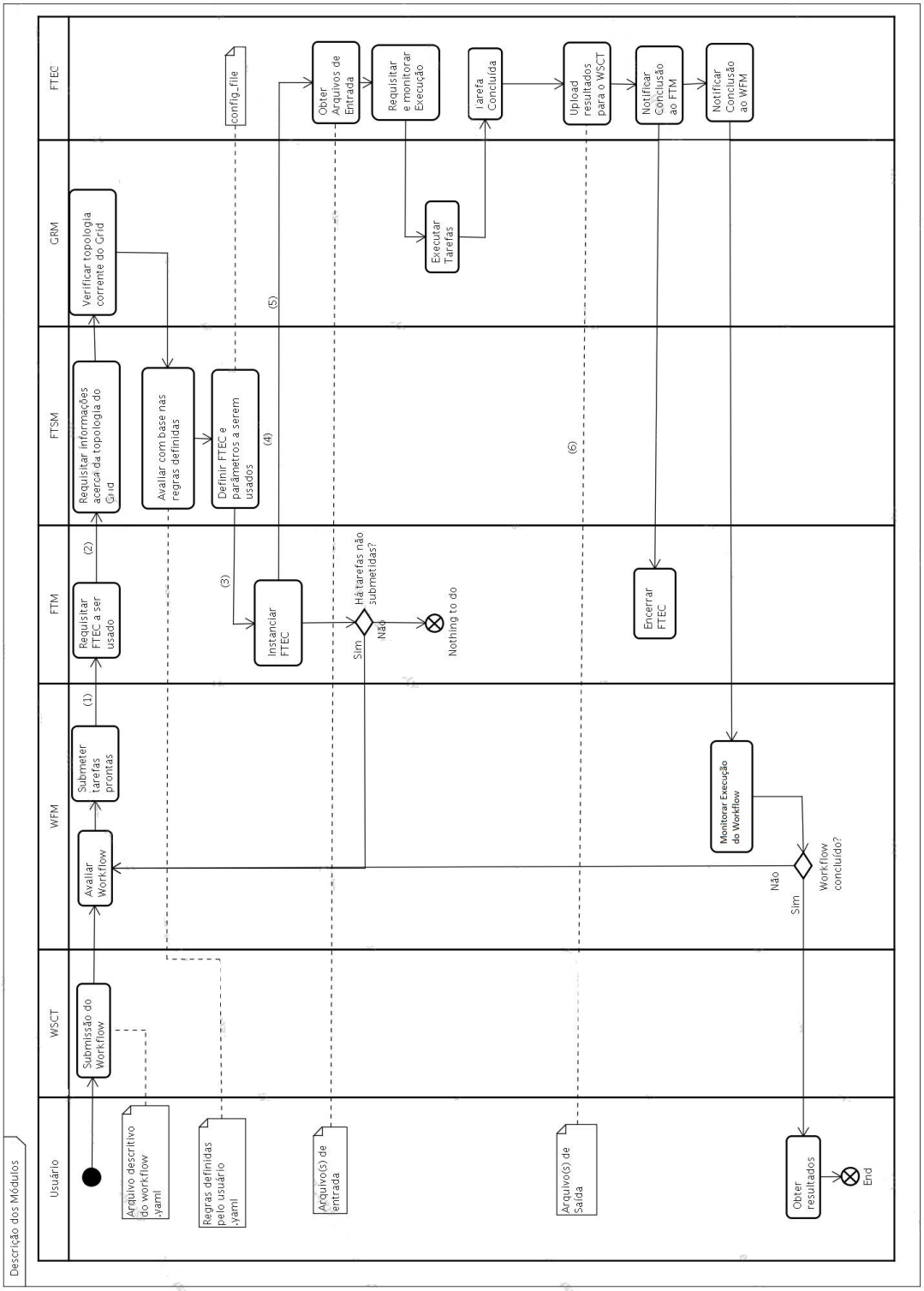

Figura 6.2: Descrição das tarefas por Módulo para Execução Tolerante a Falhas

```
---
    name :    < Denominação da tarefa >
    duration :   < Tempo previsto em minutos para a execução da tarefa>
    binary:   < Tipo do binario a ser utilizado >
    inputfile:   < Arquivo de configuração para submissão do Integrade [35]>
    outputfile:   < Arquivo de saida >
    dependencies:
    - < Denominação da tarefa predecessora >
    - < Denominação da tarefa predecessora >
```

Figura 6.3: Registro do arquivo YAML contendo a descrição de uma tarefa

Os campos para a descrição de uma tarefa para o *framework* são:

1. `name`: Identificador da tarefa para o *framework*. As tarefas são diferenciadas umas das outras por meio do valor deste campo. Assim, o valor que o usuário atribui a este campo deve ser considerado com cuidado para evitar duplicatas.
2. `duration`: Uma estimativa para o tempo total de execução daquela tarefa.
3. `binary`: O tipo de binário a ser utilizado. No caso do Integrade os valores possíveis atualmente são Linux_i686 e Linux_x64.
4. `inputfile`: O arquivo de configuração para a submissão da tarefa, conforme o padrão Integrade
5. `outputfile`: O arquivo de saída da tarefa
6. `dependencies`: Uma lista de zero ou mais tarefas, indexadas pelo nome, predecessoras a esta. Ou seja, aquelas que precisam estar concluídas para que se possa dar início à execução desta tarefa.

Após receber o arquivo YAML com a configuração do *workflow*, o WSCT realiza o *parsing* deste arquivo e o converte em dois tipos de objetos serializáveis JAVA: cada tarefa é convertida em um objeto do tipo "Tarefa" com todos os dados descritos no arquivo. Ao final desse processo, o WSCT cria um outro objeto do tipo *workflow* que contém o conjunto de todas as tarefas a ele pertinentes e o submete ao Gerenciador de *Workflows* (WFM) pela rede através de RMI [5].

Registro das tarefas no ASCT

Um dos passos necessários à execução dos *workflows* é o registro das tarefas no ambiente de Grid.

No Integrade isto é realizado da seguinte forma:

1. Inicia-se o *middleware* e o ASCT - Figura 6.4
2. Realiza-se o *upload* do arquivo binário da tarefa - Figuras 6.5 e 6.6

3. Define-se os parâmetros (arquivos de entrada e saída, argumentos de linha de comando, tipo da aplicação e *constraints*) - Figura 6.7

4. Salva-se os dados utilizando o botão "Save". O nome do arquivo de configuração gerado deverá ser o valor do campo "inputfile" do arquivo YAML de definição do *workflow*.

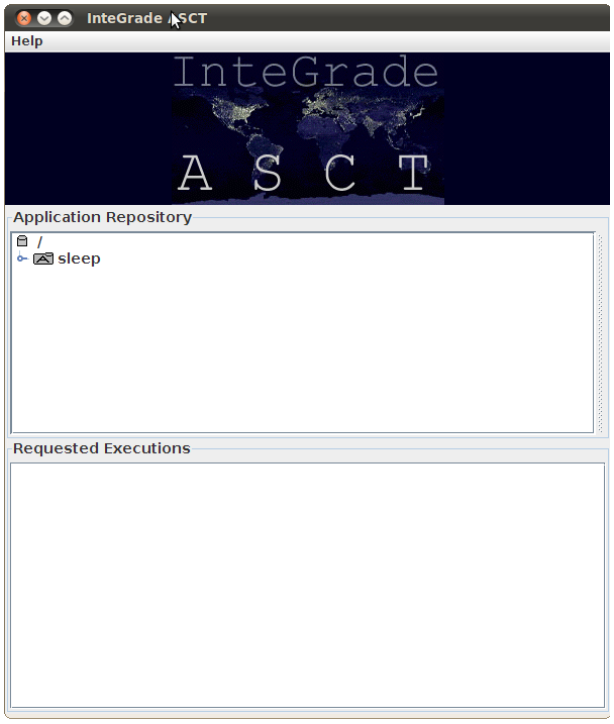

Figura 6.4: Tela inicial do ASCT

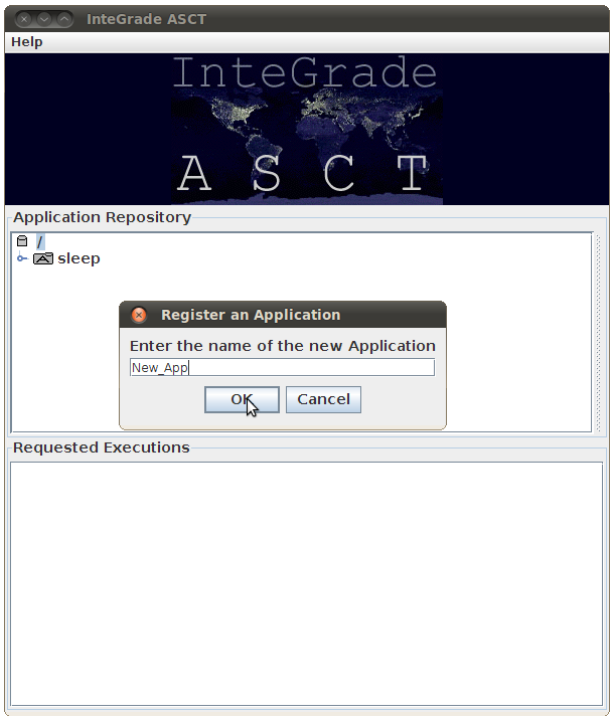

Figura 6.5: Tela de seleção do nome da aplicação

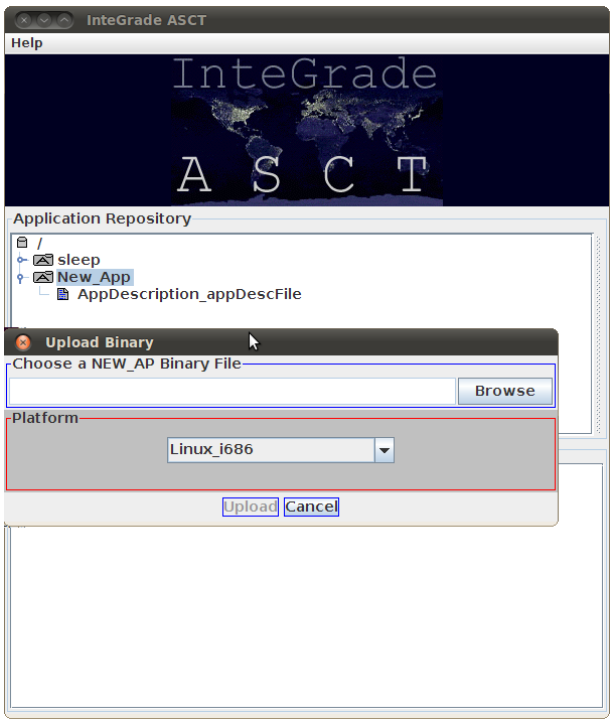

Figura 6.6: Tela para selecionar o arquivo binário da aplicação

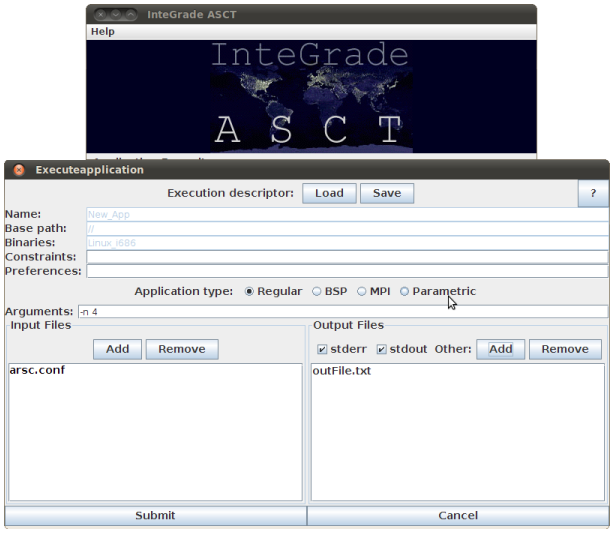

Figura 6.7: Tela para configuração e armazenamento dos parâmetros da aplicação

6.4.2 Workflow Manager - WFM

O WFM é responsável pelo gerenciamento dos *workflows* no *framework*. Cada vez que o WSCT submete um *workflow*, o WFM o recebe, adiciona-o à sua lista de *workflows* em execução e analisa quais de suas tarefas estão aptas a serem executadas. Para isso, ele varre as tarefas em busca daquelas cujas dependências já tenham sido satisfeitas e adiciona ao conjunto de tarefas prontas. Estas tarefas são então submetidas ao FTM que cuidará do processo de execução.

