TÚLIO COELHO TAVARES

DESEMPENHO E DISPONIBILIDADE EM SISTEMAS DE FLUXO DE TRABALHO CIENTÍFICO INTENSIVOS EM DADOS

Belo Horizonte, Minas Gerais
11 de agosto de 2006

Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Programa de Pós-Graduação em Ciência da Computação

DESEMPENHO E DISPONIBILIDADE EM SISTEMAS DE FLUXO DE TRABALHO CIENTÍFICO INTENSIVOS EM DADOS

Dissertação apresentada ao Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

TÚLIO COELHO TAVARES

Belo Horizonte, Minas Gerais
11 de agosto de 2006

UNIVERSIDADE FEDERAL DE MINAS GERAIS

FOLHA DE APROVAÇÃO

Desempenho e Disponibilidade em Sistemas de Fluxo de Trabalho Científico Intensivos em Dados

TÚLIO COELHO TAVARES

Dissertação defendida e aprovada pela banca examinadora constituída por:

Prof. Renato Antonio C. Ferreira – Orientador
Universidade Federal de Minas Gerais

Prof. Dorgival Olavo Guedes Neto
Universidade Federal de Minas Gerais

Prof. Wagner Meira Junior
Universidade Federal de Minas Gerais

Prof. Francisco José da Silva e Silva
Universidade Federal do Maranhão

Belo Horizonte, Minas Gerais, 11 de agosto de 2006

Sumário

1 Introdução **1**

1.1 Objetivo 4

1.2 Contribuições do Trabalho 4

1.3 Conceitos 5

1.4 Organização do Texto 7

2 Trabalhos Relacionados **8**

2.1 Anthill 8

2.1.1 Dependência de Dados 10

2.1.2 Tolerância a Faltas no Anthill 11

2.2 Mobius 11

2.2.1 Mako 12

2.3 Sistemas de Fluxo de Trabalho Científico 12

2.3.1 Sistema de Fluxo de Trabalho Científico com Suporte a Banco de Dados XML 13

2.3.2 Chimera e Pegasus 14

2.3.3 Sistema com Reuso de Componentes 15

2.4 DISC e Phoenix 16

2.5 Grid Datafarm 17

2.6 MapReduce 17

2.7 FTOP 18

2.7.1 Sumário 20

3 Disponibilidade em Sistemas Distribuídos **21**

3.1 Modelos de Faltas 21

3.1.1 Modelo de Faltas “Falha e Pára” (*Fail-stop*) 22

3.1.2 Modelo de Queda 23

3.1.3 Modelo de Falta por Omissão 23

3.1.4 Modelo de Faltas Bizantinas 24

3.1.5 Modelo *Fail-Stutter* 24
3.2 Métodos de Tolerância a Faltas 25
3.2.1 *Checkpointing* 26
3.2.2 Message Logging 27
3.2.3 Tolerância a Faltas em *Hardware* 28
3.2.4 Métodos Específicos de Aplicação 29
3.2.5 Replicação Ativa 29
3.2.6 Sumário 29

4 Sistema de Fluxo de Trabalho Científico Proposto 31
4.1 Requisitos Cobertos 31
4.2 Arquitetura Proposta 32
4.2.1 Gerente de Metadados do Fluxos de Trabalho (GMFT) 34
4.2.2 Gerente de Armazenamento de Dados em Memória (GADM) . 35
4.2.3 Gerente de Armazenamento Persistente (GAP) 36
4.3 Aplicação Exemplo: Análise de Placentas de Rato 37
4.3.1 Sumário 39

5 Implementação 41
5.1 Dados e Operações dos Componentes do Sistema 42
5.2 Aumento da Disponibilidade do Sistema 45
5.2.1 Modelo de Faltas e Método de Tolerância a Faltas Adotados . 45
5.2.2 Protocolo de Controle 48
5.2.3 Mecanismo de Recuperação 50
5.2.4 Sumário 55

6 Experimentos 56
6.1 Ambiente Experimental 56
6.2 Resultados 57
6.2.1 Resultados sem Inserção de Faltas 57
6.2.2 Resultados com Inserção de Faltas 59
6.2.3 Outras Aplicações 63

7 Conclusão e Trabalhos Futuros 65
7.1 Trabalhos Futuros 65

Referências Bibliográficas 67

Lista de Figuras

1.1 Fluxo de Trabalho de uma aplicação com quatro diferentes tarefas de análise de imagens. 2
1.2 Propagação: Falta -> Erro -> Falha [VR01] 6

2.1 Visão do modelo de programação *filter stream*. 10
2.2 A arquitetura do Componente Mako do Mobius. 13

4.1 Organização dos Componentes do Sistema de Fluxo de Trabalho Científico. 34
4.2 Fluxo de trabalho da aplicação dividido em estágios. 39
4.3 Instâncias dos filtros durante a execução do fluxo de trabalho da aplicação exemplo. 40

5.1 Camadas das entidades presentes na execução dos fluxos de trabalho. . . 42
5.2 Comunicação entre os componente quando o filtro Classificação das Cores lê um documento. 50
5.3 Comunicação entre os componentes quando o filtro Classificação das Cores envia um documento para o filtro Segmentação do Tecido. 51
5.4 Exemplo da recuperação do sistema após uma falha. Existem M documentos para serem processados da base de dados inicial, e K e V documentos dos fluxos intermediários. 52
5.5 Exemplo de recuperação dos filtros do fluxo de trabalho da aplicação biomédica. 53

6.1 Tempo de execução do filtro PP/PF sem inserção de faltas. 57
6.2 Speed-up. 58
6.3 Execução do filtro Normalização do Histograma sem inserção de faltas. . 59
6.4 Execução dos filtros Classificação das Cores e Segmentação do Tecido sem inserção de faltas. 60
6.5 Speeup da soma dos tempos de execução dos três estágios da aplicação. . 60

6.6 Tempo de execução do filtro PP/PF, rodando em 8 máquinas, em 6 diferentes cenários: 1 - sem falta; 2 - uma falta no filtro da aplicação; 3 - uma falta no GADM; 4 - uma falta no GMFT; 5 - duas faltas em filtros da aplicação; 6 - três faltas em filtros da aplicação 61
6.7 Tempo de execução do filtro Normalização do Histograma, rodando em 8 máquinas, em 6 cenários diferentes: 1 - sem falta; 2 - uma falta no filtro da aplicação; 3 - uma falta no GADM; 4 - uma falta no WMFT; 5 - duas faltas em filtros da aplicação; 6 - três faltas em filtros da aplicação 61
6.8 Tempo de execução dos filtros Classificação das Cores e Segmentação do Tecido, rodando em 6 máquinas, em 6 cenários diferentes: 1 - sem falta; 2 - uma falta no filtro da aplicação; 3 - uma falta no GADM; 4 - uma falta no GMFT; 5 - duas faltas em filtros da aplicação; 6 - três faltas em filtros da aplicação 63

Lista de Tabelas

6.1 Percentual do tempo, durante a execução do segundo estágio da aplicação, em que o GMFT fica ocioso, esperando por requisições. 59

Capítulo 1

Introdução

À medida que a Ciência da Computação vem evoluindo e alcançando diversas áreas de conhecimento, o processo de análise de dados tem-se tornado uma atividade extremamente significante em várias pesquisas científicas. Entre muitas áreas podem citar-se as ciências naturais, tais como a biologia e a geografia, cujos dados são obtidos por instrumentos como microscópios, telescópios, ou sensores climáticos, ou por simulações numéricas. Como um exemplo, considere-se o desenvolvimento de novas terapias para tratar doenças. Um laboratório biomédico começará eventualmente a pesquisa no nível molecular e celular, então conduzirá um grande número de experimentos em animais e, mais tarde, vai transferir sua pesquisa para pacientes. Para alcançar o sucesso em cada um desses passos, os cientistas precisam usar ambientes computacionais para analisar imagens, por exemplo, de células ou de órgãos, obtidas por meio de microscópios de alta resolução.

Para ajudar os pesquisadores em suas análises, foram introduzidos os **Sistemas de Fluxo de Trabalho Científico** [FVWZ02, DBG+03, ABJ+04, LAB+05, BLNC06, HRL+05]. *Fluxos de Trabalho Científicos* são processos nos quais tarefas são estruturadas baseadas nos conceitos, teorias, experimentos e dados usados pelos cientistas para conseguirem a transformação de dados brutos em resultados publicáveis [ABJ+04]. Eles são centrados nos dados e podem ser modelados como redes de processos baseadas em fluxos de dados (*dataflow process networks*) [LP95], um modelo de computação que suporta execução de processos concorrentes com comunicação baseada em fluxos. Os fluxos de trabalho podem ser descritos como um grafo direcionado cíclico ou acíclico no qual os nodos representam componentes de processamento da aplicação, e as arestas representam os fluxos de dados trocados entre esses componentes. Em um exemplo específico de uma aplicação de análise de imagens biomédicas, um fluxo de trabalho pode ser visto como o mostrado na figura 1.1. Esse exemplo envolve a análise de *slides* de placentas de rato digitalizadas para o estudo de mu-

danças do fenótipo induzidas por manipulações do genótipo. Nessa figura podem ver-se ver quatro diferentes componentes de um fluxo de trabalho científico. Cada um deles está relacionado a uma tarefa de análise de imagens, e essas tarefas devem ser aplicadas em seqüencia aos slides.

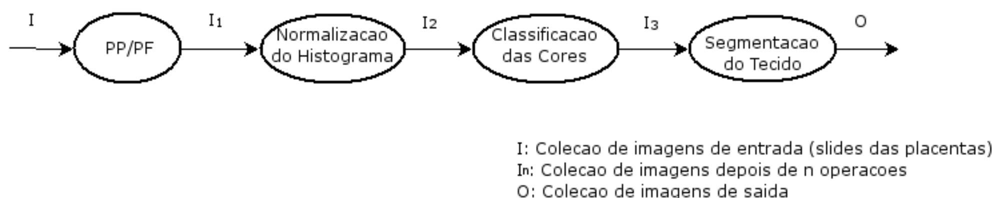


Figura 1.1: Fluxo de Trabalho de uma aplicação com quatro diferentes tarefas de análise de imagens.

Os Sistemas de Fluxo de Trabalho Científico foram criados para prover aos cientistas um ambiente no qual eles podem realizar diversas atividades. Entre várias delas podemos citar: descrever e criar componentes baseados nas tarefas que eles querem executar em seus experimentos; organizar esses componentes em fluxos de trabalho baseados na semântica da sua aplicação; executar esses fluxos de trabalho sob bases de dados; e também monitorar a execução pelo exame, por exemplo, de dados intermediários criados durante ela. Algumas das idéias introduzidas por esses sistemas são juntar componentes tais como retorno de dados, computação e visualização, em um único *pipeline*, e tornar reusáveis esses componentes . A reutilização de componentes introduz o fato de que componentes podem fazer parte dos sistemas de fluxo de trabalho, do fluxo de trabalho de uma outra aplicação, ou até mesmo de componentes externos acessados, por exemplo, via serviços web ou de grid [BLNC06].

Os desafios para o projeto e a implementação desses sistemas são muitos, principalmente devido às características das aplicações que geram os fluxos de trabalho científicos. Elas são consideradas aplicações intensivas em dados e processamento as quais criam uma enorme quantidade de dados durante a execução e executam por longos períodos. Um exemplo é o projeto Large Hadron Collider (LCD) do CERN. Começando neste ano de 2006, esse projeto gerará dados na escala de *petabytes* a partir de quatro grandes detectores de partícula subterrâneos a cada ano [CER]. Projetos como o Grid Datafarm [TMM+02] estão sendo realizados para o desenvolvimento de sistemas de fluxo de trabalho científico que processem os dados eficientemente em ambientes distribuídos. Alguns dos desafios para projetar esses sistemas são: armazenar, pesquisar e gerenciar grandes bases de dados distribuídas, gerenciar os dados de entrada e de saída assim como o escalonar e a monitorar da execução desses

fluxos de trabalho em ambientes distribuídos, e otimizar o reuso de componentes de diferentes fluxos.