Ao receber notificação de término de alguma tarefa, o WFM reinicia o processo de busca de tarefas prontas e submissão ao FTM, garantindo assim o progresso do *workflow*. Uma vez que todas as tarefas tenham sido concluídas, o WFM dá o *workflow* por encerrado.

6.4.3 Fault Tolerance Manager - FTM

O FTM é o módulo central do *framework*. Seu papel é o de concentrar informações sobre os FTECs sendo executados e coordenar o processo de criação e encerramento dos mesmos. É ele quem solicita a definição da técnica de tolerância a falhas a ser utilizada logo antes de submeter uma tarefa para a execução.

Ao receber do WFM uma tarefa a ser executada, o FTM verifica que ela não foi submetida previamente, salva uma descrição da tarefa e o FTEC a ela associado, um identificador único e seu status de execução. Em seguida, solicita ao FTSM a definição de qual FTEC deverá ser utilizado e com que configuração para aquela tarefa. Por fim, o FTM adiciona aquele registro à sua listagem de FTECs em execução e solicita seu início.

Uma vez que tenha sido notificado da conclusão de uma tarefa, o FTM a marca como concluída.

6.4.4 Fault Tolerance Execution Coordinator - FTEC

Este módulo implementa o mecanismo de tolerância a falhas propriamente dito. Cada FTEC implementa uma técnica de tolerância a falhas específica, e a disponibiliza ao FTM para que possa ser utilizada. Para isto, cada FTEC implementa um servidor para registro da implementação no *registry* do RMI. Este servidor por sua vez implementa o método de criação de *threads* para submissão e acompanhamento de tarefas. Existe um FTEC para cada técnica de tolerância a falhas e uma *thread* de um FTEC para supervisionar a execução de cada tarefa.

Cada *thread* de um FTEC supervisiona a execução de exatamente uma tarefa e é capaz de submeter tarefas ao Grid, interagir com execuções em andamento e até mesmo cancelá-las, permitindo-lhes trabalhar a redundância em tempo e/ou espaço. A *thread* que supervisiona a execução de uma dada tarefa será também a responsável por julgar quando aquela tarefa deverá ser considerada terminada. Cada *thread* fica, portanto, livre para definir se qualquer resultado final de sua tarefa é válido, se serão necessários n resultados equivalentes ou se terá outra política qualquer para este fim.

Cada implementação distinta de FTEC é criada pela definição de uma nova subclasse da classe abstrata FTEC, e deve implementar as interfaces `Runnable` e `Serializable`. Atendendo a essas premissas, a implementação fica livre para ser definida e programada

pelo usuário. Por fim, cada FTEC implementa um provedor de serviços distinto identificado por meio de uma URL composta pela concatenação do valor 'rmi://localhost:1099/' e o identificador a ser utilizado pelo FTSM.

Três métodos se destacam pela sua criticidade:

- O método estático *requestFtecThread(configFile, submittedTask, chosenFtecService, uid)* encapsula o procedimento para obter o objeto remoto do FTEC e a invocação do método *startFtecThread* implementado pela classe FTEC selecionada. Recebe como parâmetros um arquivo de configurações, uma tarefa, o identificador do FTEC selecionado e um identificador da thread (uid, ou *unique identifier*) atribuído pelo FTM. Este método não precisa ser reimplementado em caso de novas técnicas de tolerância a falhas (TF).
- O método *startFtecThread(configFile, submittedTask, uid)* é o responsável pelo *parse* das configurações e criação de uma nova thread para submissão, acompanhamento e término da tarefa. Ele é do tipo *abstract* na classe genérica FTEC e deve ser implementado pelas subclasses de acordo com o seu padrão de funcionamento interno. Recebe como parâmetros um arquivo de configurações, uma tarefa e o identificador da thread (uid, ou *unique identifier*).
- O método *run()* que define o comportamento do FTEC uma vez que uma nova *thread* tenha sido solicitada.

Ainda assim, a implementação de um novo FTEC é simplificada pois a classe genérica FTEC já provê uma diversidade de métodos para tarefas comuns como submissão, acompanhamento e conclusão de tarefas para utilização caso o programador deseje utilizar procedimentos padrões.

Uma execução padrão é descrita na Figura 6.8. Inicialmente, o FTM invoca o método estático `requestFtecThread(configFile, submittedTask, chosenFtecService, uid)` que cria uma nova instância do FTEC solicitado. Este método é o responsável por invocar, no provedor de serviços apropriado, o método responsável por criar uma nova thread. Cada thread criada desta maneira é responsável pela execução de uma única tarefa utilizando a técnica de tolerância a falhas que ela implementa. As threads dos FTECs interagem diretamente com o Grid para submeter, cancelar e acompanhar o andamento da tarefa que lhe foi designada.

Uma observação importante é a de que, assim como o Integrade, os módulos de gerenciamento de tolerância a falhas e de *workflow* (WFM, FTM, FTSM e FTEC) são executados no nodo central (juntamente ao GRM) e enquanto o WSCT e as execuções submetidas pelos FTECs são executadas em máquinas provedoras de recursos.

6.4.5 Fault Tolerance Selection Mechanism - FTSM

Dado que o usuário terá a seu dispor vários FTECs para utilização, cada um implementando uma técnica de tolerância a falhas distinta, com uma potencialmente vasta gama de parametrizações para cada um deles, torna-se primordial um módulo para definição da escolha de um dos métodos e parametrizações disponíveis. É este, portanto, o papel deste módulo: definir a maneira como o FTEC mais adequado será selecionado e retornar o nome do serviço que provê a instanciação daquele FTEC.

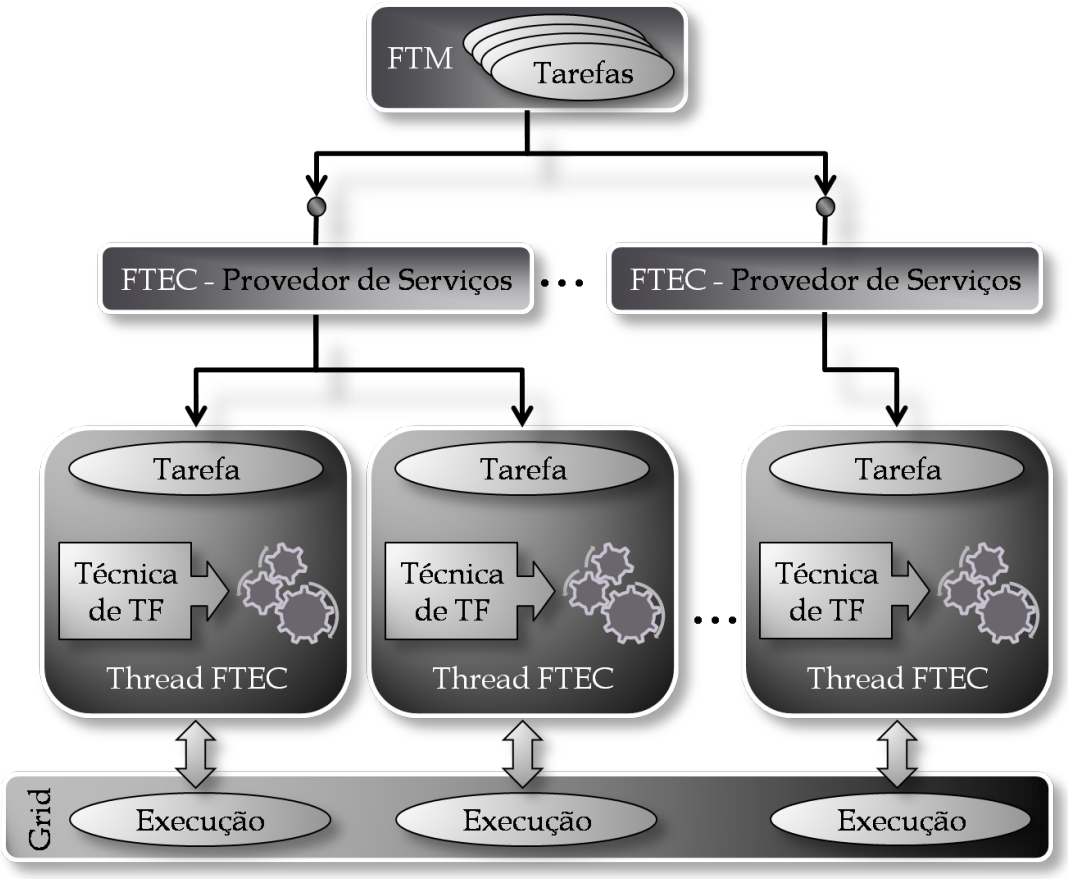

Figura 6.8: Arquitetura multi-threaded dos FTECs- cada FTEC implementa uma técnica de TF e cada *thread* é responsável por uma única tarefa

Inicialmente, há um mecanismo que recebe como entrada um arquivo com uma árvore binária de decisão. Uma possível árvore de decisão é ilustrada na Figura 6.9. Esta árvore de decisão é a maneira pela qual o usuário declara as regras para seleção do mecanismo de tolerância a falhas, levando em consideração a topologia corrente do Grid. Cada sequência de decisões gera um cenário para a topologia onde uma certa técnica de tolerância a falhas (TF) com dados parâmetros se torna a opção mais interessante para um dado usuário e *workflow* e deverá, portanto, ser a escolhida.

Estes cenários podem, por exemplo, englobar situações onde exista abundância de recursos aptos à realização da tarefa e que, possivelmente, uma abordagem que utilize a redundância ativa seja mais interessante e situações onde a escassez de recursos torna a mesma redundância ativa impraticável. Desta forma, o usuário pode definir as melhores técnicas sobre dadas circunstâncias.

Para definir esta árvore de decisão, cria-se um arquivo de descrição por meio da linguagem YAML. A árvore de decisão é composta por diversos nós intermediários que possuem uma condição e nós folha contendo a escolha do FTEC e arquivo de parâmetros selecionado pelo usuário para aquela situação específica.

Cada nó da árvore é representado no arquivo através de um registro YAML similar ao ilustrado na Figura 6.10. O mesmo *layout* de registro pode representar tanto um nó

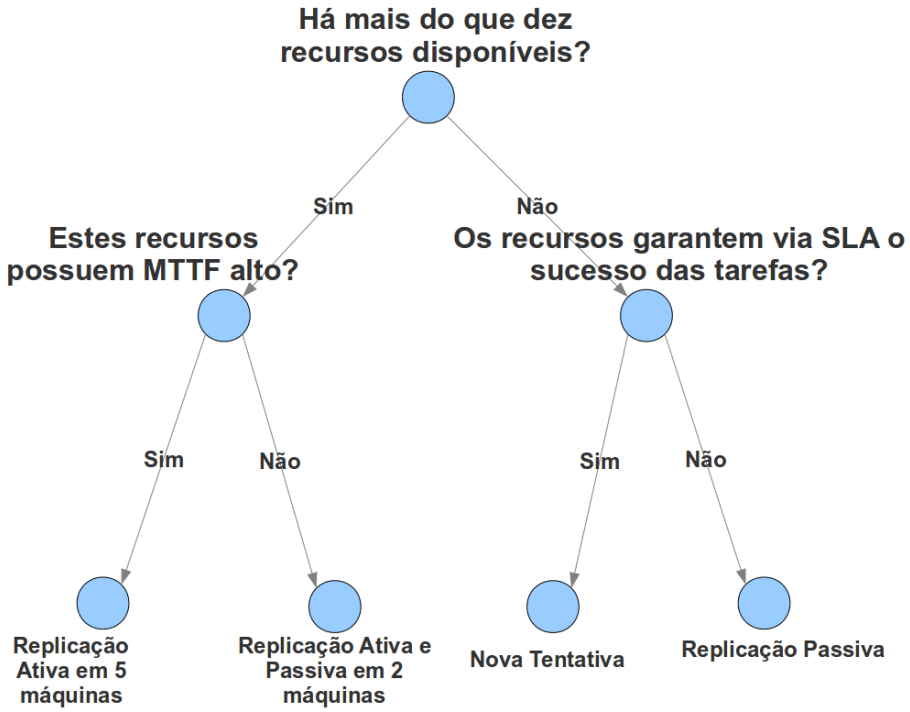

Figura 6.9: Um exemplo simplificado de árvore Binária de Execução

intermediário e a condição a ele associada ou um nó folha contendo o FTEC escolhido e o arquivo com os parâmetros adequados. A diferenciação entre os nós intermediários e os nós folha se dá pelos campos preenchidos em cada registro. Enquanto os registros dos nós intermediários preenchem os campos relativos à escolha representada (quantidade de nodos, atributo analisado, valor do atributo, tipo da comparação), os nós folha contém os valores FTEC escolhido e arquivo de configuração do FTEC. A Figura 6.10 apresenta o formato do arquivo de descrição da árvore binária de decisão e seus campos são descritos a seguir.

- - -

nodeId: *identificador do nodo*
parentNode: *Nodo Pai*
parentNodeAnswered: *Resposta dada pelo nodo pai*
nodeQuantity: *Quantidade de nodos com este atributo*
nodeVariable: *Atributo a ser considerado*
nodeValue: *Valor do atributo*
nodeComparison: *Tipo da comparação (equal, more, less)*
FtecChoice: *FTEC selecionado*
FtecConfigFile: *Arquivo de configuração a ser usado*

Figura 6.10: Registro do arquivo YAML descrevendo cada nó da árvore de decisão

1. `nodeId`: O identificador do nodo na árvore. É representado por um valor numérico, sendo o 0 reservado à raiz.
2. `parentNode`: O nodo pai, ou seja, o nodo do nível superior ao corrente que possui uma aresta conectando-o ao nó atual.
3. `parentNodeAnswered`: Sendo uma árvore binária de decisão, cada nodo pode responder sim ou não para a inquisição proposta. A resposta do pai deve ser colocada neste campo.
4. `nodeQuantity`: Esta é o primeiro campo referente à inquisição a ser feita ao Grid. Ele designa a quantidade de nodos que devem atender às especificações descritas nos campos nodeVariable, nodeComparison e nodeValue. Ele só deverá ser preenchido nos nodos raiz e intermediários.
5. `nodeVariable`: Este campo define qual o atributo dos recursos que será avaliado. Ele pode conter os valores *hostName* para seleção de uma máquina em particular; *osName* e/ou *osVersion* para indicar, respectivamente, o nome e a versão do sistema operacional; *processorName* e *processorMhz* que indicam o nome e a velocidade em MHz do processador; *totalRam*, a quantidade total de memória principal; *freeRam*, a quantidade total de memória principal livre; *totalSwap* e *freeSwap* que indicam a quantidade de disco para disponível para Swap, respectivamente total e disponível; *freeDiskSpace* para indicar o espaço livre em disco; ou *cpuUsage* que indica o grau de utilização da CPU.
6. `nodeValue`: Este campo define o valor numérico contra o qual o atributo descrito em `nodeVariable` será comparado.
7. `nodeComparison`: Este campo diz qual a comparação a ser realizada entre o valor expresso no campo `nodeValue` e o valor observado. Seus valores possíveis são *less*, *more* e *is*, significando que o valor observado nos nós do Grid deve ser, respectivamente, menor, maior ou igual ao observado no campo `nodeValue`.
8. `FtecChoice`: Este campo só é preenchido nos nós-folha da árvore. Ele descreve o identificador do FTEC que deve ser escolhido.
9. `FtecConfigFile`: Este campo também só deve ser preenchido nos nós-folha da árvore e descreve o arquivo com as configurações apropriadas para o FTEC considerando a sequência de perguntas realizadas pelos nós intermediários.

A partir da árvore definida pelo usuário nesse arquivo, o FTSM realiza o cruzamento dos dados da topologia com as regras ali estipuladas, realizando o percurso da árvore a partir da raiz até uma folha. Para cada nó intermediário, ou seja, que não possua os valores FtecChoice e FtecConfigFile preenchidos, ele realiza a avaliação descrita na Figura 6.11.

Como a árvore aqui representada é uma árvore binária, qualquer nó intermediário avaliado possui dois filhos, identificados no arquivo pelo campo `parentNode` contendo o identificador do nó avaliado. Um destes é o filho cuja aresta é o "sim", identificado pelo campo `parentNodeAnswered` com o valor "`yes`", e será o escolhido caso a avaliação seja positiva. Similarmente, o outro é o filho com aresta "não" e identificado pelo valor "no" no campo `parentNodeAnswered` e será escolhido caso a avaliação seja negativa.

Dados os recursos atualmente disponíveis no Grid, existem *n* nós *(nodeQuantity)* cujo atributo `X` *(nodeVariable)* seja `maior/menor/igual` *(nodeComparison)* ao valor `Y` *(nodeValue)*?

Figura 6.11: Decisão realizada em um nó intermediário

Este processo se repete até que se encontre um nó folha. Neste ponto, não haverá mais uma pergunta expressa no nó mas uma resposta. Os nós folhas definem no campo `FtecChoice` o nome do serviço de FTEC a ser utilizado e no campo `FtecConfigFile` o arquivo com as configurações apropriadas para aquele FTEC.

6.4.6 Inclusão de novos FTECs

Um dos objetivos do *framework* é o de permitir a inclusão posterior de novas técnicas de tolerância a falhas. Para isso, o *framework* trabalha com interfaces bem definidas e comunicação via RMI.

Cada técnica é identificada por um nome. Este nome define o nome do provedor de serviços que implementa a interface `FtecInterface`. Esta interface é composta de um único método, ilustrado na Figura 6.12.

```
public interface FtecInterface extends Remote{

    public Long startFtecThread(String configFile,
                        Task submittedTask, int uid)
    throws RemoteException;

}
```

Figura 6.12: Interface remota a ser implementada pelos FTECs

Este método recebe como entrada o arquivo genérico de parâmetros de configuração do FTEC, um objeto Java descrevendo a tarefa e um identificador para o controle do FTM. Ele será responsável pela criação de um mecanismo para submissão da tarefa ao Grid, acompanhamento de sua execução e encerramento da execução da tarefa notificando ao FTM e WFM sua conclusão.

Haverá também, conforme a arquitetura RMI, o provedor de serviços cuja função é instanciar a classe responsável pela implementação e registrá-la no *registry* de forma que a interface seja localizável. A título de exemplo, o código para a implementação de um *provedor de serviços* para o FTEC implementado na classe `RetryFTEC` e cujo identificador seja "Retry" é ilustrado na Figura 6.13.

O endereço URL ao qual o serviço é associado é composto pela concatenação do endereço básico "rmi://localhost:1099/" e o identificador apropriado daquele FTEC que neste exemplo é "Retry". Este identificador foi também incluído na árvore de decisões do FTSM de forma que este saiba que, de acordo com os parâmetros fornecidos pelo usuário, esta será a melhor escolha para a topologia atual do Grid. Assim, todas as vezes que o

```
public RetryFTECServer() throws InterruptedException {
   try {
         LocateRegistry.createRegistry(1099);
   } catch (RemoteException e1) {
         System.out.println("Registry not created.
                     Perhaps previously created.");
   }

  try {
        ftec = new RetryFTEC();
        Naming.rebind("rmi://localhost:1099/Retry", ftec);
  } catch (Exception e) {
        System.out.println("Error associating Registry");
        System.out.println("Trouble: " + e);
  }
}

public static void main(String[] args)
 throws RemoteException, InterruptedException {
     new RetryFTECServer();
}
```

Figura 6.13: Exemplo: Servidor RMI do FTEC "Retry"

FTSM decidir que, por exemplo, o FTEC de identificador "RepPas" deva ser utilizado, o FTM terá certeza de que ele poderá ser encontrado no endereço "rmi://localhost/RepPas".

O código para a execução do FTEC em si fica encapsulado na classe implementada, sendo que esta implementação tem como requisitos estender a classe FTEC e implementar as interfaces Runnable e Serializable.

Assim, o protótipo básico da classe Retry seria:

```
public class RetryFTEC extends Ftec implements Runnable, Serializable
```

Cada FTEC deverá então implementar seu mecanismo de tolerância a falhas, interagindo com o Grid da forma que mais lhe convir. Todos os FTECs herdarão mecanismos padronizados para submeter, acompanhar e encerrar tarefas no Grid.

Dos métodos abstratos a serem implementados, os mais importantes são:

Long startFtecThread: Este é um método de visibilidade pública pré-definido na classe FTEC porém que deverá ser implementado em cada novo tipo de FTEC. Seu protótipo é ilustrado na Figura 6.14

Este é o método principal de todo FTEC. Como já explicado na Subseção 6.4.4, é ele quem irá criar a thread do FTEC para o acompanhamento de uma dada tarefa.

Uma observação importante é a de que, como explicado nas decisões de projeto (Seção 6.1), apesar de ser obrigatória a presença do arquivo de configuração, o conteúdo desse arquivo dependerá apenas da implementação do FTEC apropriado, que é o responsável

```
public abstract Long
   startFtecThread(String configFile,Task submittedTask,int uid);
```

Figura 6.14: Protótipo da função startFtecThread

pelo *parse* das informações ali contidas. Assim, um FTEC pode optar por utilizar um arquivo de configuração com os dados em XML, outro pode ter uma formatação própria com um parâmetro em cada linha e um terceiro pode nem mesmo ter configurações a utilizar, recebendo portanto um arquivo vazio.

6.5 Inclusão de Novos Mecanismos de Seleção de Técnicas de Tolerância a Falhas

Outra característica importante de ser ressaltada do *framework* é a possibilidade de se desenvolver um FTSM totalmente novo, no caso do mecanismo *default* com percurso de árvores não atender às necessidades do usuário ou de um *workflow* em particular.

O FTSM é implementado por uma classe Java cujo único método necessariamente implementado é o método da Figura 6.15

```
public static String getFtecName(String inputParametersFile)
     throws IOException {
           return FtecName;
     }
```

Figura 6.15: Protótipo da função getFtecName

Esse é o método que é invocado pelo FTM para a definição do FTEC a ser utilizado. Ele recebe uma *String* com o caminho para um arquivo de parâmetros e retorna uma outra *String* com o identificador do FTEC a ser utilizado pela tarefa. Por *default*, o nome do arquivo de parâmetros se chama `inputParametersFTSM.txt` e este nome deve ser preservado.