Como os sistemas de fluxo de trabalho científico estão sendo construídos para executar aplicações por longos períodos, a probabilidade de ocorrer uma falha durante a execução tem aumentado bastante [KKL05] e não tem recebido tanta atenção. Dessa forma, é desejável que esses sistemas possuam mecanismos que aumentem a sua disponibilidade [HRL+05, LAB+05]. A finalidade desses mecanismos é a de que o trabalho (processamento) realizado anteriormente ao momento da falha não seja perdido, ou seja, para que não haja a necessidade de começar a execução das aplicações do seu início. Embora muitos sistemas distribuídos já tenham tratado esse problema, ainda é um desafio prover mecanismos que lidem com grandes bases distribuídas e que gerenciem trocas maciças de dados entre as aplicações a um baixo custo.

Esse desafio se torna ainda maior quando os sistemas de fluxo de trabalho são tratados. Esses sistemas disponibilizam os resultados intermediários da execução de um fluxo de trabalho para serem inspecionados pelos seus usuários, ou até mesmo para servirem de entrada para outros fluxos de trabalho ou para o próprio fluxo que os gerou, para que parte desse possa ser reexecutada sob novos parâmetros. Essa tarefa é bastante cara; para que seja realizada, há a necessidade do uso de um sistema de armazenamento persistente distribuído, onde dados intermediários serão salvos em bases de dados criadas sob demanda. Na figura 1.1, por exemplo, uma base de dados diferente seria criada para cada coleção de imagens: I_1, I_2 e I_3. É interessante observar que o tamanho dessas imagens pode ser da ordem de centenas ou milhares de *megabytes*, fazendo com que as bases de dados sejam muito grandes.

Este trabalho investiga o uso de mecanismos que, de forma transparente, aumentem a disponibilidade de sistemas de fluxo de trabalho científico, de tal forma que o trabalho a ser refeito após uma falha no sistema seja mínimo. Esses mecanismos procuram utilizar como base características próprias desses sistemas, como a disponibilização de resultados intermediários, para a construção de um sistema de armazenamento dos dados necessários para a recuperação dos sistemas após uma falha. Do ponto de vista de eficiência, esse sistema deve ser capaz de escalar grandes bases de dados, e deve prover um armazenamento assíncrono dos dados de tal forma que não haja necessidade do travamento da execução dos fluxos de trabalho para que ele aconteça.

1.1 Objetivo

Os sistemas de fluxo de trabalho científico estão sendo construídos para executar aplicações por longos períodos de tempo que lidem com quantias de dados cada vez maiores. A possibilidade de falhas ocorrerem durante as execuções dessas aplicações tem aumentado bastante, fazendo com que o trabalho realizado anteriormente à falha seja todo perdido. Introduzir mecanismos capazes de aumentar a disponibilidade desses sistemas é necessário, entretanto esses mecanismos não podem introduzir grande *overhead* nesses sistemas.

O objetivo deste trabalho é projetar, desenvolver, e analisar mecanismos que aumentem a disponibilidade em sistemas de fluxo de trabalho científico, de tal forma que o trabalho perdido por uma falha seja mínimo. Esses mecanismos devem ser capazes de gerenciar grandes quantidades de dados que serão utilizados para recuperar esses sistemas, e o impacto introduzido por eles deve ser pequeno.

1.2 Contribuições do Trabalho

As contribuições deste trabalho são:

- Aumento da disponibilidade dos sistemas de fluxo de trabalho: com as técnicas introduzidas nesses sistemas, eles serão capazes de se recuperar quando uma falha acontecer em algum dos seus componentes ou até mesmo nos nodos nos quais eles estão processando.

- Melhora na dinâmica da execução dos sistemas de fluxo de trabalho: uma vez que o processo de disponibilização de resultados intermediários é beneficiado com as técnicas para aumento da disponibilidade desses sistemas, há uma melhora na dinâmica execução dos fluxos de trabalho. Como os dados podem ser compartilhados entre vários fluxos, essa melhora pode ser ainda maior, pois eles não precisam ser computados novamente.

- Gerenciamento e armazenamento eficiente de dados replicáveis: para que o impacto das técnicas que aumentem a disponibilidade dos sistemas de fluxo de trabalho seja pequeno, há necessidade de um gerenciamento e armazenamento eficiente dos dados que podem ser utilizados para recuperar o sistema de uma falha.

1.3 Conceitos

Três termos que geram bastante confusão, quando a disponibilidade em sistemas é discutida, são: falta (*fault*), falha (*failure*) e erro (*error*). Vários autores tem-se ocupado da nomenclatura e conceitos básicos da área. Os conceitos apresentados aqui são derivados dos trabalhos de Avizienis e Laprie [Lap85, AL86, ALR01] e Anderson e Lee [AL81].

A falha do sistema ocorre quando o serviço fornecido se desvia das condições mencionadas na sua especificação, ou seja, na descrição do serviço esperado. A construção de um sistema confiável consiste em prevenir a ocorrência de falhas. A falha ocorre porque o sistema estava incorreto: um erro é a parte do estado do sistema que pode conduzir à falha, isto é, ao fornecimento de um resultado que não está de acordo com o serviço especificado. Dessa forma a falha pode ser observada externamente como o efeito de um erro. A causa de um erro é uma falta. Uma falta pode existir muito antes de produzir efeitos: diz-se inativa. Exposto em outros termos, um erro é a manifestação de uma falta no sistema, e uma falha é a manifestação de um erro no serviço.

Um defeito no código de um programador é considerado uma falta. A conseqüência dessa falta será um erro (latente) no *software* escrito, por meio de instruções ou dados errados. Quando ativado o módulo no qual o erro reside, ou seja, quando um padrão de entrada fizer utilizar a instrução, seqüencia de instruções ou dados errados, o erro tornará efetivo. Quando esse erro efetivo produz dados errados, que afetem o serviço do sistema, a falha ocorre.

Adotando a nomenclatura de Laprie [Lap85], um sistema pode ser decomposto em um conjunto de componentes ligados uns aos outros para interagirem, sendo que um componente pode ser um outro sistema. Essa recursão é finalizada quando o sistema é considerado atômico: qualquer outra estrutura interna não pode ser discernida, ou não é de interesse e pode ser ignorada. Dessa forma, como podemos observar na figura 1.2, o erro ocorrido em um componente pode se propagar por outros de modo que cabe ao projetista do sistema decidir em qual desses componentes esse erro deve ser tratado.

Quando sistemas distribuídos são discutidos em vários trabalhos, muitas são as nomenclaturas utilizadas para os descreverem. Neste trabalho, assumimos que um sistema distribuído é composto por um conjunto de processos que executam em nodos de processamento, que são ligados utilizando canais de comunicação. Quando os sistemas de fluxo de trabalho são mencionados neste trabalho, por natureza estamos assumindo que eles são distribuídos.

Em um sistema distribuído onde os componentes são desenvolvidos utilizando

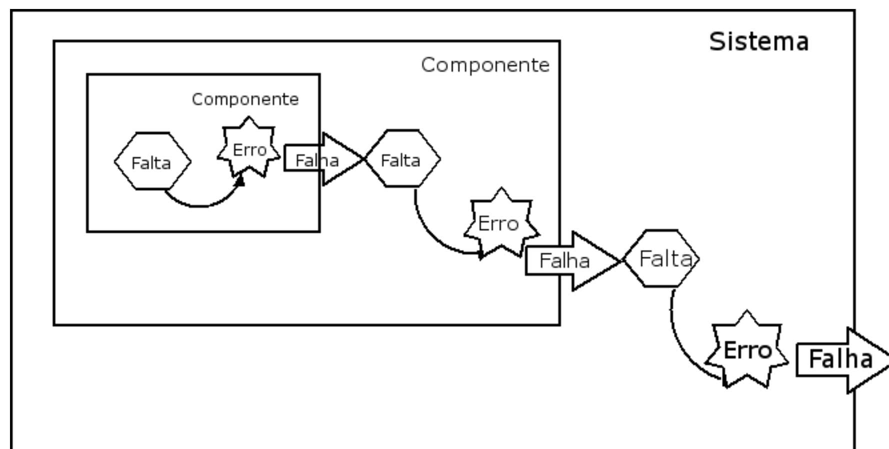


Figura 1.2: Propagação: Falta -> Erro -> Falha [VR01]

múltiplas camadas de forma independente, a decisão de quais erros devem ser propagados e quais erros devem ser tratados em cada camada não é uma decisão muito bem entendida [TL02, LKL04]. O argumento fim-a-fim (*end-to-end*) [SRC84] declara que o lugar correto para uma funcionalidade é no ponto final, mas ela pode ser colocada em níveis mais baixos por questões de desempenho. Colocar todas as funcionalidades o ponto final aumenta a complexidade e requer que desenvolvedores finais possuam um conhecimento do que pode ocorrer nas camadas inferiores. Nos sistemas de fluxo de trabalho, onde os desenvolvedores (usuários) são especialistas de um determinado domínio, esse argumento iria exigir que os usuários possuíssem conhecimento sobre como lidar com os erros.

Thain e Livny [TL02] desenvolveram uma teoria de propagação de erro. Eles definem o escopo do erro como a porção do sistema que um erro invalida e descrevem que um erro tem que ser propagado para o processo que gerencia esse escopo. Dessa forma os erros que podem acontecer em um ambiente distribuído são do escopo do sistema que está executando nesse ambiente, e não do escopo da aplicação. Com o auxílio dessa teoria, decidimos colocar a camada para tratar os erros e aumentar a disponibilidade do sistema de fluxo de trabalho em camadas intermediárias que podem gerenciar esses erros.

Em todo o texto, quando mencionarmos usuários, estamos nos referindo aos usuários do sistema de fluxo de trabalho. Esses usuários são os cientistas das diversas áreas de conhecimento que desenvolvem suas aplicações como fluxos de trabalho, e utilizam o sistema para executá-los.

1.4 Organização do Texto

Este trabalho está dividido em 7 capítulos. O restante dele está dividido da seguinte maneira:

Capítulo 2 [Trabalhos Relacionados] Nesse capítulo os trabalhos relacionados são descritos e discutidos em relação ao trabalho apresentado nesta dissertação.

Capítulo 3 [Disponibilidade em sistemas Distribuídos] Nesse capítulo serão apresentados os modelos de faltas, assim como os mecanismos de tolerância a faltas existentes, para aumentar a disponibilidade dos sistemas distribuídos.

Capítulo 4 [Sistema de Fluxo de Trabalho Científico Proposto] Nesse capítulo serão apresentados alguns dos requisitos dos sistemas de fluxo de trabalho científicos, assim como a arquitetura do sistema proposta neste trabalho.

Capítulo 5 [Implementação] Baseado nos modelos e mecanismos apresentados no capítulo 3, apresentaremos aqueles que foram utilizados nos sistemas de fluxo de trabalho científico. Nesse capítulo ainda detalharemos os algoritmos, os protocolos e as estruturas de dados utilizados na implementação dos mecanismos que aumentam a disponibilidade do sistema de fluxo de trabalho científico.

Capítulo 6 [Experimentos] Nesse capítulo serão descritos os experimentos realizados para avaliar o desempenho do sistema, utilizando-se os mecanismos para aumentar sua disponibilidade, assim como seu comportamento quando falhas são inseridas durante a sua execução.

Capítulo 7 [Conclusão e Trabalhos Futuros] Nesse capítulo serão apresentas as conclusões obtidas neste trabalho, assim como possíveis trabalhos futuros.

Capítulo 2

Trabalhos Relacionados

Neste capítulo serão apresentadas sínteses dos principais trabalhos relacionados a sistemas de fluxo de trabalho científico e à disponibilidade em sistemas computacionais de larga escala. Na seção 2.1, apresentamos o *framework* Anthill cujo modelo de programação filtro-fluxo permite que aplicações sejam decompostas em diferentes unidades de processamento (filtros) que, ligadas pelos fluxos, formam *pipelines* parecidos com os das aplicações dos sistemas de fluxo de trabalho. Esse sistema foi utilizado como base para a construção do sistema apresentado neste trabalho e possui um mecanismo de tolerância a faltas que não foi aproveitado aqui. Outro *framework* utilizado foi o Mobius, apresentado na seção 2.2.

Na seção 2.3, apresentamos diferentes sistemas de fluxo de trabalho científico que apresentam esforços para a implantação de mecanismos que aumentem a sua disponibilidade. Nas seções 2.4 e 2.5, serão apresentados três sistemas que são desenvolvidos para executarem aplicações intensivas em dados e que provêem tolerância a faltas. Finalmente, nas seções 2.6 e 2.7, apresentamos implementações de mecanismos de tolerância a faltas de dois sistemas de computação distribuída, MapReduce e PVM, respectivamente.

2.1 Anthill

O Anthill [FJG+05] é um ambiente para desenvolvimento e execução de aplicações distribuídas escaláveis. Ele foi desenvolvido com o objetivo de atender aplicações não regulares, intensivas em processamento e em entrada-e-saída (E/S) de dados. Nesses tipos de aplicação, os dados encontram-se distribuídos nos vários nodos do sistema, e uma característica que o Anthill utiliza sabiamente é levar a computação aonde o dado se encontra, reduzindo a comunicação através da rede. Ele procura tirar vantagem da premissa de que mover os dados entre os nodos para

então serem processados é freqüentemente uma operação ineficiente, principalmente porque, à medida que o processamento avança, os dados resultantes tendem a ser muitas vezes menores que os dados de entrada [PFT+05].

Nesse contexto, aplicações a serem paralelizadas no Anthill devem levar em consideração tanto o paralelismo de dados quanto o de tarefas, ou seja, a aplicação deve ser dividida em etapas que sejam passíveis de execução em nodos diferentes do sistema, e cada uma dessas etapas irá executar parte das transformações sobre os dados, iniciando-se com o conjunto de dados de entrada, até que se atinja o conjunto de dados de saída [PFT+05], formando um *pipeline* de tarefas ou transformações. A estratégia do Anthill aplica os dois enfoques, agregando uma terceira dimensão que permite explorar o grau de assincronia existente entre diferentes tarefas independentes no sistema ao longo do tempo. Os benefício dessas três dimensões combinadas permite atingir *speedups* elevados experimentalmente [VJF+04, FJG+05].

Alguns dos conceitos implementados no Anthill são derivados do *Datacutter*, um sistema de execução de aplicações distribuídas, baseado no modelo de programação *filter stream*. Nesse modelo, o processamento é dividido em tarefas que operam sob os dados que fluem pelo sistema. Cada filtro implementa uma tarefa que transforma os dados segundo a necessidade da aplicação e se comunica com outros filtros pelos canais de comunicação responsáveis pela transmissão contínua de dados (*streams* ou fluxos). Dessa forma, criar uma aplicação no Anthill é um processo de decomposição de processamento em filtros.

A figura 2.1 apresenta a visão desse modelo de programação baseado no *filter stream*. Durante a execução, o processo definido para cada filtro é instanciado em diferentes nodos do ambiente distribuído. A esses processos dá-se o nome de *cópias transparentes* ou *instâncias* de um filtro, como mostrado nessa figura. Dessa forma, o processamento pode ser distribuído por muitos nodos, e os dados que devem fluir por aquele filtro podem ser particionados pelas suas instâncias, produzindo o paralelismo de dados desejado.

O Anthill possui duas extensões do modelo *filter stream* original. Foi verificado que muitas aplicações precisavam compartilhar certo estado global sobre a evolução da computação, o que levou os autores a definir um *broadcast stream*, que permite esse padrão de comunicação para todas as instâncias de um filtro. Além disso, para garantir uma localidade de processamento, ou seja, para que dados que compartilham uma certa característica na semântica da aplicação possam ser processados pela mesma instância, foi criado o *labeled stream*. Esse tipo de comunicação permite exatamente que a instância de destino dos dados seja determinada em função de alguma propriedade derivada do seu conteúdo.

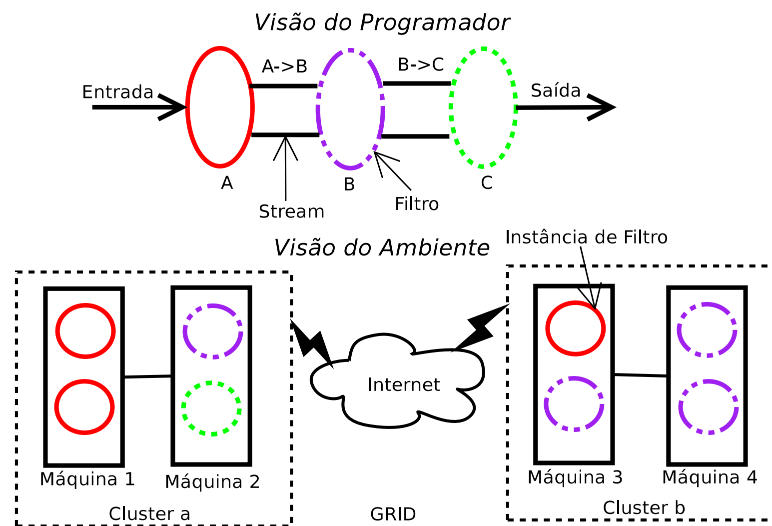


Figura 2.1: Visão do modelo de programação *filter stream*.

2.1.1 Dependência de Dados

Quando as aplicações utilizam o modelo de programação *filter stream*, elas apresentam uma característica descrita aqui como dependência de dados. Para um filtro gerar uma saída para outro, ele tem que primeiro receber uma quantia de dados (algumas mensagens pelo *stream*) e esses dados podem ser definidos como dependência da saída. Os tipos de dependência são descritos a baixo:

- ***1 para 1*:** essa é o tipo mais simples. Quando um filtro recebe uma mensagem, ele pode processar os dados nessa mensagem e enviar o resultado para o próximo filtro no *pipeline*. Dessa forma, a mensagem de entrada é dependente da de saída.

- ***n para 1*:** um filtro tem que receber n mensagens, de um ou mais *streams*, antes de gerar uma saída. O n pode variar de acordo com a aplicação e também assumir diferentes valores durante a execução de uma mesma aplicação. A mensagem de saída é dependente das n de entrada.

- ***n para m*:** esse tipo é similar ao *n para 1*, entretanto, m mensagens de saída podem ser geradas pelo filtro, em um ou mais *streams* de saída.

- ***1 para n*:** esse é caracterizado por um filtro que recebe uma mensagem de entrada e gera várias de saída, usando um ou mais *stream* de saída.

O conceito de dependência de dados é muito importante neste trabalho. O mecanismo para aumentar a disponibilidade do sistema de fluxo de trabalho o uti-

liza para controlar todas as mensagens trocadas durante a execução dos fluxos de trabalho. Listas de dependência de mensagens são criadas para cada mensagem enviada. Os capítulos 4 e 5 descrevem como esse controle é realizado.

2.1.2 Tolerância a Faltas no Anthill

O Anthill possui um mecanismo de tolerância a faltas descrito por Coutinho em [Cou05]. Esse mecanismo é baseado no conceito de tarefa. A execução de uma aplicação é definida como a execução de um conjunto finito e definido de tarefas. Uma tarefa constitui uma seqüencia de operações bem delimitadas, onde dados podem ser armazenados de forma perene, de tal forma que tarefas posteriores possam acessar esses dados, criando, assim, dependência entre as tarefas.

Coutinho definiu que a execução de uma tarefa é atômica, o que lhe permitiu que o grão da tolerância a faltas fosse uma tarefa inteira. Ele utiliza mecanismos de *checkpoint* para replicar os estados das tarefas, e, como não são permitidas mensagens entre tarefas, esses mecanismos foram simplificados, não havendo necessidade de lidar com a comunicação entre as instâncias dos filtros.

Embora essa abordagem de tarefas seja bastante interessante, ela impõe algumas consideráveis restrições sobre o modelo de programação das aplicações. A primeira restrição diz respeito à composição da aplicação em filtros. É esperado que as aplicações sejam cíclicas, ou seja, que exista um *loop* entre os filtros. A segunda restrição é quanto à complexidade de programação de aplicações. Nesse modelo, os programadores das aplicações devem dividir seus programas em tarefas, divisão que pode não ser simples.

2.2 Mobius

O Mobius [HLOS04] é um *framework* projetado para o gerenciamento eficiente de metadados e dados em ambientes dinâmicos e distribuídos. Ele provê um conjunto de serviços e protocolos genéricos para suportar a criação e gerenciamento de esquemas de bases de dados, criação sob demanda de bases de dados, federação de bases de dados existentes, e consulta a dados em ambientes distribuídos. Seus serviços empregam esquemas XML para representar definições de metadados e documentos XML para representar e trocar instâncias de metadados.

O Mobius possui três componentes principais:

- GME (Global Model Exchange): um serviço parecido com o DNS para a criação, geração de versões e compartilhamento universal de descrições de dados.

- Mako (data instance management): um serviço que expõe e abstrai fontes de dados como XML e permite a instanciação de armazenamento de dados e gerenciamento federativo de armazenamentos existentes distribuídos.
- DTS (Data Translation Service): um serviço que permite a tradução eficiente e confiável de descrições de dados entre instituições.

Neste trabalho, o componente Mako será descrito em mais detalhes devido ao fato de ele ser o único componente utilizado na arquitetura do sistema de fluxo de trabalho proposta e apresentada na seção 4.2.

2.2.1 Mako

O Mako provê serviços para ambientes *grid* para armazenar e consultar dados e metadados. Ele permite que dados sejam armazenados por meio de máquinas heterogêneas e fracamente integradas. Ele também permite a usuários confiáveis a habilidade de atualizar, consultar e apagar os dados que eles armazenam. Bases de dados podem ser instanciadas para um fim específico e podem ser, na prática, de qualquer tamanho e escala, crescendo dinamicamente de acordo com a necessidade.

O Mako expõe os dados como serviços XML através de um conjunto de interfaces bem definidas baseadas no protocolo Mako. Esse protocolo define métodos para submeter, atualizar, remover e retornar documentos XML. Na submissão, o Mako assinala um identificador único a cada documento XML submetido. Documentos, ou subconjuntos dos documentos XML, podem ser retornados ou removidos especificando seus identificadores únicos, ou por meio de expressões XPath [BWC+03].

A arquitetura do Mako, como mostrada na figura 2.2, contém um conjunto de ouvintes (*listeners*) que permitem a clientes comunicar-se com uma instância do Mako utilizando protocolos de comunicação tais como TCP, SSL, ou GSI (*Globus Security Infrastructure*). Os pacotes são então transmitidos para um roteador de pacotes, o qual determina se o pacote possui um *handler* no Mako e, se sim, transmite o pacote para o *handler* para processar e enviar uma resposta para o cliente. A implementação atual provê suporte para expor bases de dados XML que suportam a API do XMLDB e contém uma implementação do MakoDB. O MakoDB é um banco de dados XML construído sob o MySQL [htt].

2.3 Sistemas de Fluxo de Trabalho Científico

Hoje em dia podemos encontrar muitos sistemas de fluxo de trabalho científico na literatura [FVWZ02, DBG+03, ABJ+04, LAB+05, BLNC06, HRL+05]. Embora

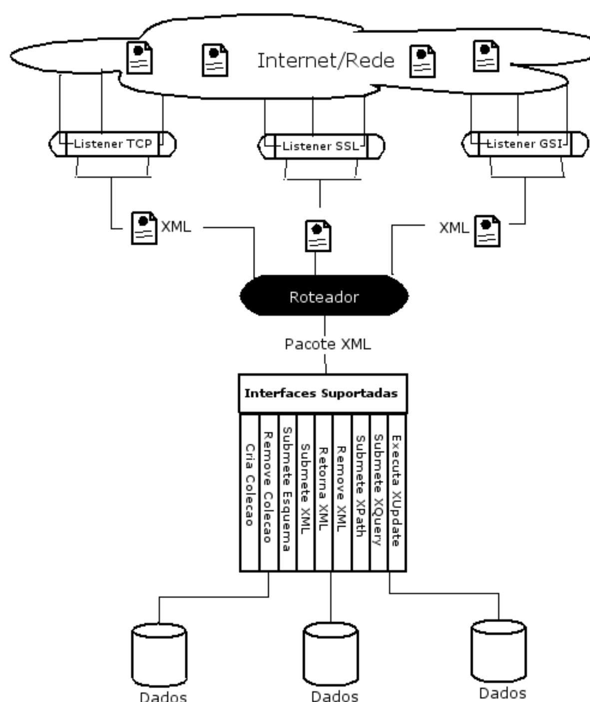


Figura 2.2: A arquitetura do Componente Mako do Mobius.

esses sistemas tenham sido desenvolvidos para executar aplicações intensivas em dados, poucos deles apresentam a preocupação de aumentar sua disponibilidade. Nesta seção apresentamos os principais sistemas e procuramos dar uma maior ênfase nos mecanismos que aumentam a sua disponibilidade.