Para a inclusão de um mecanismo FTSM distinto basta que o usuário reimplemente esta função como desejar.

A título de exemplo, para realizar os testes sem a sobrecarga do FTSM com percurso da árvore de decisão foi implementado um novo mecanismo para o FTSM que sempre retornava o valor "null" como o FTEC escolhido.

Para implementá-lo, bastou definir a classe FTSM como ilustrado na Figura 6.16.

6.6 Exemplo de Execução

Suponha que um usuário possua o *workflow* descrito na Figura 6.17 para ser executado. Primeiramente, ele descreve este *workflow*, por meio de um arquivo YAML similar ao da Figura 6.18.

```
package ftsm;
import java.io.*;

public class FTSM {
    public static String getFtecName(String inputParametersFile)
      throws IOException {
       return "NULL";
     }
}
```

Figura 6.16: Código fonte do FTEC Null

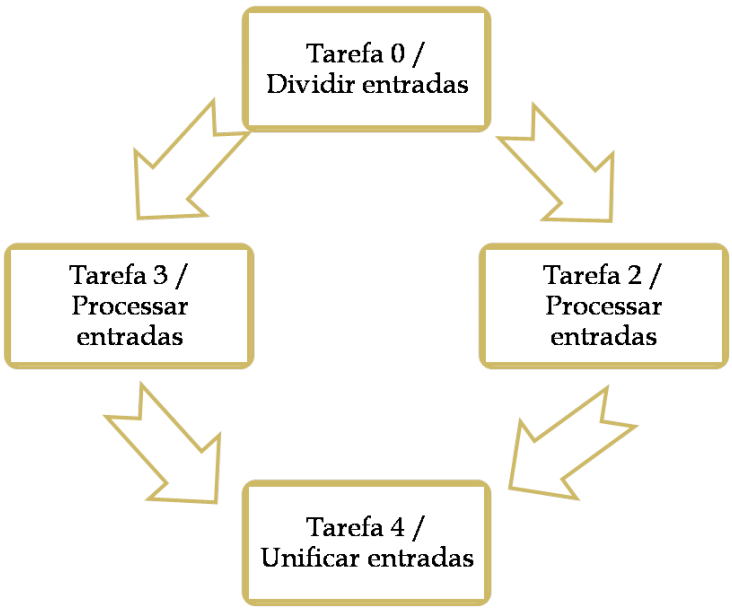

Figura 6.17: Exemplo de *workflow* do usuário

Em seguida, ele define uma árvore binária de decisão que selecione os FTECs que ele considera mais interessantes para cada cenário da topologia do Grid. Neste exemplo, a árvore de decisão criada pelo usuário é a representada na Figura 6.19 e descrita por meio do arquivo apresentado na Figura 6.20. Nessa árvore de decisão, o usuário explicita que, caso haja mais de 3 máquinas com 1 GB ou mais de memória RAM, deve ser utilizado o FTEC "Replicação Ativa" com as configurações expressas no arquivo " 3_maquinas.xml". Caso contrário, deve ser utilizado o FTEC "Nova Tentativa" com as configurações expressas em "5_vezes.txt".

A raiz é, por definição, o nó de identificador 0 (zero) e a partir dele se dará o percurso.

Uma vez que o usuário tenha criado os arquivos desta maneira, ele o submete ao *framework* por meio do WSCT. O WSCT realiza o parse do arquivo descritor do *workflow* convertendo-o em um objeto Java com a descrição de todas as tarefas. Esse objeto é submetido ao WFM que irá analisá-lo e, dadas as dependências apresentadas, concluir que apenas a tarefa 0 está apta a ser executada. Ela é então repassada ao FTM que solicita ao FTSM o FTEC apropriado. Suponhamos que o Grid neste instante possua

```
- - -
    name: 0
    duration: 15
    binary: Linux_i686
    inputfile: divide_input
    dependencies:
- - -
    name : 1
    duration : 60
    binary: Linux_i686
    inputfile: process_input1
    dependencies:
    - 0
- - -
    name: 2
    duration : 60
    binary: Linux_i686
    inputfile: process_input2
    dependencies:
    - 0
- - -
    name : 3
    duration : 15
    binary: Linux_i686
    inputfile: unite_input
    dependencies:
    - 1
    - 2
```

Figura 6.18: Arquivo YAML descrevendo o *workflow* da Figura 6.17

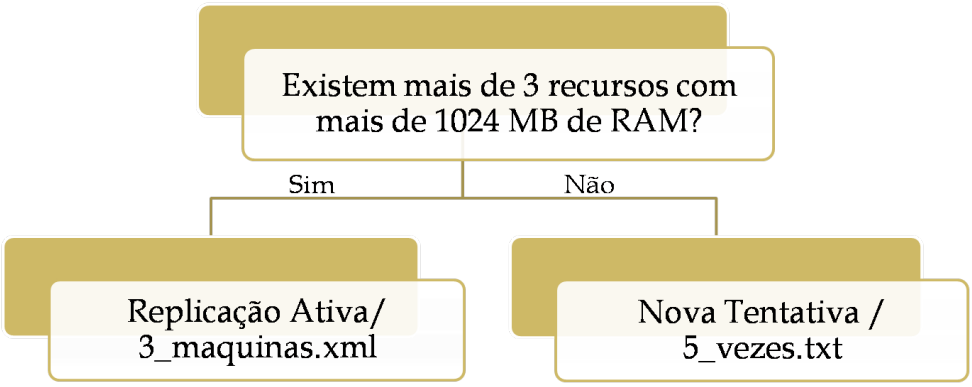

Figura 6.19: Exemplo de árvore binária de decisão

apenas duas máquinas com mais de 1024MB de RAM total. Logo, o FTSM irá indicar o FTEC "Retry" como o mais apropriado, com os parâmetros descritos no arquivo de configurações "5_vezes.txt".

O FTM irá armazenar aqueles dados e solicitar ao servidor de FTECs no endereço "rmi://localhost:1099/Retry" que providencie a execução da tarefa 0. O servidor irá criar uma *Thread* Java para acompanhar a tarefa e iniciá-la. Ela irá submeter a tarefa ao Grid, acompanhar sua execução e notificar o FTM quando de sua conclusão utilizando seu mecanismo interno para prover a técnica de TF que implementa. Uma vez completa, a *thread* FTEC notifica o FTM, que dá a tarefa por completa, e ao WFM que, tendo recebido a notificação reinicia a avaliação do *workflow*.

Neste momento o WFM verifica que tanto a tarefa 1 quanto a 2 podem ser realizadas e as dispara ao FTM. Similarmente ao que se deu com a tarefa 0, ele solicita ao FTSM o FTEC apropriado. Suponhamos aqui que outras 7 máquinas com mais de 1GB de RAM ingressaram no Grid, tendo ao todo 9 máquinas atendendo ao critério da árvore de decisão. Neste caso, o FTSM irá indicar o FTEC "Ativa" com o arquivo de configuração "3_maquinas.xml".

O FTM solicita ao servidor do FTEC no endereço "rmi://localhost:1099/Ativa" a execução das tarefas. Um FTEC irá então ser criado e supervisionará o processo de execução, notificando o FTM e WFM quando a mesma tiver sido concluída.

Mais uma reavaliação do *workflow* e a tarefa 3 é submetida. Como não ocorreram mudanças nos recursos do Grid, o FTEC "Ativa" foi novamente selecionado e a tarefa 3 foi submetida ao FTM, executada pelo FTEC e notificada a conclusão ao WFM que novamente reavalia o *workflow* e já pode considerá-lo completo.

```
- - -
    nodeId: 0
    parentNode:
    parentNodeAnswered:
    nodeQuantity: 3
    nodeVariable: totalRam
    nodeValue: 1024
    nodeComparison: more
    FtecChoice:
    FtecConfigFile:
- - -
    nodeId: 1
    parentNode: 0
    parentNodeAnswered:no
    nodeQuantity:
    nodeVariable:
    nodeValue:
    nodeComparison:
    FtecChoice: Retry
    FtecConfigFile: 5_vezes.txt
- - -
    nodeId: 1
    parentNode: 0
    parentNodeAnswered: yes
    nodeQuantity:
    nodeVariable:
    nodeValue:
    nodeComparison:
    FtecChoice: Ativa
    FtecConfigFile: 3_maquinas.xml
```

Figura 6.20: Arquivo YAML a árvore de decisão da Figura 6.19

Capítulo 7

Resultados Experimentais

7.1 Configuração das Máquinas

Para a execução dos testes foi criado no Laboratório de Pós-Graduação (LabPos) do Departamento de Ciências da Computação da UnB, um Grid composto por 5 máquinas denominadas `nodo 1` a `nodo 5`. O `nodo 1` exerce o papel de gerenciador global de recursos (GRM) e de provedor de serviço (LRM) enquanto os demais nodos (`nodos 2` a `5`) exercem apenas o papel de provedores de serviço (LRM). As configurações de hardware das máquinas estão descritas na Tabela 7.1.

Em termos de software, o `nodo 1` era equipado com Ubuntu Linux 9.10 Kernel 2.6.32-22-generic #36-Ubuntu SMP e utilizava os softwares OpenSSH 1.5.3, Sun Java JDK 6.20 e Integrade versão 0.5.2. Já os `nodos 2` a `5` utilizavam o Debian Linux Kernel 2.6.26-1-686 SMP e os softwares OpenSSH 1.5.1, Sun Java JDK 6.12.1 e Integrade versão 0.5.2.

Máquina	**Processador**	**Clock Processador**	**Memória RAM**
Nodo 1	AMD Sempron 2400+	1650 MHz	2 GB
Nodo 2	AMD Sempron 3000+	1800 MHz	2 GB
Nodo 3	AMD Sempron 2400+	1650 MHz	2 GB
Nodo 4	AMD Sempron 2400+	1650 MHz	2 GB
Nodo 5	AMD Sempron	1330 MHz	2 GB

Tabela 7.1: Tabela de Configurações de Hardware dos nós utilizados no Grid

7.2 Workflows para Testes

Para a realização dos testes foram utilizados três *workflows* distintos descritos sucintamente nas seções que se seguem.