2.3.1 Sistema de Fluxo de Trabalho Científico com Suporte a Banco de Dados XML

XML tem se tornado um padrão para armazenamento, pesquisa e troca de dados em ambientes como Web e Grid. Um número grande de ferramentas têm sido desenvolvidas para criar, fazer *parsing*, validar e pesquisar documentos XML. Com uma proposta baseada em XML, torna-se possível para ambos, clientes e desenvolvedores de aplicações, utilizar essas ferramentas. Em [HRL+05], Hastings entre outros investiga a aplicação efetiva do suporte a banco de dados XML distribuído em fluxos de trabalho científicos. Ele examina o uso de tecnologias XML para tratar assuntos associados com composição e gerenciamento de fluxos de trabalho, definições e armazenamento de dados em ambientes distribuídos.

Naquele artigo, os autores descrevem um sistema de fluxo de trabalho científico que utiliza um *middleware* para processamento de dados distribuído baseado

no paradigma filtro-fluxo, e outro *middleware* para armazenamento de dados distribuído. A arquitetura apresentada permite que aplicações desenvolvidas utilizando o primeiro *middleware* sejam executadas e possuam os seus dados intermediários armazenados no segundo. Para que esse armazenamento seja realizado, há a necessidade de os desenvolvedores das aplicações explicitamente utilizarem em seu código uma API (*Application Programming Interface*) para criar bases de dados sob demanda, escrever e ler os dados. No nosso sistema, todo o gerenciamento e o armazenamento dos dados é realizado transparentemente à aplicação.

Outro ponto negativo para o trabalho apresentado por Hastings é que, quando os dados intermediários são armazenados, a execução da aplicação tem que ser realizada em estágios, de tal forma que um estágio deve terminar seu processamento por completo e realizar o armazenamento dos dados para então o próximo estágio utilizar esses dados como entrada. Esse problema é resolvido no nosso trabalho utilizando-se uma camada de armazenamento eficiente em memória que permite que a execução da aplicação seja desacoplada da tarefa de armazenamento de dados.

Os autores em [HRL+05] discutem que, utilizando os dados armazenados no banco de dados distribuído, é possível aumentar a robustez do sistema, tornando-o tolerante a faltas. Entretanto, mecanismos para realizar essa tarefa, como detecção de faltas e recuperação, não são discutidos no artigo.

2.3.2 Chimera e Pegasus

O projeto Chimera [FVWZ02] tem desenvolvido um sistema de dados virtual capaz de representar o processo de derivação de dados, e dados já derivados, para explicar sua origem. Seus autores querem ser capazes de rastrear como as produções de dados são originadas, com precisão suficiente para que esses dados possam ser utilizados para reexecutar aplicações e gerar os dados novamente. Um catálogo de dados virtuais (baseado em um esquema de dados virtual relacional) provê uma representação compacta e expressiva de procedimentos utilizados para gerar os dados, assim como invocações a esses procedimentos e bases de dados produzidas por essas invocações. Um interpretador de linguagem de dados executa requisições para construir e executar consultas a bancos de dados.

Utilizando o Chimera, pessoas podem criar ou recriar dados através do conhecimento adquirido no rastreamento. Elas podem então explicar definitivamente como os dados produzidos foram criados, prática que geralmente não é possível mesmo em banco de dados cuidadosamente investigados. Elas podem ainda implementar uma nova classe de operações de "gerenciamento de dados virtual" que, por exemplo, "rematerializa" dados que foram apagados, gera dados que foram definidos mas nunca

foram criados, geram novamente dados quando suas dependências ou programas de transformações mudam, e/ou até mesmo cria réplicas de dados produzidos em locais remotos, quando sua criação for mais eficiente que sua transferência.

O Chimera é acoplado a outros serviços de dados em grid [CFK+99, KF98] que permitem a criação de novos dados pela execução de escalonadores de computação, obtidos de consultas a banco de dados, e o gerenciamento distribuído de dados resultantes.

O Pegasus [DBG+03] pode criar sistemas de dados virtuais que salvam informações sobre a processo de derivação dos dados e dados derivados, utilizando o Chimera. Ele também mapeia os fluxos de trabalho abstratos do Chimera em fluxos de trabalho concretos DAG que o escalonador DAGMan [FTFT01] executa.

Diferentemente do Chimera, o sistema de fluxo de trabalho científico apresentado neste trabalho foca o armazenamento de resultados parciais e a utilização desses dados para aumento da disponibilidade desse sistema. Nós não armazenamos uma grande quantidade de informação sobre a origem dos dados, mas somos capazes de eficientemente armazenar bases de dados geradas durante a execução das aplicações.

2.3.3 Sistema com Reuso de Componentes

Em [BLNC06], Bowers apresenta uma arquitetura para a construção de componentes reusáveis em sistemas de fluxos de trabalho científicos. A abordagem proposta é baseada no encapsulamento de especificações do comportamento genérico dos componentes nos chamados *templates*. Os *templates* são componentes distintos e separados que podem ser reutilizados em outros fluxos de trabalho. Eles são especificações parciais e contém "buracos", chamados *frames*, que atuam como espaços reservados para subcomponentes definidos independentemente. Compor *templates* com componentes de fluxo de dados existentes resulta em aplicar o comportamento associado ao componente de tal forma que a separação entre o controle e os dados do fluxo é mantido, permitindo que componentes de base dos fluxos de dados sejam trocados facilmente.

A arquitetura apresentada é composta de três camadas: a superior é representada como uma "armadura" dentro de um grafo de fluxo de dados e denota uma tarefa particular (por exemplo, transferência de dados ou execução remota). Essa "armadura" superior pode ser encaixada a um de vários *templates* (camada intermediária), onde cada um define um comportamento de controle para uma tarefa. Um *template* tem um ou mais *frames* que podem ser encaixados para a implementação de uma tarefa (por exemplo, *scp* ou *ssh*). As várias implementações dos *frames* formam a camada inferior da arquitetura.

O autor argumenta que, utilizando componentes construídos baseados nessa arquitetura, é possível aumentar a confiança do sistema, se esses componentes apresentarem mecanismos de tolerância a faltas. Por exemplo, quando um componente de transferência de dados, que possui mecanismos para fazer a retransmissão dos dados quando essa não for bem sucedida, fosse utilizado, por conseqüência a confiança do sistema estaria aumentando. Assim como esse componente, o autor apresenta um outro para execução remota de tarefas que também possui um mecanismo muito simples de tolerância a faltas. Ambos os mecanismos não permitem a reconstrução do sistema após uma falha, a qual minimiza o trabalho a ser refeito.

2.4 DISC e Phoenix

Kola, em [KKF+04], analisa os problemas com os atuais sistemas que lidam com aplicações intensivas em dados em ambientes distribuídos, e usa o resultado dessa análise para projetar o DISC. Um dos problemas apresentados pela análise é a movimentação dos dados como parte da computação, o que acarreta a recomputação dos dados, quando a transferência de dados falha. Para resolver esse problema ele propõe uma estratégia de isolamento de falta que desacopla a localização dos dados da sua computação. A camada de armazenamento eficiente para armazenar os dados intermediários das execuções das aplicações, apresentado nesta dissertação, também resolve esse problema.

O DISC é capaz de diferenciar as falhas que podem ocorrer em *grids* de computadores como persistentes e transientes, e automaticamente recuperar falhas transientes. Entretanto, essa recuperação segue instruções antes fornecidas pelos usuários, ou seja, não são utilizados mecanismos transparentes.

Em [LKL04], Kola propõe o Phoenix. Esse sistema provê ferramentas, para ambientes *grid*, que permitem tolerância a faltas para aplicações intensivas em dados que operem nesse tipo de ambiente. Essas ferramentas, assim como o DISC, de forma transparente à aplicação, são capazes de detectar e classificar as falhas, entre transientes e permanentes, e tratar cada falha transiente da melhor forma possível que foi anteriormente definida pelo usuário. Diferentemente deste trabalho de mestrado, que está mais preocupado no armazenamento eficiente dos dados e na recuperação do sistema após a falha, o Phoenix está mais preocupado na detecção e na classificação transparente das falhas.

2.5 Grid Datafarm

Em [TMM+02], Tatebe apresenta a arquitetura do Grid Datafarm (Gfarm). Esse sistema de arquivos possui como objetivo o gerenciamento de grandes bases de dados na escala de *petabytes*. O modelo apresentado cobre aplicações onde os dados primários consistem de um conjunto de objetos os quais são analisados independentemente, e ele procura tirar proveito dessa localidade de acesso para escalonar o processamento sobre os dados distribuídos, conseguindo uma boa escalabilidade. O autor discute a idéia de que, replicando os dados, ele automaticamente consegue tolerância a faltas. Entretanto, o Gfarm não provê transparência na replicação e no controle dos dados, exigindo que seus usuários tenham conhecimento e realizem grande parte do trabalho.

Um arquivo do Gfarm é considerado como um arquivo de larga escala que é dividido em fragmentos que são distribuídos entre os discos do sistema de arquivo Gfarm e que serão acessados em paralelo. O Gfarm apresenta uma API para criação, visão e acesso paralelo a esses arquivos:

- ***gfs_pio_open:***. Abre um arquivo determinado. Flags como a GFARM_FILE_REPLICATION podem ser especificadas para replicar o arquivo em disco local quando o acesso for remoto.
- ***gfs_pio_create:***. Cria um arquivo Gfarm. Permissões de acesso, como escrita e leitura, podem ser especificadas na criação.
- ***gfs_pio_set_view_local:*** Visão local de um arquivo. Permite que os processos acessem os próprios fragmentos do arquivo.
- ***gfs_pio_set_view_index:*** Explicitamente especifica um fragmento de um arquivo.
- ***gfs_pio_read:*** Bloqueia a acesso ao arquivo e realiza a leitura do número de *bytes* especificados de um fragmento.
- ***gfs_pio_write:*** Bloqueia a acesso ao arquivo e realiza a escrita do número de *bytes* especificados a um fragmento.

2.6 MapReduce

O MapReduce, proposto por Dean em [DG04], é um modelo de programação e uma implementação associada para processar e gerar grandes bases de dados.

Uma abstração foi criada para permitir que os usuários expressem computações simples que eles desejam realizar, mas esconde detalhes complicados de paralelismo, tolerância a faltas, distribuição de dados e balanceamento de cargas em uma biblioteca. A abstração foi inspirada em primitivas de mapear (*map*) e reduzir (*reduce*) presentes em Lisp e em muitas linguagens funcionais. Os usuários especificam uma função de mapeamento que processa um par chave/valor para gerar um conjunto de pares chave/valor intermediários, e uma função de redução que une todos os valores intermediários associados com a mesma chave intermediária.

Muitas são as similaridades do MapReduce, do Anthill e do Datacutter. Programas seqüenciais escritos utilizando o estilo necessário são automaticamente paralelizados e executados em grandes aglomerados de computadores. Esses sistemas ficam responsáveis por detalhes como execução dos programas nos conjuntos de máquinas, tratamento de falhas de máquina (com exceção do Datacutter), e gerenciamento da comunicação necessária entre os processos das máquinas. O MapReduce ainda possui funcionalidades para particionamento dos dados de entrada e escalonamento dos processos.

O MapReduce também possui um gerente responsável por enviar o trabalho a ser realizado para os trabalhadores. Esses trabalhadores são criados pela biblioteca do MapReduce e quando estiverem ociosos, receberão tarefas de mapeamento ou redução. À medida que essas tarefas forem sendo completadas, os resultados intermediários vão sendo armazenados no disco local de cada máquina. Utilizando chamadas de procedimento remoto, esses dados podem ser acessados por trabalhadores de outras máquinas. O processo de armazenamento de dados pode limitar a eficiência desse ambiente, uma vez que ele é realizado diretamente no disco. Neste trabalho propomos uma camada de armazenamento de dados eficiente que primeiramente armazena os dados em memória para mais tarde armazená-los no disco, o que não acarreta o travamento da aplicação durante esse processo.

O mecanismo de tolerância a faltas implementado no MapReduce é baseado na reexecução de tarefas que estavam sendo ou foram executadas nas máquinas que falharam. O gerente fica responsável por notificar trabalhadores que estavam acessando os dados nas máquinas que falharam para realizar a leitura nas novas máquinas que estarão executando a tarefa novamente.

2.7 FTOP

O FTOP [BGS02] é uma biblioteca de *checkpoint* integrada com o PVM (*Parallel Virtual Machine*). Ele implementa um mecanismo de checkpoint para aplicações

distribuídas executando sobre o PVM. Os desenvolvedores das aplicações precisam inserir algumas chamadas a procedimentos em suas aplicações para utilizarem o mecanismo de tolerância a faltas, sendo que nenhuma modificação é necessária no núcleo do sistema operacional.

O FTOP assume que um sistema distribuído consiste num grupo de máquinas rodando LINUX e conectadas através de uma LAN de alta velocidade. Em cada uma dessas máquinas há necessidade de um PVM executando. Todas elas podem falhar com exceção de uma que pode executar tanto o gerente que é responsável pelo escalonamento de tarefas, assim como pela coordenação dos protocolos de checkpoint e recuperação, quanto o armazenamento confiante que consiste dos arquivos de checkpoint, logs de mensagens etc. O modelo de faltas assumido é o de faltas “falha e pára”, em que os processos do sistema distribuído são notificados quando uma falta ocorre (esse modelo é descrito em mais detalhes na seção 3.1.1).

No checkpoint, o estado de um processo é congelado e armazenado em um armazenamento confiável, do qual ele pode ser recuperado no caso de falhas. O FTOP escolheu o protocolo de checkpoint coordenado não blocante no qual os processos decidem em realizar os checkpoints (maiores detalhes estão descritos na seção 3.2.1). Esse protocolo permite que os processos continuem seu processamento durante a realização do checkpoint, o que pode introduzir menos *overhead*.

Salvar e recuperar o estado de processos envolve o salvamento de GPRs, registradores de ponto flutuante etc, os quais podem depender da arquitetura. Em vez de de acessar os módulos dependentes de linguagens de máquina para cada arquitetura para efetuar essa tarefa, o FTOP lida com isso através de sinais. Os mecanismos tratadores de sinais do sistema operacional requisitam o salvamento do estado de execução do processo o qual pode ser mais tarde restaurado, uma vez que o sinal foi tratado. O FTOP trata de arquivos abertos suportando apenas operações de leitura e anexação (*append*). Operações de escrita não são tratadas uma vez que, segundo os autores, elas não são comuns e envolvem um grande *overhead*.

A detecção de faltas é realizada utilizando-se mecanismos presentes no PVM. Processos PVM ocasionalmente checam seus pares através de envios de mensagens de *ping*. Quando um processo PVM não responde, dado um tempo, o processo que o detectou avisa os demais que houve falta no sistema.

A recuperação de faltas envolve a restauração de um estado sem faltas da execução. Um protocolo de duas fases foi implementado no FTOP para realizar essa recuperação. Na primeira fase o gerente informa aos processos que eles devem voltar para seu último checkpoint salvo. Quando todos os processos conseguem voltar para o último checkpoint com sucesso, o gerente envia uma mensagem confirmando.

Durante esse protocolo de recuperação não é permitido o envio e o recebimento de mensagens pelos processos.

Apesar de o FTOP apresentar um protocolo de checkpoint eficiente, existem alguns gargalos que tornam difícil utilizar como base para construção de sistemas de fluxo de trabalho científico. Primeiramente, os checkpoints são armazenados em uma só máquina do ambiente distribuído, que se torna um grande gargalo, quando arquivos muito grandes forem tratados. Segundo, ele não lida com operações de escrita de arquivos, o que torna complicado garantir que os resultados intermediários salvos durante a execução dos fluxos de trabalho realmente vão apresentar o conteúdo correto durante uma falta.

2.7.1 Sumário

Neste capítulo apresentamos os principais trabalhos relacionados a sistemas de fluxo de trabalho científico e disponibilidade em sistemas de larga escala. Como podemos perceber, os mecanismos utilizados para aumentar a disponibilidade dos sistemas de fluxo de trabalho são bastante limitados. Pouco foi estudado até o momento, o que faz com que haja uma grande carência nessa área. Antes de apresentarmos o sistema proposto neste trabalho, assim como os mecanismos para aumentar a sua disponibilidade, no próximo capítulo vamos apresentar os principais modelos de faltas e métodos de tolerância a faltas que são utilizados em sistemas distribuídos no intuito de torná-los mais disponíveis.

Capítulo 3

Disponibilidade em Sistemas Distribuídos

Aumentar a disponibilidade em sistemas distribuídos significa incluir mecanismos de tolerância a faltas nesses sistemas, os quais permitem que processos sobrevivam a faltas que ocorram dentro do sistema, seja nos seus componentes seja nos nodos que o executa. Sem tolerância a faltas, um sistema executando em paralelo, em diversos nodos, poderia falhar inteiramente se apenas um simples nodo executando parte dele falhasse. A escolha de um método para ser utilizado por um sistema deve ser feita levando-se em consideração suas características de tal forma que um *overhead* muito grande não seja introduzido.

Nas seções 3.1 e 3.2 serão descritos os modelos de faltas e os métodos de tolerância a faltas, respectivamente, encontrados na literatura, que geralmente são utilizados como base nos estudos de tolerância a faltas.

3.1 Modelos de Faltas

Um modelo de faltas especifica as suposições de um projetista sobre a natureza das faltas que um sistema, ou componente de um sistema, pode sofrer. Ele caracteriza o modo como um componente vai falhar, sem fazer nenhum relato sobre as causas atuais da falta. Um modelo de faltas então limita o número e os tipos de faltas que um desenvolvedor de um sistema tem que antecipar e lidar. Nesta seção serão apresentados os principais modelos de faltas assim como alguns exemplos de faltas cobertas por cada modelo.

3.1.1 Modelo de Faltas "Falha e Pára" (*Fail-stop*)

O Modelo de Faltas "Falha e Pára" (*Fail-stop*), primeiramente apresentado por Schlichting *et al.* em [SS83], permite que qualquer nodo ou componente do sistema falhe a qualquer momento, mas, quando a falha ocorrer, ele cessa a produção de saídas e a interação com o resto do sistema, não sendo capaz de produzir respostas erradas ou maliciosas. Dessa forma, cada componente está sempre trabalhando ou não, e, quando um componente falha, todos os outros tornam-se cientes da falha. Este modelo representa faltas comuns como travamento e queda do sistema, mas não lida com faltas mais sutis tais como corrupção aleatória de memória [Tre04].

Faltas do tipo "Falha e Pára" são geralmente muito simples de detectar, uma vez que o componente defeituoso do sistema cessa suas operações. Desta forma, a simplicidade desse modelo tem feito com que ele seja a base para muitos trabalhos de tolerância a faltas. De fato, embora a maioria das faltas não se encaixem nesse modelo, o comportamento dessas demais faltas podem ser adaptadas para o comportamento "Falha e Pára", usando-se mecanismos ortogonais bem conhecidos, tais como a redundância modular tripla [GR93] ou a replicação de estado esperta [CL99]. Entretanto, quanto maior a complexidade da falta, maior será o impacto da sua detecção na execução do sistema.

Vários são os exemplos que produzem faltas do tipo "Falha e Pára" nos sistemas. Numa primeira categoria podem citar-se as faltas que causam a queda do sistema. Nessa categoria encontram-se faltas na aplicação, como as que surgem através de erros de programação, erros em sistemas operacionais, exceções não tratadas etc, assim como as faltas no próprio nodo, ou máquina, como queima da fonte da máquina, queda de energia etc.

Uma segunda categoria de faltas envolve aquelas que proporcionam o travamento do nodo, ou máquina. Nesse caso, a máquina pára de realizar processamentos e de responder a requisições externas; por exemplo: pára de enviar pacotes de rede, não responde a interações de entradas padrão etc. Essas faltas geralmente são causadas por erros no sistema operacional ou por defeitos de *chipset*. Em 2004, por exemplo, foi descoberta uma vulnerabilidade no Kernel do Linux em que usuários, mesmo não possuindo acesso privilegiado, quando compilavam alguns programas utilizando versões do GCC 3.0 a 3.3.2 rodando no Kernel 2.4.2x ou 2.6.x, faziam com que a máquina travasse [CAI04]. Geralmente quando esse tipo de falta ocorre, há a necessidade da intervenção humana para reiniciar a máquina. Entretanto, existem alguns sistemas que automatizam essa operação, reiniciando adequadamente a máquina, como é o caso do HP Integrated Lights Out Standard (iLO) [L.P], o Linux Networks IceboxTM Cluster Management Appliance [LN], além de watchdog timers [MM88].

3.1.2 Modelo de Queda

O modelo de Queda (*Crash*) pode ser considerado uma extensão do modelo "Falha e Pára". Ele permite que qualquer nodo ou componente do sistema falhe a qualquer momento, cessando permanentemente a execução de suas ações. Entretanto nesse modelo não há garantia de que os outros componentes do sistema irão detectar a falta. Devido a essa característica, ele também é chamado de modelo de falta silenciosa (*fail-silent*) [PVB+88].

Um exemplo clássico de falha desse modelo é o travamento da aplicação. Um estudo realizado por Kola *et al.* sobre faltas em duas grandes aplicações distribuídas, US-CMS e BMRB BLAST, mostrou que essa é uma das faltas mais freqüentes [KKL05]. Alguns dos processos paravam indefinidamente e nunca retornavam, o que era bastante difícil determinar se o processo estava fazendo algum progresso ou se estava realmente travado. A causa mais comum era o travamento na transferência de dados, devido à perda do reconhecimento de para quem o arquivo estava sendo transferido. Numa fração menor dos travamentos estava um problema não conhecido envolvendo o servidor NFS.

Outros exemplos de travamento de aplicação que podem ser citados são erros de programação como *loops* infinitos e *deadlock* que podem ser causados em aplicações paralelas. Esses erros também são muito difíceis de detectar. Dessa forma, técnicas de verificação automática de código, como ferramentas de análise estática de código e verificação formal, principalmente o projeto Meta-Level Compilation [AE02, YTEM04], têm ganho destaque ultimamente [Cou05].

3.1.3 Modelo de Falta por Omissão

Uma falha por omissão ocorre quando, em resposta a uma seqüência de entrada, o componente nunca dá a saída especificada, ou seja, um componente do sistema sempre vai produzir uma saída correta em tempo ou nunca a produzirá [ES86]. As faltas do modelo de queda podem ser vistas como uma subclasse das faltas por omissão, de tal forma que elas ocorrem depois que um componente sistematicamente omite responder a todos os eventos de entrada subseqüentes à primeira omissão de uma saída [CASD85].

As faltas por omissão são utilizadas para modelar componentes como uma rede que ocasionalmente abandona pacotes. Outros exemplos de faltas por omissão são: a queda de um processador, o colapso de um link, um processador que de vez em quanto não passa adiante uma mensagem, entre outros.

3.1.4 Modelo de Faltas Bizantinas

O Modelo de Faltas Bizantinas, inicialmente apresentado por Lamport através do Problema dos Generais Bizantinos [LSP82], permite que os componentes do sistema possuam um comportamento totalmente arbitrário e inconsistente. Nesse modelo, os nodos defeituosos podem continuar interagindo com o resto do sistema, e também é permitido que eles colaborem com o objetivo de elaborar saídas maliciosas. Os nodos operando corretamente não podem detectar automaticamente que algum dos nodos falhou, muito menos saber quais nodos em particular falharam, se a existência da falha é conhecida. Esse modelo pode representar faltas aleatórias no sistema, assim como ataques maliciosos de um *hacker*.

As faltas bizantinas geralmente são ignoradas devida à sua complexidade de detecção. Por causa da dificuldade natural de detectá-las, tem-se assumido que essas faltas são extremamente raras, e os problemas associados na sua detecção e tolerância podem ser, na sua maior parte, ignorados [DHSZ03]. É provado que nenhuma garantia na detecção de faltas pode ser feita em um sistema com $3m + 1$ nodos em operação normal quando mais que m nodos estão experimentando faltas bizantinas.