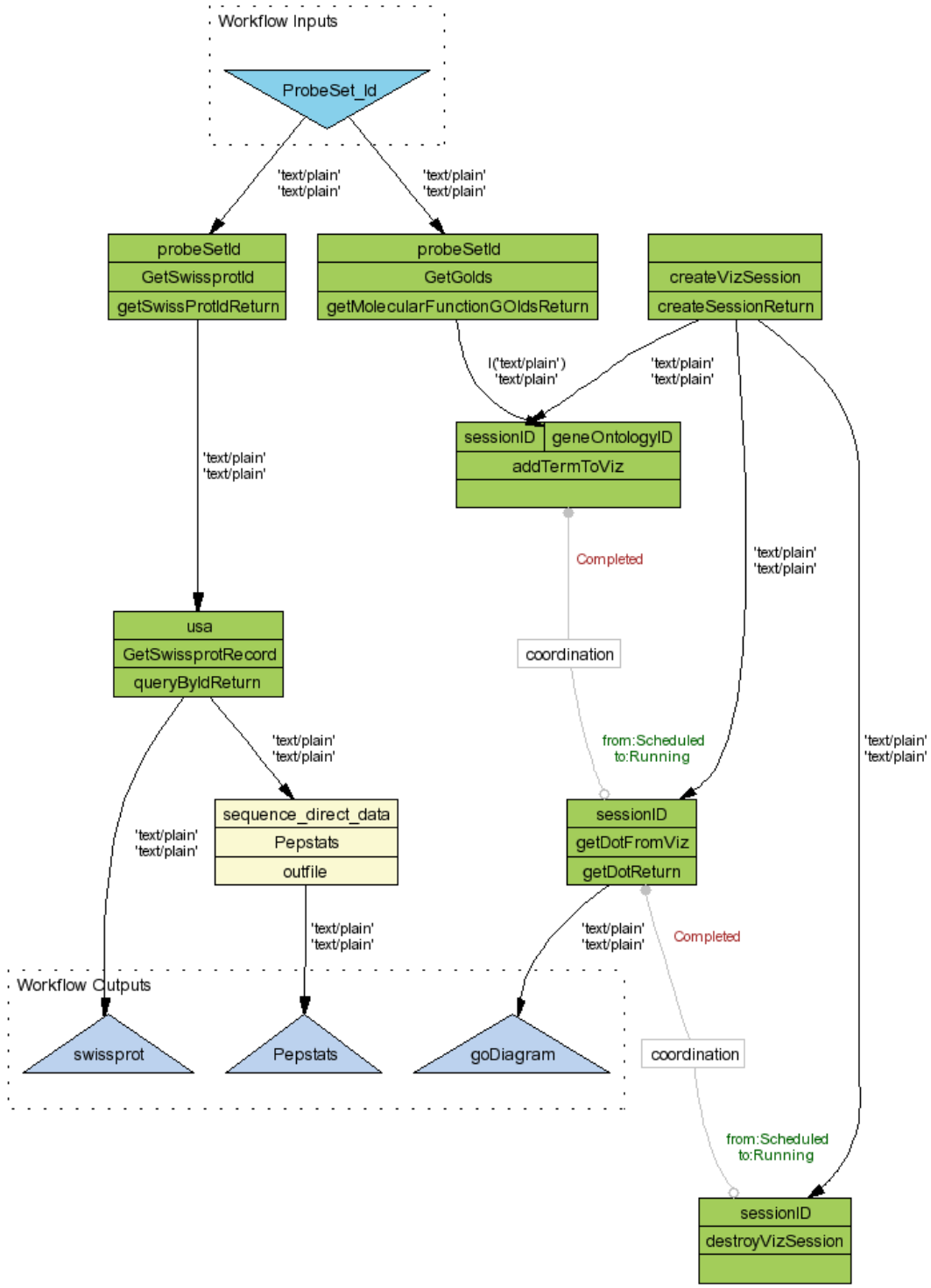

Figura 7.1: *Workflow* `TavernaBio` - [37]

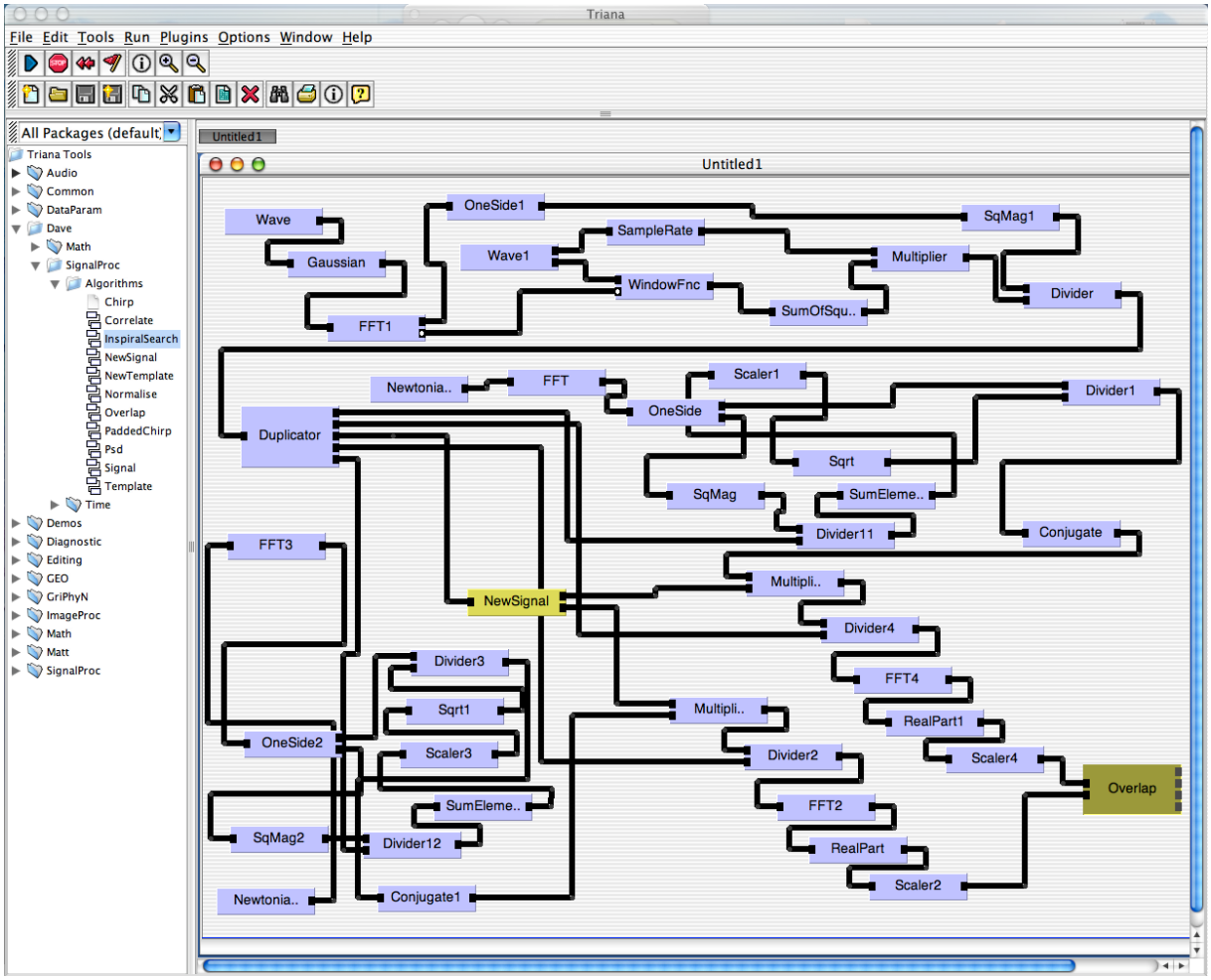

Figura 7.2: *Workflow* Triana completo, apresentado em [9].

7.2.1 Workflow: Taverna Bio

O primeiro *workflow* escolhido foi apresentado no trabalho de [37], está ilustrado na Figura 7.1 e será denominado "`TavernaBio`". Este é um *workflow* simples, o que faz dele ideal para um caso inicial de testes.

O *workflow* `TavernaBio` é um *workflow* de bioinformática que exemplifica um modelo *Scufl* particular de um estudo de caso da Síndrome de Graves [37]. Apresenta uma entrada, oito pontos de processamento e três saídas. Os pontos de processamento estão rotulados com as portas de entrada, nome do processo e portas de saída. Representa um *workflow* mais simples porém com múltiplas tarefas iniciais. Foi selecionado para testes em *workflows* mais simples e com menos tarefas.

7.2.2 Workflow: Triana Parcial

O segundo é uma fração do *workflow* ilustrado na Figura 7.2 e apresentado em [9]. Este *workflow* será denominado "`Triana Parcial`" e pode ser visualizado na Figura 7.3.

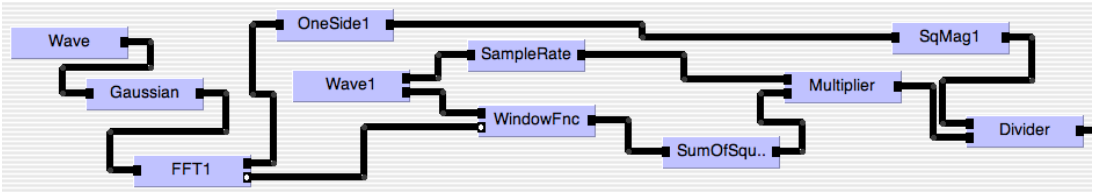

Figura 7.3: *Workflow* `Triana Parcial` - Fração do *workflow* da Figura 7.2

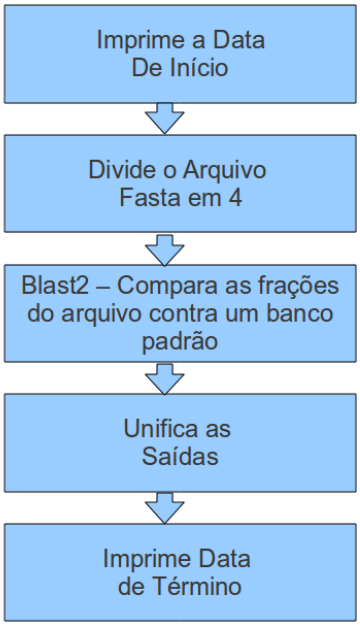

Figura 7.4: *Workflow* `EmbrapaBio` - Aplicação real criada a partir de interações com pesquisadores

Este *workflow* descreve um processo de processamento de sinais [9]. Ele possui uma complexa rede de relacionamentos, grande quantidade de tarefas e um bom nível de paralelismo e por isso foi escolhido para a bateria de testes.

7.2.3 Workflow: Embrapa Bio

Finalmente, o *workflow* `EmbrapaBio` (Figura 7.4) foi resultado de interações com pesquisadores de bioinformática e genômica comparativa da Embrapa. Ele representa um *workflow* real para alinhamento genético em ambientes de *Grid*. O *workflow* tem como entrada dois arquivos FASTA de sequências de nucletídeos. Destes arquivos, um será escolhido para fragmentação de forma que se possa distribuir as frações para diversos nodos realizarem as comparações contra o segundo arquivo FASTA. Os resultados de cada um dos nodos

são coletados e, ao final, uma última tarefa os aglutina em um único arquivo de saída que é o que permitirá aos pesquisadores identificar os melhores alinhamentos encontrados.

Este *workflow* é o composto de tarefas reais e não de tarefas *dummy* como os demais. Por esse motivo a medida de complexidade do *workflow* e o tempo de execução depende em grande parte dos arquivos de entrada e do desempenho da máquina na qual estão sendo executados.

Para a realização de nossos testes, obtivemos do *site* ftp do *National Center for Biotechnology Information* (NCBI) dois arquivos FASTA tradicionais: o `mito.nt` e o `yeast.nt`, que contém, respectivamente, os genes identificados das mitocôndrias e da levedura (*Saccharomyces cerevisiae*). O arquivo `yeast.nt` possui 17 sequências genéticas com tamanho médio de 715115 bases nitrogenadas e o arquivo `mito.nt` possui 1170 sequências genéticas com tamanho médio de 104833 bases nitrogenadas, sendo dividido pela segunda tarefa em 4 arquivos de aproximadamente 292 sequências.

7.3 Resultados obtidos

Para efeito de comparação e da medida de sobrecarga no sistema devida ao *framework* foram criados quatro cenários distintos:

1. *Workflows* com tarefas *dummy* rodando no Grid Completo utilizando FTEC Retry
2. *Workflow* com tarefas reais rodando no Grid Completo utilizando FTEC Retry
3. *Workflows* com tarefas reais rodando no Grid com 2 máquinas utilizando FTEC Retry
4. *Workflow* `TavernaBio` com tarefas *dummy* rodando no Grid Completo utilizando FTEC Active
5. *Workflow* `TavernaBio` rodando no Grid Completo utilizando FTEC Retry, ocorrendo erro na execução

Os cenários criados possuem diferentes objetivos. Os três primeiros cenários visam a avaliação da sobrecarga agregada pelo *framework* na execução dos *workflows*. O quarto e quinto cenários, na realidade, buscam fornecer medidas de desempenho e também servir como provas de conceito. O quarto cenário compara o desempenho obtido com FTECs distintos e demonstra que é possível incluir outras técnicas de TF no *framework* apenas implementando o novo FTEC, isto é, sem alterações no código de outros módulos. Por fim, o quinto cenário busca mostrar que a tolerância a falhas é de fato implementada através da utilização do *framework*.