A filosofia do projeto de aglomerados de computadores de alto desempenho a baixo custo tem influenciado a utilização de um *hardware* não tão preciso e seguro, uma vez que um *hardware* com confiança alta é muito caro. À medida que esse *hardware* é integrado em um circuito para alcançar o alto desempenho, a elevada freqüência do *clock*, assim como as geometrias mais densas e as voltagens mais baixas de força fornecidas podem elevar a taxa com a qual os circuitos entrem em colapso, e conseqüentemente as faltas bizantinas podem vir a tornar-se mais comuns.

Exemplos de faltas bizantinas que podem ser citadas são: corrupção dos dados, por exemplo, através da reversão de bits de saída que podem ser causadas por eventos tais como interferência eletromagnética e radiação externa [DNR02]; faltas causadas por defeitos de *hardware* ou *software* devido ao envelhecimento, danos externos ou sabotagem, as quais causam erros repetidos em computações para o resto do tempo de vida do sistema.

3.1.5 Modelo *Fail-Stutter*

O modelo *fail-stutter* [ADAD01, KKL05] é uma extensão do modelo “Falha e Pára”. Ele tenta manter a tradicionalidade do modelo, ou seja, leva em consideração que os componentes de um sistema de vez em quando falham, mas também expande o conjunto de faltas reais que ele inclui, introduzindo as *faltas de desempenho*. Uma

falta de desempenho é um evento no qual um componente provê um desempenho abaixo do esperado, mas não interrompe o seu funcionamento. Essa extensão permite que o modelo inclua faltas tais como um baixo desempenho na latência de um switch de rede, quando repentinamente alcança um tráfego de carga muito elevado. Embora introduza muitas vantagens, esse modelo de faltas ainda não tem sido muito aceito pelo comunidade [Tre04].

Alguns exemplos dessas faltas são as que ocorrem com os processadores. Existem faltas que fazem com que processadores de um mesmo tipo possam vir a apresentar desempenho diferente quando testados sobre as mesmas condições. Por exemplo, geralmente o mascaramento de faltas é utilizado para aumentar a produção de processadores, permitindo que chips parcialmente defeituosos sejam usados; o resultado é que chips com características diferentes são vendidos como idênticos. Arpaci *et al.* [ADV95] examinaram o tamanho da cache de uma série de processadores Viking da Sun e descobriram, entre outras coisas, que processadores que eram vendidos com especificação de cache nível 1 de 16 KB tinham na verdade 4KB.

Outros exemplos são as faltas que ocorrem nos discos. Os discos também apresentam um grau de mascaramento de faltas. Como documentado em [AD99], um experimento de largura de banda mostra desempenho diferente entre alguns discos 5400-RPM Seagate Hawk. Embora a maioria apresente 5.5 MB/s em leituras seqüenciais, um disco apresentou apenas 5.0 MB/s.

Algumas faltas de software também podem ser enquadradas nesse modelo. Virtualmente, todas as máquinas hoje em dia usam endereços físicos no endereçamento das caches. A menos que a cache seja pequena o suficiente, então o *offset* da página não é utilizado nesse endereçamento, e a alocação de páginas vai afetar a taxa de *cache-miss*. Chen e Bershad mostraram que decisões de mapeamento de memória virtual podem reduzir o desempenho de aplicações em até 50% [CB93].

Esse modelo também inclui outros tipos de faltas como as de temporização que ocorrem quando um componente dá a saída especificada muito cedo, muito tarde, ou nunca [CASD85].

3.2 Métodos de Tolerância a Faltas

Existem vários métodos que podem ser incorporados a sistemas distribuídos para torná-los tolerantes a faltas. Entre esses métodos, os mais discutidos na literatura são os mecanismos de *checkpointing* e de *message logging*. Além desses métodos, também existem alguns outros que já foram empregados em sistemas de computação, como tolerância a faltas em *hardware*, métodos específicos de aplicação, replicação

ativa etc. Nessa seção serão apresentados esses métodos clássicos individualmente. É interessante observar que geralmente eles são combinados e então empregados nos sistemas.

3.2.1 *Checkpointing*

Os mecanismos de *checkpoints* se baseiam em protocolos nos quais cada nodo periodicamente salva o seu estado em algum dispositivo de armazenamento estável. O estado salvo contém informação suficiente para reiniciar a execução do processo. Um *checkpoint* global consistente é um conjunto de N *checkpoints* locais, um para cada nodo, formando um estado consistente para o sistema. Qualquer um dos *checkpoints* globais podem ser utilizados para reiniciar a execução do nodo após uma falha, sendo desejável minimizar o trabalho gasto na restauração do sistema [EAWJ96].

Os protocolos de recuperação de falta baseados em *checkpoints* possuem poucas restrições e são bastante simples de serem implementados. Entretanto, esses protocolos não garantem que a execução anterior à ocorrência da falha possa ser deterministicamente restaurada depois da recuperação [EAWJ96].

As técnicas para recuperação de falta baseadas em *checkpoints* podem ser classificadas em três categorias: *checkpointing* não coordenado, *checkpointing* coordenado, e *checkpointing* induzido pela comunicação. Analisamos cada categoria a seguir.

- ***Checkpointing* Não Coordenado**: As técnicas de *checkpointing* não coordenado permitem a cada nodo ter uma autonomia máxima ao decidir quando realizar *checkpoints*. A principal vantagem dessa autonomia é que cada nodo pode tomar a decisão de armazenar o seu estado no momento que considerar mais conveniente. Por exemplo, um nodo pode reduzir o *overhead* realizando *checkpoints*, quando a quantidade de informação a ser salva é pequena [Di87]. Entretanto, existem muitas desvantagens. Primeira: existe a possibilidade do efeito dominó, que pode fazer com que todos os nodos voltem ao inicio da sua computação [Di87]. Segunda: um nodo pode realizar um *checkpoint* sem necessidade, fazendo com que *checkpoints* nunca sejam utilizados para o estado global consistente do sistema. Terceira: *checkpoints* não coordenados forçam cada nodo a manter várias cópias de seus estados anteriores em algum dispositivo de armazenamento seguro e periodicamente invocar um algoritmo coletor de lixo para remover cópias que não serão mais utilizadas. Para determinar um *checkpoint* global na recuperação, os nodos usam as dependências entre seus *checkpoints* que foram guardadas utilizando-se técnicas discutidas em [EAWJ96].

- ***Checkpointing* Coordenado**: As técnicas de *checkpointing* coordenado requerem que os nodos coordenem seus *checkpoints* para formar um estado global consistente. Essas técnicas simplificam a recuperação de faltas e não são suscetíveis ao efeito dominó. Uma vantagem que podemos destacar é que essas técnicas requerem que cada nodo mantenha apenas um *checkpoint* em um dispositivo de armazenamento seguro, eliminando a necessidade de um coletor de lixo. Sua desvantagem principal, entretanto, é a grande latência envolvida no envio de informações para armazenamentos estáveis, havendo necessidade de um *checkpoint* global antes desse envio. Para minimizar essa situação existem técnicas não blocantes que permitem o *checkpointing* coordenado [EAWJ96].

- ***Checkpointing* Induzido pela Comunicação**: As técnicas de *checkpointing* induzido pela comunicação evitam o efeito dominó, enquanto permitem que os nodos tenham alguma independência nas suas verificações. Entretanto os nodos podem ser forçados a realizar verificações adicionais para garantir que haverá sucesso na recuperação da falha. Os *checkpoints* que um nodo realiza independentemente são chamados de *checkpoints* locais, enquanto aqueles que um nodo é forçado a realizar são chamados de *checkpoints* forçados. Essas técnicas adicionam informações nas mensagens trocadas entre os nodos. O receptor de cada mensagem usa as informações adicionais para determinar se há necessidade de realizar uma verificação forçada para ser incorporada ao estado global do sistema.

3.2.2 Message Logging

O mecanismo de *message logging* é baseado na suposição de que a execução de um processo em um nodo é determinística entre as mensagens de entrada recebidas, ou seja, se dois nodos começam no mesmo estado e recebem a mesma seqüência de mensagens, eles têm que produzir a mesma seqüencia de saída e têm que terminar no mesmo estado. O estado do nodo é então completamente determinado por seu estado inicial e pela seqüência de mensagens recebidas [Joh89].

Os protocolos usados para *message logging* podem ser divididos em dois grupos, chamados de *message logging pessimista* e *message logging otimista*, de acordo com o grau de sincronização imposto pelo protocolo na execução do sistema.

- ***Message Logging* Pessimista**: Os protocolos pessimistas armazenam as mensagens sincronamente. O protocolo garante que qualquer nodo defeituoso pode ser recuperado individualmente sem afetar os estados de outros nodos que

não tenham falhado, e previne os nodos de prosseguirem até que o armazenamento das mensagens tenha sido completado. Esses protocolos são pessimistas porque assumem que a falha pode ocorrer a qualquer momento, possivelmente antes que o armazenamento das mensagens necessárias seja completado. As vantagens encontradas nesses protocolos é que eles são capazes de restaurar o sistema depois de uma falha sem afetar os estados de outros nodos que não falharam. Entretanto, sua principal desvantagem é a degradação do desempenho causada pela sincronização no protocolo de armazenamento de mensagens.

- ***Message Logging* Otimista**: Em contraste com os protocolos pessimistas, os otimistas operam assincronamente. O receptor de uma mensagem não é bloqueado, e mensagens são armazenadas após seu recebimento, por exemplo agrupando várias mensagens e escrevendo-as em um dispositivo de armazenamento seguro em uma só operação. Entretanto, o estado corrente de um nodo só pode ser recuperado, se todas as mensagens recebidas anteriormente à ocorrência da falha foram armazenadas corretamente. Esses protocolos são chamados de otimistas porque assumem que o armazenamento de cada mensagem recebida pelo nodo será completado antes de o nodo falhar, e são projetados para tratar esse caso mais efetivamente. Esses protocolos otimistas possuem a vantagem de significantemente reduzir o *overhead* causado pelo armazenamento de mensagens. Apesar de os protocolos otimistas requererem um procedimento mais complexo para recuperar o sistema de uma falha, esse procedimento só é usado quando uma falta ocorre. A principal desvantagem dos protocolos otimistas é que a recuperação do sistema devido a uma falha pode levar um tempo maior para completar, uma vez que mais nodos podem participar [Joh89].

3.2.3 Tolerância a Faltas em *Hardware*

Métodos de tolerância a faltas inteiramente implementados em *hardware* geralmente apresentam um melhor desempenho do que aqueles implementados em *software* [Car86, Sie86, Joh89]. Dois exemplos clássicos de sistemas que utilizam métodos de tolerância a faltas em *hardware* são o sistema de telefone eletrônico ESS desenvolvido pela AT&T [CG87] e o ARPANET Pluribus IMP [KEM+78]. Tais métodos de *hardware*, entretanto, são menos flexíveis e não podem ser facilmente adicionados a sistemas existentes.

Uma maneira de conseguir tolerância a faltas em *hardware* é através da redundância dos componentes dentro do sistema, por exemplo, uso de múltiplos processadores,

barramentos, ou fontes de energia que operem simultaneamente e em paralelo, comparando resultados das operações realizadas.

3.2.4 Métodos Específicos de Aplicação

Métodos de tolerância a faltas específicos de aplicação [SH82, RK06] são aqueles desenvolvidos especificamente para um programa particular que os use. Esses desenvolvimentos requerem conhecimento tanto da aplicação quanto do seu ambiente. Cada tipo de falta que possa vir a ocorrer no sistema deve ser antecipada, e soluções específicas para cada falta devem ser adotadas. A implementação desses métodos pode seguir alguma estrutura geral tal como o uso de blocos de recuperação [SS83], ou pode ser estruturada especialmente para cada aplicação. Entretanto esses métodos não são transparentes e requerem que programas existentes sejam cuidadosamente modificados ou reescritos para serem tolerantes a faltas. Em contraste, métodos de tolerância a faltas que usam *message logging* e *checkpointing* são de propósito geral e podem ser aplicados transparentemente a novos programas existentes. Embora em alguns casos os métodos específicos de aplicação possam ser construídos para serem mais eficientes do que os métodos de propósito geral, eles são limitados a sua falta de transparência.