Para os três primeiros cenários, foram realizadas duas baterias de teste, sem a ocorrência de falhas durante a execução:

Primeira Bateria: Na primeira bateria de testes mediu-se o tempo total de execução (*wallclock time*) para os *workflows* sem utilizar seleção dinâmica ou técnicas de tolerância a falhas, de forma a retirar toda a sobrecarga de execução e estipular o tempo padrão de execução dos *workflows* no Grid. Para isto, o processo de seleção de técnicas foi desativado e os FTECs utilizados eram nulos, *ie*, não implementavam qualquer técnica de TF.

Segunda Bateria: Na segunda bateria de testes, os mesmos *workflows* foram executados de forma que para cada tarefa o *framework* escolhesse a técnica de TF mais apropriada conforme as regras definidas. Em seguida, o gerenciador de execução tolerante a falhas escolhido prosseguia com a execução tolerante a falhas conforme sua programação interna. Para esta bateria de testes foi utilizado o sistema de seleção de técnicas de árvore de decisão apresentado na Figura 7.5 e um FTEC que implementava a técnica de nova tentativa.

7.3.1 Métodos de Seleção de Técnicas de Tolerância a Falhas

Como *default* foi definido o mecanismo de árvore binária de decisão descrito na Seção 6.4.5. Este foi o mecanismo utilizado pelo *framework* em todos os testes realizados. No entanto, outros mecanismos podem ser implementados, como apresentado na Seção 6.5.

Mecanismo de Seleção *Default* - Árvore de Decisão Nos três cenários, utilizamos a arvore de decisões ilustrada na Figura 7.5 simplesmente como forma de mostrar que esta pode ser definida pelo usuário. Em todos os cenários, a árvore utilizada foi a mesma, exceto pelo quarto cenário, quando apenas as folhas da árvore foram alteradas para que selecionassem o FTEC "active" ao invés do FTEC "Retry".

É importante ressaltar que a árvore de decisão criada é deliberadamente simples. Ela foi concebida de forma a realizar sempre a mesma opção de FTEC evitando a ocorrência de variações no tempo de execução em função da escolha de diferentes FTECs e garantindo a homogeneidade nos testes.

A avaliação do desempenho do algoritmo de percurso da árvore de decisão será realizada na Seção 7.4.

Mecanismo de Seleção Alternativo - "Null" Para as baterias de teste dos três primeiros cenários nas quais mediu-se o desempenho básico sem a seleção dinâmica e sem tolerância a falhas, utilizou-se o mecanismo "null" descrito na Seção 6.5. Este mecanismo, apesar da simplicidade, mostra que o framework permite a utilização de mecanismos distintos do *default*.

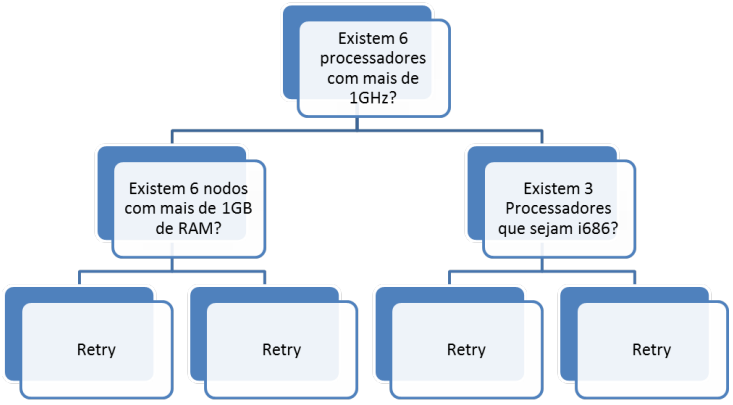

Figura 7.5: Árvore de Decisões utilizada nos experimentos

7.3.2 Cenário 1: *Workflows* com tarefas *dummy* rodando no Grid Completo utilizando FTEC Retry

No primeiro cenário, foram executados os *workflows* `TrianaParcial` e `TavernaBio` no ambiente de Grid completo com tarefas *dummy* com duração de 2 minutos cada uma.

Nesta configuração, pode-se definir o tempo mínimo de execução de cada *workflow* de acordo com as relações de dependendência entre suas atividades.

Avaliando-se o *workflow* `TavernaBio` podemos concluir que este levaria no mínimo 8 minutos para ser concluído, pois precisaríamos de:

1. 2 minutos para as tarefas sem precedência (*getSwissProtId, getGoIds, createVizSession*);
2. 2 minutos para as que dependem das iniciais (*getSwissProtRecord, addTermToViz*);
3. 2 minutos para a tarefa *getDotFromViz* que depende da *addTermToViz* e;
4. 2 minutos para a tarefa *destroyVizSession* que depende da tarefa *addTermToViz*

Similarmente, pode-se verificar que o *workflow* `TrianaParcial` demorará no mínimo 14 minutos.

Para implementar este *workflow* utilizamos o binário do comando nativo do linux "*sleep*"para atuar como tarefa *dummy*. Desta forma pode-se garantir que as tarefas possuam uma duração bem próxima da definida (120 segundos) independentemente da carga do nodo executor. Com isto, foi possível manter o foco na sobrecarga consequente do uso do *framework* de seleção dinâmica de técnicas de tolerância a falhas.

Os resultados das baterias de teste para cada um dos *workflows* encontram-se descritos nas Tabelas 7.2 e 7.3.

Tarefa	Tempo de Execução sem TF	Tempo de Execução com TF
1° teste	8'21" (501 segundos)	8'31" (510 segundos)
2° teste	8'15" (495 segundos)	8'24" (504 segundos)
3° teste	8'18" (498 segundos)	8'29" (509 segundos)
Média	**8'18" (498 segundos)**	**8'28" (508 segundos)**

Tabela 7.2: Tempos de Execução obtidos com e sem a utilização das técnicas de TF e da seleção dinâmica das mesmas para o *workflow* `TavernaBio`

Tarefa	Tempo Médio de Execução sem TF	Tempo Médio de Execução com TF
1° teste	14'29" (869 segundos)	14'38" (878 segundos)
2° teste	14'31" (871 segundos)	14'39" (879 segundos)
3° teste	14'28" (868 segundos)	14'48" (888 segundos)
Média	**14'29"' (869 segundos)**	**14'41" (881 segundos)**

Tabela 7.3: Tempos de Execução obtidos com e sem a utilização das técnicas de TF e da seleção dinâmica das mesmas para o *workflow* `TrianaParcial`

Resultados: *Workflows* TrianaParcial e TavernaBio implementados com tarefas *dummy* rodando no Grid completo Em nosso primeiro cenário de testes, a diferença entre o tempo de execução sem seleção dinâmica e sem tolerância a falhas foi, em média, de 10 segundos para o *workflow* TavernaBio e de 12 segundos para o `TrianaParcial`. Quando comparamos este valor com o tempo total de execução podemos concluir que o overhead de 1.96% para o `TavernaBio` e de 1.26% para o TrianaParcial é muito pequeno senão desprezível para os *workflows* utilizados.

7.3.3 Cenário 2: *Workflow* com tarefas reais rodando no Grid Completo utilizando FTEC Retry

No segundo cenário, o *workflow* `EmbrapaBio` foi executado no Grid completo. Neste cenário, todas as máquinas estavam disponíveis e o gerenciador global de recursos (GRM) do Integrade realizava a alocação dos recursos conforme sua política. Os resultados obtidos estão descritos na Tabela 7.4.

Tarefa	**Tempo de Execução sem TF**	**Tempo de Execução com TF**
`1° teste`	22'15" (1335 segundos)	23'23" (1403 segundos)
`2° teste`	8'16" (496 segundos)	16'36" (996 segundos)
`3° teste`	12'20" (740 segundos)	18'27" (1107 segundos)
`Média`	**14'17"' (857 segundos)**	**19'28" (1168 segundos)**
`Desvio Padrão`	**431.56**	**210.39**

Tabela 7.4: Tempos de Execução obtidos com e sem a utilização das técnicas de TF e da seleção dinâmica das mesmas para o *workflow* `EmbrapaBio` em Grid com 5 máquinas distintas.

Resultados: *Workflow* EmbrapaBio implementado com tarefas reais rodando no Grid completo Este cenário descreve um *workflow* com tarefas reais e não *dummy* como as do primeiro cenário. Por este motivo as tarefas de processamento possuem uma grande demanda por CPU o que acarretou uma variação além do aceitável nos resultados, com o desvio padrão chegando a 41% do valor da média de uma bateria de testes. Esta variação se deve à diferença de desempenho entre as máquinas alocadas pelo GRM para realizar o processamento dos dados.

Assim sendo, apesar de ser um teste válido, as baterias de teste do segundo cenário não contribuíram com dados úteis para a avaliação do desempenho do *framework*.

7.3.4 Cenário 3: *Workflow* com tarefas reais rodando no Grid com 2 máquinas utilizando FTEC Retry

Por se tratar de aplicação real com intenso uso de CPU e memória, o observado no segundo cenário foi uma variação muito grande nos resultados, tanto entre os testes de baterias distintas quanto entre testes da mesma bateria. Desta forma, os dados da Tabela 7.4 não representam uma medida confiável para a avaliação do *framework*.

Exatamente em função desta alta variação nos valores criou-se o terceiro cenário. Neste novo cenário retiramos a variação do desempenho dos recursos alocados mantendo apenas 2 máquinas no Grid, uma com as funções do GRM e outra provendo recursos através de um LRM. Isto permitiu-nos obter medidas mais coerentes com o que nos propomos a avaliar.

Como esperado, os resultados desta bateria de teste foram muito mais uniformes e encontram-se descritos na Tabela 7.5.

Resultados: *Workflow* EmbrapaBio implementado com tarefas reais rodando no Grid com 2 máquinas

Tarefa	**Tempo de Execução sem TF**	**Tempo de Execução com TF**
1° teste	14'49" (889 segundos)	14'58" (898 segundos)
2° teste	14'53" (893 segundos)	14'59" (899 segundos)
3° teste	14'54" (894 segundos)	14'59" (899 segundos)
Média	**14'52"' (892 segundos)**	**14'59" (899 segundos)**

Tabela 7.5: Tempos de Execução obtidos com e sem a utilização das técnicas de TF e da seleção dinâmica das mesmas para o *workflow* `EmbrapaBio` em Grid apenas com uma máquina com o GRM e outra com o LRM

Como era esperado, deixando apenas um LRM ativo conseguiu-se eliminar a fonte da variação de desempenho e pôde-se obter dados significativos da execução de um *workflow* real.

O que se observa agora é que neste cenário os resultados foram ainda melhores que os do primeiro, com o *framework* incluindo um overhead de, em média, apenas 7 segundos ou 0.77% ao tempo total de execução.

7.3.5 Cenário 4: *Workflow* `TavernaBio` com tarefas *dummy* rodando no Grid Completo utilizando FTEC Active

Neste cenário buscamos realizar a análise de como a alteração do FTEC pode alterar o tempo de execução do *workflow*, bem como demonstrar que, em conformidade ao que está dito no Capítulo 6, a alteração e inclusão de um novo FTEC é viável sob a arquitetura criada e apresentada neste documento.