3.2.5 Replicação Ativa

A replicação ativa envolve a execução concorrente de múltiplas cópias independentes de cada processo em nodos (processadores) separados, de tal forma que cada réplica do mesmo processo receba a mesma seqüência de entrada, e é esperado que produza a mesma seqüencia de saída. Se uma réplica falhar, as demais réplicas daquele processo continuam a computação sem interrupção. Exemplos de sistemas que utilizam replicação ativa incluem o sistema ISIS [BJ87] e o WAFT [AM98].

O método de replicação ativa é bem adaptado para uso em sistemas de tempo real, uma vez que a recuperação de uma falha é essencialmente imediata. Entretanto, essa habilidade requer processadores extras para serem dedicados a cada programa para suas réplicas.

3.2.6 Sumário

Uma vez que os modelos de faltas e os métodos de tolerância a faltas foram estudados, e suas vantagens e desvantagens foram identificadas, temos a base necessária para propormos mecanismos que aumentem a disponibilidade em sistemas distribuídos. Para que esses mecanismos sejem adequados e eficientes para os sistemas de

fluxo de trabalho científico, precisamos identificar as caracteríticas desses sistemas. No próximo capítulo (4) apresentamos as principais características assim como a arquitetura do sistema proposta neste trabalho, e posteriormente, no capítulo 5, mostramos quais foram os modelos de faltas e os métodos de tolerância a faltas utilizados como base.

Capítulo 4

Sistema de Fluxo de Trabalho Científico Proposto

A visão de que cientistas tipicamente executam experimentos e de que esses experimentos podem ser considerados coleções ordenadas de tarefas atuando sobre dados e envolvendo uma variedade de atividades distintas motiva a exploração do paradigma de *fluxo de trabalho científico*. Como foi dito anteriormente, o gerenciamento, a manipulação e o armazenamento de grandes volumes de dados, assim como a garantia de disponibilidade, nesses ambientes, são os grandes desafios a serem enfrentados.

Neste capítulo serão discutidos quais dos requisitos necessários para os sistemas que dão suporte à execução de fluxos de trabalho científicos foram cobertos neste trabalho, e será descrita a arquitetura do sistema proposto.

4.1 Requisitos Cobertos

Para suportar fluxos de trabalho científicos complexos, um sistema de fluxo de trabalho deve satisfazer uma grande variedade de requisitos [HRL+05, SM96]. Esses requisitos incluem interfaces de usuário e linguagens para fácil composição de fluxos de trabalho, escalonamento, execução e monitoramento de fluxos de trabalho, gerenciamento de coleções de dados, confiança e robustez. Os principais requisitos tratados neste trabalho são:

- Composição de fluxos de trabalho: já que a pesquisa é um processo de evolução e mudanças, pesquisadores devem ser capazes de compor, registrar, e controlar diferentes versões de fluxos de trabalho. As definições e instâncias desses fluxos devem ser gerenciadas de uma maneira eficiente e padrão de tal forma que

pesquisadores possam compartilhar seus fluxos e referenciar outros. A criação de bases de dados com resultados intermediários da execução dos fluxos deve acontecer para que elas sirvam tanto para inspeção desses resultados por seus usuários, quanto de entrada para outros fluxos de trabalho ou para o próprio fluxo de trabalho sob novos parâmetros. Mais ainda, a otimização da criação e do uso desses resultados intermediários por diferentes fluxos de trabalho é muito importante para aumentar a eficiência da execução destes últimos.

- Gerenciamento de coleções de dados: um fluxo de trabalho pode ser executado de forma que todas as tarefas executem e troquem dados dinamicamente através de uma rede. Uma outra abordagem seria a utilização de um sistema de armazenamento para atuar como canal persistente de dados e informações entre tarefas em diferentes estágios do fluxo de trabalho. Tal abordagem permite um agendamento mais flexível das tarefas do fluxo de acordo com a disponibilidade de recursos e também permite que coleções de dados intermediárias sejam verificadas, podendo ter um melhor entendimento do resultado final.

- Disponibilidade: o sistema deve garantir que os fluxos de trabalho executados pelos usuários serão terminados, e o resultado gerado vai estar de acordo com suas expectativas. Para isso ele deve ter mecanismos que sejam capazes de reiniciar a execução dos fluxos de trabalho científicos após uma falha. Levando-se em consideração que esses fluxos podem executar por períodos muito grandes de tempo, por exemplo dias ou semanas, esses mecanismos devem garantir que a execução a ser refeita seja mínima, ou seja, o processamento realizado antes da falha deve ser utilizado ao máximo. Esses mecanismos também devem se adequar a características do sistema, de modo que a eficiência dos fluxos de trabalho não sofram um impacto muito grande.

- Arquitetura descentralizada: o sistema de gerenciamento de fluxo de trabalho deve ser organizado de uma forma descentralizada, ou seja, ele não deve depender de servidores ou banco de dados centrais; tais componentes iriam representar fáceis pontos de falhas e gargalos no desempenho.

4.2 Arquitetura Proposta

No sistema de fluxo de trabalho científico proposto, os fluxos de trabalho dos usuários são descritos como grafos direcionados, mapeados para pipelines do Anthill, utilizando-se filtros (*filters*) e fluxos (*streams*) como unidades para processamento e comunicação de dados, respectivamente. Usando um arquivo XML de configuração,

tal como do Anthill, o usuário pode descrever quais são os seus filtros e como eles comunicam um com o outro.

Basicamente o sistema é composto de três componentes: o Gerente de Metadados dos Fluxos de Trabalho (GMFT), o Gerente de Armazenamento de Dados em Memória (GADM) e o Gerente de Armazenamento Persistente (GAP). O GMFT é responsável por monitorar e gerenciar os dados recebidos e criados pelos fluxos de trabalho dos usuários. O GADM armazena em memória tanto dados de entrada e saída quanto intermediários. E, por fim, o GAP provê um armazenamento e gerenciamento dos dados persistentes.

Antes de executar o fluxo de trabalho, o usuário deve armazenar sua base de dados inicial no GAP, como bases de dados XML, utilizando uma API (*Application Programming Interface*) provida. A base de dados inicial é considerada como uma coleção de documentos, com a qual o usuário pode associar descrições de metadados. Esses documentos serão distribuídos nos vários nodos de armazenamento rodando servidores GAP. Por exemplo, todas as imagens que serão processadas pela aplicação biomédica de análise de imagens serão armazenadas em um banco de dados XML distribuído em diferentes nodos, e metadados como nome, tamanho e dimensão da imagem, podem ser associados a cada uma delas.

Uma abstração para leitura de dados, desenvolvida em outro trabalho [TTF+06], e baseada em consultas (*queries*) à base de dados inicial, retorna documentos para as instâncias do primeiro filtro no pipeline assim que eles estiverem prontos para processar dados. Desta forma, os usuários que desenvolvem aplicações para serem executadas pelo sistema não se preocupam com a leitura de dados. As aplicações devem ser desenvolvidas de tal forma que o primeiro filtro do pipeline leia dados de um *stream* de entrada assim como os demais filtros, e o sistema é que fica responsável por retornar documentos do GAP para ele.

Todos os dados de entrada e saída, assim como as mensagens intermediárias trocadas no pipeline são definidos por esquemas XML, e são chamados neste trabalho de *documentos*. A utilização do nome *documento* em vez de *mensagem* foi devido ao tamanho que eles podem alcançar, podendo ser da ordem de centenas de *megabytes*, por exemplo. Os documentos intermediários são armazenados nos servidores GAP para serem disponibilizados para inspeção ou para servirem como entrada para outros fluxos de trabalho. Esse armazenamento é realizado durante a comunicação entre os filtros do pipeline, e cada um dos três gerentes possui tarefas específicas que serão descritas nas próximas três seções..

Utilizar as próprias características (funcionalidades) dos sistemas de fluxo de trabalho científico como base para o projeto de mecanismos que os tornem mais

disponíveis é uma ótima idéia. Elas introduzem menos *overhead* e, se forem aperfeiçoadas, podem melhorar o desempenho das próprias funcionalidades desses sistemas. Em particular, os dados intermediários da execução dos fluxos serão utilizados como base para recuperar o sistema depois de uma falha. Para essa razão é que foi criado o Gerente de Armazenamento de Dados em Memória (GADM). Ele atua como um repositório de dados em memória, o qual provê os dados necessários para os fluxos de trabalho e também cuida do armazenamento dos dados intermediários e de saída na memória primária. Uma das suas características mais importantes é que ele não introduz nenhum sincronização entre a execução dos fluxos de trabalho e o processo de armazenamento de dados intermediários. Em outras palavras, os fluxos de trabalho podem executar, não havendo necessidade de parar durante o armazenamento dos dados.

A figura 4.1 mostra a arquitetura do sistema com os seus três componentes principais. A seguir, as fucionalidades de cada um deles, assim como os seus papéis no mecanismo para aumentar a disponibilidade do sistema, serão apresentados em mais detalhes.

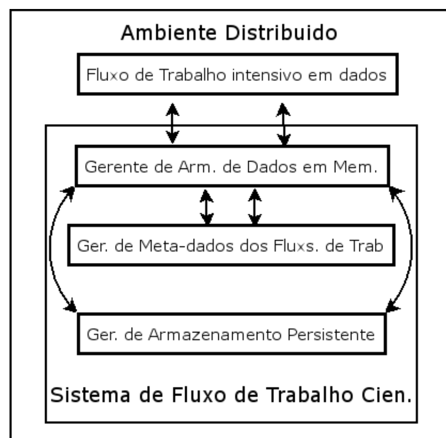


Figura 4.1: Organização dos Componentes do Sistema de Fluxo de Trabalho Científico.

4.2.1 Gerente de Metadados do Fluxos de Trabalho (GMFT)

Esse componente é desenvolvido como um filtro Anthill e é responsável pelo monitoramento e gerenciamento dos documentos recebidos e criados, sob demanda, pelos fluxos de trabalho dos usuários. Ele armazena todos os metadados relacionados a esses documentos. Por exemplo, para cada um deles, ele cria um identificador

único e também sabe onde ele está armazenado, qual instância de filtro o processou ou criou. Ele provê um protocolo para, basicamente, acessar, criar e atualizar os metadados durante a execução.

Para aumentar a disponibilidade do sistema, dois outros metadados são controlados por esse componente. O primeiro é a dependência de dados (descrita na seção 2.1.1). Ele se mantém informado de todas as dependências entre os documentos do sistema. O segundo é o estado dos documentos. Os documentos, durante a execução dos fluxos de trabalho, podem assumir três estados diferentes:

- **Não Processado**: Todo documento na base de dados inicial é considerado *"não processado"* no inicio da execução do fluxo de trabalho. Isso significa que eles estão disponíveis para serem processados.

- **Processando**: O documento está no estado *"processando"* quando um filtro no fluxo de trabalho o pega para processar. Todos os documentos criados durante a execução dos filtros dos fluxos de trabalho são considerados "processando", uma vez que eles foram criados e enviados para outro filtro para processar.

- **Processado**: Um documento é considerado *"processado"* quando o sistema reconhecer que não haverá necessidade de reprocessá-lo na ocorrência de uma falha. Para que esse reconhecimento seja feito, o sistema controla dependência entre os documentos.

Como foi descrito em [TTF+06], quando um fluxo de trabalho começa a executar, o GMFT decide quais documentos serão enviados para as instâncias de seus filtros. O escalonamento é realizado sob demanda e trabalha de forma *mestre-escravo*: sempre que os escravos (instâncias dos filtros do fluxo) estão disponíveis para processar, eles pedem por documentos para o mestre (o sistema, em particular o Gerente de Metadados dos Fluxos de Trabalho) que vai decidir quais documentos serão enviados. Essa decisão é baseada em quais documentos estão disponíveis para processar, e onde será o processamento. Se, por exemplo, existir um Gerente de Armazenamento Persistente rodando no mesmo nodo que uma instância do filtro, um documento *"não processado"* armazenado nesse nodo será escolhido.

4.2.2 Gerente de Armazenamento de Dados em Memória (GADM)

Esse componente é desenvolvido como um filtro Anthill e trabalha como um elo entre o fluxo de trabalho do usuário e o Gerente de Armazenamento Persistente

(GAP). Ele basicamente provê alguns mecanismos para ler/escrever documentos do/para o GAP.