Para este cenário, criamos um FTEC nomeado "Active" que implementa a replicação ativa e retorna o valor encontrado pela primeira instância encerrada com sucesso.TavernaBio

Os resultados obtidos neste cenário estão descritos na Tabela 7.6, juntamente com os resultados deste mesmo *workflow* utilizando o FTEC `Retry` apresentados previamente na Tabela 7.2 da Seção 7.3.2.

A comparação entre os resultados obtidos nos dá mais um argumento para ressaltarmos que nenhuma técnica de tolerância a falhas deve ser considerada ideal para qualquer caso. Diferentemente do que foi percebido no trabalho [13], nas nossas baterias de testes o `retry` se revelou o mais eficiente que o `Active`.

Notoriamente, isto se deve ao fato que em comparamos dois cenários que possuíam a premissa de serem livres de falhas. Porém isto já pode ser levado em consideração, criando regras que remetam ao FTEC `Retry` para ambientes de alta disponibilidade ou para o `Active` em ambientes de menor disponibilidade.

Resultados: *Workflow* `TavernaBio` implementado com tarefas *dummy* rodando no Grid completo utilizando FTEC Active

Tarefa	**Tempo de Execução com FTEC Active**	**Tempo de Execução com FTEC Retry**
`1° teste`	8'54" (534 segundos)	8'31" (511 segundos)
`2° teste`	8'53" (533 segundos)	8'24" (504 segundos)
`3° teste`	8'51" (531 segundos)	8'29" (509 segundos)
`Média`	**8'52" (532 segundos)**	8'31" (508 segundos)

Tabela 7.6: Tempos de Execução obtidos com a utilização das técnicas de TF Active e Retry para o *workflow* `TavernaBio` no Grid completo

Tarefa	Operação	Horário
createVizSession	submetido	06:45:12
getGoIDs	submetido	06:45:17
getSwissProtId	submetido	06:45:22
createVizSession	erro; resubmissão	06:45:33
getGoIDs	erro; resubmissão	06:45:38
getSwissProtId	erro; resubmissão	06:45:43
createVizSession	erro; resubmissão	06:46:15
getGoIDs	erro; resubmissão	06:46:20
getSwissProtId	erro; resubmissão	06:46:25
createVizSession	erro; resubmissão	06:46:57
getGoIDs	erro; resubmissão	06:47:02
getSwissProtId	erro; resubmissão	06:47:07
center createVizSession	erro; resubmissão	06:47:18
getGoIDs	erro; resubmissão	06:47:23
getSwissProtId	erro; resubmissão	06:47:28
createVizSession	erro; resubmissão	06:47:39
getGoIDs	erro; resubmissão	06:47:44
getSwissProtId	erro; resubmissão	06:47:49
createVizSession	erro; resubmissão	06:48:21
getGoIDs	erro; resubmissão	06:48:26
getSwissProtId	erro; resubmissão	06:48:32
createVizSession	erro; resubmissão	06:48:42
getGoIDs	erro; resubmissão	06:48:48
getSwissProtId	erro; resubmissão	06:48:53
createVizSession	**concluído**	06:50:48
getGoIDs	**concluído**	06:50:54
addTermToViz	submetido	06:50:54
getSwissProtId	**concluído**	06:50:59
getSwissProtRecord	submetido	06:50:59
addTermToViz	**concluído**	06:53:00
getDotFromViz	submetido	06:53:00
getSwissProtRecord	**concluído**	06:53:05
getDotFromViz	**concluído**	06:55:06
destroyVizSession	submetido	06:55:07
destroyVizSession	**concluído**	06:57:13

Tabela 7.7: Histórico de submissão e execução das tarefas

7.3.6 Cenário 5: *Workflow* `TavernaBio` rodando no Grid Completo utilizando FTEC Retry, ocorrendo erro na execução

Neste cenário, buscamos verificar a funcionalidade do *framework* desenvolvido. Para tal, foi executado o *workflow* `TavernaBio` porém durante o início da execução não havia nenhum LRM apto para receber as tarefas. Neste caso, o comportamento padrão do Integrade é o de recusar todas as solicitações de execução com uma mensagem de erro.

Uma vez que o *framework* estava configurado para utilizar o FTEC que implementa o método "`Retry`" de tolerância a falhas, o que se percebeu foi que eventualmente, após várias solicitações de execução para as tarefas, um recurso capaz de executar tal tarefa foi adicionado ao Grid permitindo portanto a execução das tarefas.

Neste cenário, o tempo total de execução foi de 12 minutos e 9 segundos.

O histórico da execução encontra-se descrito na Tabela 7.7.

7.4 Performance da Árvore de Decisões

Para a medição do desempenho do algoritmo implementado para percurso da árvore de decisões definida pelo usuário, utilizamos a mesma árvore utilizada em todos os demais experimentos e ilustrada na Figura 7.5.

Foram realizadas 4 baterias de medição sendo que cada bateria consistia na medição do tempo que o algoritmo levava para escolher o FTEC apropriado para cada uma das tarefas do *workflow* `TavernaBio`.

Os resultados obtidos estão descritos na Tabela 7.8. A partir dos dados podemos perceber que, em todos os casos, a primeira execução tende a ser muito mais demorada que as demais, com um tempo de execução maior que o dobro das demais execuções. Esta particularidade pode ter sido ocasionada pela ocorrência de *page faults* na memória RAM fazendo com que seja necessário buscar as informações em disco.

Nas execuções posteriores observa-se também que o tempo continua com tendência de queda. Entendemos que este comportamento seja consequência do mecanismo gerenciador de *cache*, pois a cada execução aquele trecho de código é utilizado, tornando-o um candidato mais interessante a permanecer na memória *cache* do sistema, e assim reduzindo o tempo de execução do algoritmo.

Primeira Bateria	Segunda Bateria	Terceira Bateria	Quarta Bateria
606	656	1019	617
266	241	359	221
305	236	273	290
170	118	115	166
113	115	308	274
87	171	175	93
56	126	75	65

Tabela 7.8: Duração do algoritmo de percurso da árvore de decisão em milisegundos

7.5 Análise dos Resultados do Framework

Os resultados obtidos e apresentados na Seção 7.3 foram muito positivos. Neles fica claro que, para a plataforma de testes utilizada, a sobrecarga agregada pelo *framework* de seleção dinâmica de técnicas de tolerância a falhas em uma execução *fault-free* é muito pequeno.

Isso permite-nos argumentar que o custo para se realizar um processo de seleção de técnicas de tolerância a falhas de acordo com os recursos disponíveis pode ser muito baixo.

Quando acrescentamos a este fato, os testes realizados em [13] demonstrando que a melhor escolha depende das condições e características do ambiente podemos concluir que a utilização de uma técnica de tolerância a falhas apropriada pode, partindo do pressuposto que a árvore de decisões montada pelo usuário é razoavelmente boa, representar ganho significativo no desempenho e no tempo de execução da tarefa sendo executada.

Isto é ainda melhor evidenciado tendo em vista os resultados da Seção 7.3.5 onde o desempenho obtido no *workflow* varia consideravelmente em função do FTEC escolhido. Dessa forma, em um ambiente onde a ocorrência de falhas é improvável, a escolha pelo FTEC `Retry` parece adequada; contudo, se a ocorrência de falhas for provável, a escolha pelo FTEC `Active` deverá ser mais interessante.

Capítulo 8

Conclusões e Trabalhos Futuros

8.1 Conclusões

A presente dissertação de mestrado propôs e avaliou um *framework* para execução tolerante a falhas de *workflows* no ambiente de Grid. Tal *framework* tinha como principal objetivo prover maior flexibilidade na escolha das técnicas de tolerância a falhas, tanto no sentido de permitir a inclusão de novas técnicas quanto no sentido de permitir que o processo de seleção atenda a regras definidas pelo usuário para um dado *workflow* de forma que a técnica de tolerância a falhas seja selecionada em tempo de execução e não antes do início da execução do *workflow*.

O *framework* proposto atende a esses requisitos. Ele possibilita ao usuário definir uma árvore de decisão ou um mecanismo de decisão totalmente novo para que a seleção das técnicas de tolerância a falhas seja realizada de acordo com as suas necessidades e/ou características do *workflow*. Isto atende à funcionalidade de possibilitar ao usuário a definição das regras de seleção das técnicas de tolerância a falhas.

Tendo as regras definidas, o *framework* pode realizar a escolha da técnica de tolerância a falhas de cada tarefa no momento da submissão da tarefa. Assim, evita-se que a escolha da técnica de tolerância a falhas de cada tarefa seja definida rigidamente antes do início da execução. Isto atende à funcionalidade de realizar a seleção das técnicas de tolerância a falhas em tempo de execução.

Finalmente, o *framework* também define um sistema de comunicação onde o usuário possui liberdade de implementar novas técnicas de tolerância a falhas através da implementação de um novo FTEC. Esta possibilidade está explicitada nos testes onde foram utilizados 3 FTECs distintos: o FTEC `NULL` que não implementava técnicas de tolerância a falhas, o FTEC `Retry` que implementava a técnica de nova tentativa e o FTEC `Active` que implementa a replicação ativa. Isto atende à funcionalidade de permitir que o usuário inclua novas técnicas de tolerância a falhas.

O *framework* foi implantado em um Grid do Laboratório de Pós-Graduação da UnB composto de 5 máquinas. Os resultados obtidos foram divididos em cenários. Os cenários iniciais visavam avaliar a sobrecarga que ocorre em razão da utilização do *framework*. Os demais avaliavam a capacidade do *framework* de lidar com falhas e de absorver novas técnicas sem alterações em seu código.

No primeiro cenário, utilizamos 2 *workflows* com tarefas *dummy* para simular uma execução. Os resultados obtidos foram muito bons pois o aumento no tempo de execução decorrente do uso do *framework* foi de apenas 1.96% e 1.26% nos *workflows* avaliados.

O segundo cenário avaliava um *workflow* com tarefas reais. Os resultados deste cenário não foram avaliados em razão da grande variação dos resultados obtidos. O desvio-padrão foi de 431.56, ou seja, 41% de variação sobre a média em uma mesma bateria (execução sem tolerância a falhas e sem seleção), tornando os dados não confiáveis para a avaliação do desempenho.

O terceiro cenário contornou e apresentou resultados coerentes com o esperado. Neste, a sobrecarga foi ainda menor, com apenas 0.77% de aumento no tempo de execução em função do *framework*.

No quarto cenário, mostramos que é possível incluir novas técnicas de tolerância a falhas sem alterações no código do *framework* através da inclusão da técnica `Active` de replicação ativa. Além disso, mostramos novamente que o desempenho de diferentes técnicas de tolerância a falhas varia de acordo com o ambiente: a técnica *Retry* superou a técnica *Active* no nosso cenário *fault-free*, ao contrário do que ocorreu em [13].

Finalmente, no quinto cenário mostramos que o *framework* é funcional induzindo um erro na execução do *workflow* e verificando que a execução do *workflow* foi retomada tão logo o erro foi sanado.

Considerando estes resultados, pode-se dizer que o custo de se utilizar o *framework* para execução adaptativa e tolerante a falhas de *workflows* em Grid é baixo para os cenários avaliados. Assim sendo, pode-se argumentar que o custo de se utilizar técnicas de tolerância a falhas inadequadas à situação do Grid no momento da execução, selecionadas com base em informações defasadas [26, 13] ou por meio de algoritmos fixos [31, 40, 3, 49, 51], pode facilmente superar a sobrecarga do *framework* proposto nessa dissertação.

8.2 Trabalhos Futuros

Como trabalhos futuros sugere-se:

Previsão de falhas: Verificar a possibilidade de incluir um sistema de previsão de falhas que auxilie o processo de seleção da técnica de TF a ser utilizada em cada tarefa.

Técnicas de Tolerância a falhas em nível de *workflow*: Buscar formas de adaptar o *framework* para lidar com falhas em nível de *workflow*. Atualmente, o *framework* está lida apenas com falhas em nível de tarefa (ver taxonomia da Figura 5.1).

Portabilidade do *framework* para outros Grids: Avaliar a viabilidade de utilizar este *framework* em conjunto com outros *middlewares* que não o Integrade. Apesar de ter sido criado na expectativa de que a portabilidade seja realizada sem maiores problemas, uma vez que apenas dois módulos interagem com o Grid diretamente, isto não foi avaliado.

Avaliação do ganho de desempenho em decorrência da escolha boa ou ruim de técnicas de TF: Realizar uma análise quantitativa do ganho ou perda de desempenho em razão do uso de diferentes conjuntos de regras em diferentes *workflows*.

Introdução de Inteligência Artificial no algoritmo de seleção: Avaliar a possibilidade do uso de técnicas mais elaboradas, como redes neurais ou bayesianas ao invés de árvores de decisão.

Referências

[1] R. Alfieri, R. Cecchini, V. Ciaschini, L. DellâAgnello, A. Frohner, A. Gianoli, K. Lorentey, and F. Spataro. VOMS, an authorization system for virtual organizations. In *Grid Computing*, pages 33–40. Springer, 2004.

[2] O. Ben-Kiki, C. Evans, and I. dot Net. Yaml 1.2 specification, 3rd edition. Disponível em http://www.yaml.org/spec/1.2, Outubro 2009.

[3] J. Bian, C. Weng, J. Du, and M. Li. A QoS-Aware and Fault-Tolerant Workflow Composition for Grid. In *Grid and Cooperative Computing, 2008. GCC'08. Seventh International Conference on*, pages 510–516, 2008.

[4] J. Blythe, S. Jain, E. Deelman, Y. Gil, K. Vahi, A. Mandal, and K. Kennedy. Task scheduling strategies for workflow-based applications in grids. In *IEEE International Symposium on Cluster Computing and the Grid, 2005. CCGrid 2005*, volume 2, 2005.

[5] F. Breg, S. Diwan, J. Villacis, J. Balasubramanian, E. Akman, and D. Gannon. Java RMI performance and object model interoperability: Experiments with Java/HPC++ distributed components. In *ACM 1998 Workshop on Java for High-Performance Network Computing*, 1998.

[6] A. Brown, S. Johnston, and K. Kelly. Using service-oriented architecture and component-based development to build web service applications. *Rational Software Corporation*, 2002.

[7] R. Buyya. Grid computing info centre: frequently asked questions (FAQ). Disponível em http://www.gridcomputing.com/gridfaq.html, 2005.

[8] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services description language (WSDL) 1.1, 2001.

[9] D. Churches, G. Gombas, A. Harrison, J. Maassen, C. Robinson, M. Shields, I. Taylor, and I. Wang. Programming scientific and distributed workflow with Triana services. *Concurrency and Computation: Practice and Experience*, 18(10):1021–1037, 2006.

[10] K. Cooper, A. Dasgupta, K. Kennedy, C. Koelbel, A. Mandal, G. Marin, M. Mazina, J. Mellor-Crummey, F. Berman, H. Casanova, et al. New grid scheduling and rescheduling methods in the grads project. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, 2004.

[11] K. Czajkowski, D. Ferguson, I. Foster, J. Frey, S. Graham, T. Maguire, D. Snelling, and S. Tuecke. From open grid services infrastructure to ws-resource framework: Refactoring & evolution. In *Global Grid Forum Draft Recommendation*, 2004.

[12] K. Czajkowski, D.F. Ferguson, I. Foster, J. Frey, S. Graham, I. Sedukhin, D. Snelling, S. Tuecke, and W. Vambenepe. The WS-resource framework version 1.0. In *Global Grid Forum*, Março 2004.

[13] A. de Sousa et al. A Flexible Fault-Tolerance Mechanism for the Integrade Grid Middleware. In *Proceedings of the Third International Conference on Networking and Services*, page 26. IEEE Computer Society, 2007.

[14] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil, M.H. Su, K. Vahi, and M. Livny. Pegasus: Mapping scientific workflows onto the grid. In *Grid Computing*, pages 11–20. Springer, 2004.

[15] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, K. Blackburn, A. Lazzarini, A. Arbree, R. Cavanaugh, et al. Mapping abstract complex workflows onto grid environments. *Journal of Grid Computing*, 1(1):25–39, 2003.

[16] F. Dong and S.G. Akl. Scheduling algorithms for grid computing: State of the art and open problems. *Queen's University School of Computing*, 2006.

[17] D. Fernández-Baca. Allocating modules to processors in a distributed system. *IEEE Transactions on Software Engineering*, 15(11):1427–1436, 1989.

[18] I. Foster. Globus toolkit version 4: Software for service-oriented systems. *Journal of Computer Science and Technology*, 21(4):513–520, 2006.

[19] I. Foster and C. Kesselman. *The grid: blueprint for a new computing infrastructure*. Morgan Kaufmann, 2004.

[20] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The physiology of the grid: An open grid services architecture for distributed systems integration. In *Open Grid Service Infrastructure WG, Global Grid Forum*, volume 22, page 2002. USA, 2002.

[21] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *International Journal of High Performance Computing Applications*, 15(3):200, 2001.

[22] I. Foster, H. Kishimoto, A. Savva, D. Berry, A. Djaoui, A. Grimshaw, B. Horn, F. Maciel, F. Siebenlist, R. Subramaniam, et al. The open grid services architecture. *Version*, 1:2004, 2004.

[23] F.C. GÄRTNER. Fundamentals of Fault-Tolerant Distributed Computing in Asynchronous Environments. *ACM Computing Surveys*, 31(1), 1999.

[24] 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. *Concurrency and Computation: Practice and Experience*, 16(5):449–459, 2004.

[25] S. Hwang and C. Kesselman. A generic failure detection service for the Grid. *Information Sciences Institute, University of Southern California. Technical Report ISI-TR-568*, 2003.

[26] S. Hwang and C. Kesselman. Grid workflow: a flexible failure handling framework for the grid. In *12th IEEE International Symposium on High Performance Distributed Computing, 2003. Proceedings*, pages 126–137, 2003.

[27] P. Jalote. *Fault tolerance in distributed systems.* Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1994.

[28] G. Kandaswamy, A. Mandal, and D.A. Reed. Fault tolerance and recovery of scientific workflows on computational grids. In *8th IEEE International Symposium on Cluster Computing and the Grid, 2008. CCGRID'08*, pages 777–782, 2008.

[29] J. Koch. Mecanismos de checkpointing coordenado e recuperação para sistemas dsm. Master's thesis, Universidade de Brasília, 2004.

[30] B. Kruatrachue and T. Lewis. Grain size determination for parallel processing. *IEEE software*, 5(1):23–32, 1988.

[31] Y. Li and Z. Lan. Exploit failure prediction for adaptive fault-tolerance in cluster computing. In *Sixth IEEE International Symposium on Cluster Computing and the Grid, 2006. CCGRID 06*, volume 1, 2006.

[32] B. Ludascher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E.A. Lee, J. Tao, and Y. Zhao. Scientific workflow management and the Kepler system. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, 2006.

[33] T. Ma and R. Buyya. Critical-path and priority based algorithms for scheduling workflows with parameter sweep tasks on global grids. In *Computer Architecture and High Performance Computing, 2005. SBAC-PAD 2005. 17th International Symposium on*, pages 251–258, 2005.

[34] J. Nabrzyski, J.M. Schopf, and J. Węglarz. *Grid resource management: state of the art and future trends.* Springer Netherlands, 2004.

[35] F. Nascimento, A. Goldman, and R. Camargo. Implantação automatizada de grades computacionais oportunistas. In *IV Simpósio do Instituto de Matemática e Estatística*, Setembro 2008.

[36] VP Nelson. Fault-tolerant computing: Fundamental concepts. *Computer*, 23(7):19–25, 1990.

[37] T. Oinn, M. Greenwood, M. Addis, M.N. Alpdemir, J. Ferris, K. Glover, C. Goble, A. Goderis, D. Hull, D. Marvin, et al. Taverna: Lessons in creating a workflow environment for the life sciences. *Concurrency and Computation: Practice and Experience*, 18(10):1067–1100, 2006.

[38] K. Plankensteiner, R. Prodan, T. Fahringer, A. Kertesz, and P. Kacsuk. Fault-tolerant behavior in state-of-the-art Grid Workflow Management Systems. Technical report, TR-0091, CoreGRID, 2007.

[39] D. Powell, European Strategic Programme of Research, and Development in Information Technology. *Delta-4, a generic architecture for dependable distributed computing*. Springer-Verlag New York, 1991.

[40] D.M. Quan. Error recovery mechanism for grid-based workflow within SLA context. *International Journal of High Performance Computing and Networking*, 5(1):110–121, 2007.

[41] B. Randell, P. Lee, and PC Treleaven. Reliability issues in computing system design. *ACM Computing Surveys (CSUR)*, 10(2):123–165, 1978.

[42] D.A. Reed. Grids, the teragrid, and beyond. *Computer*, 36(1):62–68, 2003.

[43] M.A.S. Sallem, S.A. de Sousa, and F.J.S. e Silva. AutoGrid: Towards an Autonomic Grid Middleware. *Proceedings of the 16th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 223–228, 2007.

[44] F.B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[45] T.L. Sterling. *Beowulf cluster computing with Linux*. The MIT Press, 2002.

[46] A.S. Tanenbaum and R. Van Renesse. Distributed operating systems. *ACM Computing Surveys (CSUR)*, 17(4):419–470, 1985.

[47] H. Topcuoglu, S. Hariri, and M. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE transactions on parallel and distributed systems*, pages 260–274, 2002.

[48] P. Veríssimo and R. de Lemos. Confiança no funcionamento: Proposta para uma terminologia em portugues. *Publicação conjunta INESC e LCMI/UFSC*, 1989.

[49] M. Wang, K. Ramamohanarao, and J. Chen. Trust-based robust scheduling and runtime adaptation of scientific workflow. *Concurrency and Computation: Practice and Experience*, 21(16):1982–1998, 2009.

[50] J. Yu and R. Buyya. A taxonomy of workflow management systems for grid computing. *Journal of Grid Computing*, 3(3):171–200, 2005.

[51] Y. Zhang, A. Mandal, C. Koelbel, and K. Cooper. Combined Fault Tolerance and Scheduling Techniques for Workflow Applications on Computational Grids. In *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid-Volume 00*, pages 244–251. IEEE Computer Society, 2009.

Livros Grátis

(http://www.livrosgratis.com.br)

Milhares de Livros para Download:

Baixar livros de Literatura
Baixar livros de Literatura de Cordel
Baixar livros de Literatura Infantil
Baixar livros de Matemática
Baixar livros de Medicina
Baixar livros de Medicina Veterinária
Baixar livros de Meio Ambiente
Baixar livros de Meteorologia
Baixar Monografias e TCC
Baixar livros Multidisciplinar
Baixar livros de Música
Baixar livros de Psicologia
Baixar livros de Química
Baixar livros de Saúde Coletiva
Baixar livros de Serviço Social
Baixar livros de Sociologia
Baixar livros de Teologia
Baixar livros de Trabalho
Baixar livros de Turismo