Esse componente, na inicialização do sistema, é instanciado em vários nodos do ambiente distribuído. Cada uma de suas instâncias possui seu próprio limite de espaço de memória, o qual significa a quantidade máxima de memória que eles podem utilizar para armazenar os documentos. Quando o fluxo de trabalho do usuário é iniciado, as instâncias de seus filtros são "ligadas" às instâncias do GADM, utilizando o tipo de comunicação *labeled stream* do Anthill. Essa ligação é feita de tal forma que uma instância do GADM fica responsável por um conjunto de instâncias dos filtros do usuário. Por exemplo: ela fica responsável pelas instâncias que estão sendo executadas no mesmo nodo que ela.

Na execução dos fluxos de trabalho, quando uma instância de um filtro requisita um documento para processar, essa requisição é enviada à instância do Gerente de Armazenamento de Dados em Memória (GADM) a qual está responsável por aquela instância. Para saber qual documento deve ser lido, o GADM pergunta ao Gerente de Metadados dos Fluxos de Trabalho (GMFT). Baseada nos metadados providos pelo GMFT, ele realiza a leitura no Gerente de Armazenamento Persistente do documento necessário, armazena-o em memória, e envia-o para a instância do filtro da aplicação. Esse componente também armazena tanto os documentos intermediários enviados através dos *streams* do fluxo de trabalho quanto os de saída. É interessante observar que as instâncias do GADM também foram ligadas a instâncias do GMFT de tal forma que uma instância do GMFT fica responsável por um conjunto de instâncias do GADM.

Todos os documentos armazenados nesse componente possuem um identificador único criado pelo Gerente de Metadados dos Fluxos de Trabalho. Então, sempre que eles forem removidos ou tiverem suas informações atualizadas, isso é realizado baseado nesse identificador. Quando não há mais espaço para armazenar os documentos em uma instância do GADM, ela começa a enviar esses documentos para o Gerente de Armazenamento Persistente e a removê-los, utilizando uma política FIFO (*First In First Out*). Tanto para os documentos de saída quanto para os documentos intermediários no pipeline, ele é responsável por criar uma base de dados XML no Gerente de Armazenamento Persistente que vai armazenar esses documentos.

4.2.3 Gerente de Armazenamento Persistente (GAP)

O Gerente de Armazenamento Persistente é construído sobre o Mobius, em particular o componente Mako. Esse componente expõe e abstrai fontes de dados como XML e permite a instanciação de dados armazenados e o gerenciamento federativo

de armazenamentos distribuídos existentes. Ele controla múltiplas bases de dados distribuídas em múltiplos nodos.

Esse componente é utilizado para armazenar tanto os documentos de entrada e saída quanto os documentos intermediários dos fluxos de trabalho nos vários nodos. Esses documentos são armazenados em bases de dados distribuídas e definidos através de esquemas XML. Quando as instâncias dos filtros dos fluxos de trabalho criam um documento e o envia através dos *streams*, esse documento é primeiramente enviado para o Gerente de Armazenamento de Dados em Memória que realiza os cuidados necessários para armazenar esse documento no Gerente de Armazenamento Persistente. Bases de dados distribuídas são criadas sob demanda nesse componente para armazenar os estados intermediários. Uma base é criada para cada *stream* de saída.

O Mobius disponibiliza uma interface gráfica na qual as informações dos documentos armazenados nas bases de dados podem ser vistas. Uma dessas informações é onde o documento está armazenado. Dessa forma, quando os usuários desejam fazer a inspeção dos documentos intermediários, eles verificam essa informação e abrem os documentos para serem examinados.

4.3 Aplicação Exemplo: Análise de Placentas de Rato

A aplicação utilizada como exemplo neste trabalho utiliza imagens de microscópios de alta resolução para estudar mudanças no fenótipo da placenta do rato induzido por manipulações genéticas. Em particular, ela foca na segmentação das imagens que compõem uma placenta 3D do rato em regiões correspondendo a três camadas de tecido: labirinto, spongiotrophoblasto e glicogênio, como foi descrito em [PH05].

As imagens microscópicas são obtidas de slides coloridos de forma padrão do ponto de vista histológico de ambas as placentas de rato naturais e as que sofreram mutação. Os slides são coletados pelo corte de placentas cobertas de cera com uma densidade de 3μ. Depois elas são digitalizadas, utilizando-se um digitalizador Aperio Acanscope com lente objetiva com magnitude de 20x. Dessa forma, uma placenta equivale a um conjunto de várias imagens.

A aplicação é composta de quatro operações principais de análise de imagens: separação do primeiro plano e do plano de fundo, normalização do histograma, classificação das cores, e segmentação do tecido. Cada uma delas foi mapeada para

um filtro Anthill, e elas foram organizadas como na figura 1.1, para formar um fluxo de trabalho. A descrição básica de cada uma dessas operações é a seguinte:

- Separação do primeiro plano e do plano de fundo (PP/PF): as imagens são convertidas de RGB - Vermelha (**R**ed), Verde (**G**reen) and Azul (**B**lue)) - para CMYK - Ciano (**C**yan), Magenta (**M**agenta), Amarelo (**Y**ellow), e Preto (blac**K**) -, e uma combinação dos canais das cores passa por um limiar para se obter o tecido do primeiro plano. Como saída dessa conversão, é gerada uma máscara para cada uma das imagens.

- Normalização do histograma: anteriormente ao processamento da placenta inteira, as imagens precisam individualmente ser corrigidas das variações das cores. Essa correção é necessária porque elas apresentam variações nas cores devido a largura dos slides, ao tempo de exposição e à concentração de coloração. Isso consiste em três sub-operações. Primeiro: calcula-se a média das cores para as imagens, e então para toda a placenta. Depois seleciona-se uma imagem como alvo de normalização das cores. E, por fim, gera-se um histograma para cada um dos canais (vermelho, azul e verde), e então as imagens são corrigidas utilizando-se esse histograma.

- Classificação das cores: os pixels das imagens são classificados, utilizando-se um classificador Bayesiano, em um de 8 tipos diferentes: núcleo escuro, núcleo com nível de intensidade médio, núcleo claro, núcleo extraclaro, célula de sangue vermelha, citoplasma claro, citoplasma escuro e fundo da imagem.

- Segmentação do Tecido: Finalmente, também utilizando-se um classificador Bayesiano, os tecidos são classificados nos 3 tipos: labirinto, spongiotrophoblasto e glicogênio.

Devido à necessidade de interações humanas, essa aplicação deve ser executada em três estágios diferentes. O primeiro é composto pelo filtro PP/PF, o segundo pelo filtro de normalização do histograma, e o último pelos filtros de classificação das cores e segmentação do tecido. Para que os estágios 2 e 3 sejam executados, é necessário que os estágios 1 e 2, respectivamente, sejam completados, e a interação humana tenha ocorrido. A divisão da aplicação em estágios é possível devido ao fato de o sistema de fluxo de trabalho armazenar os dados de saída de cada estágio e utilizá-lo como entrada para o próximo, como podemos verificar na figura 4.2.

Primeiramente o sistema de fluxo de trabalho científico envia os documentos da base de dados inicial para o filtro PP/PF, e à medida que esse filtro processa

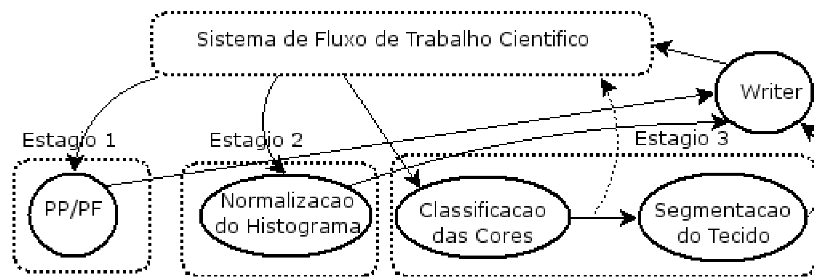


Figura 4.2: Fluxo de trabalho da aplicação dividido em estágios.

os dados, eles são enviados para o filtro *escritor*, através de um *stream* de saída, fazendo com que os documentos sejam salvos no sistema. Essa mesma situação ocorre com os próximos dois estágios da aplicação, uma vez que os documentos iniciais para esses dois estágios foram armazenados no sistema. Como podemos perceber, entre os filtros Classificação das Cores e Segmentação do Tecido, os documentos intermediários são armazenados diretamente no sistema, não havendo necessidade do filtro escritor. Esse armazenamento é realizado de forma transparente a aplicação e descrita em maiores detalhes no capítulo 5.

A figura 4.3 mostra um exemplo da execução do fluxo de trabalho dessa aplicação quando várias instâncias dos seus filtros são criadas. Todo o processamento e a comunicação é realizada por/entre essas diversas instâncias. É interessante observar que a criação das instâncias é realizada nos vários nodos do ambiente distribuído, o que não está representado nessa figura.

4.3.1 Sumário

Nesse capítulo foram discutidos os requisitos necessários dos sistemas de fluxo de trabalho cienítico que foram cobertos neste trabalho. A arquitetura descrita é de extrema importância para introduzirmos o mecanismo para aumentar sua disponibilidade. No próximo capítulo apresentamos como esse sistema foi desenvolvido assim como o mecanismo proposto neste trabalho, detalhando o seu funcionamento.

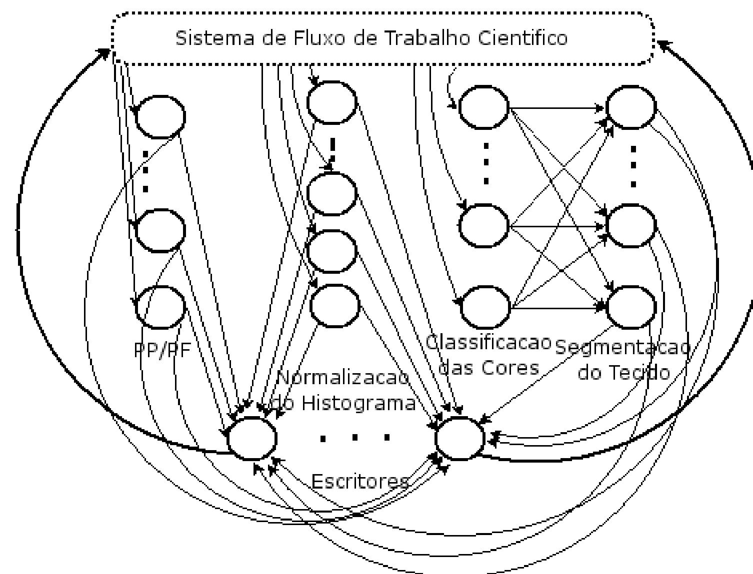


Figura 4.3: Instâncias dos filtros durante a execução do fluxo de trabalho da aplicação exemplo.

Capítulo 5

Implementação

O sistema de fluxo de trabalho científico, apresentado na seção 4.2, é constituído de 3 componentes principais (o Gerente de Metadados dos Fluxos de Trabalho, o Gerente de Armazenamento de Dados em Memória e o Gerente de Armazenamento de Dados Persistente), os quais foram desenvolvidos utilizando-se os *frameworks* Anthill e Mobius. A figura 5.1 apresenta as entidades presentes na execução dos fluxos de trabalho assim como as camadas presentes em cada uma delas.

Como podemos perceber, os componentes GMFT e GADM são desenvolvidos sob o Anthill, o qual utiliza a biblioteca de computação paralela PVM (*Parallel Virtual Machine*), para instanciar e realizar a comunicação das instâncias dos filtros. O Gerente do Anthill é o responsável por instanciar as cópias dos filtros de acordo com a especificação feita no arquivo XML de configuração do Anthill. Quando uma falta ocorre no sistema, é esse gerente o responsável por reiniciar as instâncias dos filtros necessárias, e notificar às outras a ocorrência dessa falta.

Para que uma aplicação desenvolvida no Anthill seja executada como um fluxo de trabalho, é necessário que ela utilize o suporte a sistema de fluxo de trabalho científico do Anthill. Esse suporte, de forma transparente à aplicação, realiza chamadas a uma API (*Application Programming Interface*) desenvolvida para esse trabalho. Essa API cuida da comunicação do fluxo de trabalho da aplicação com o Gerente de Armazenamento de Dados em Memória, provendo todas as funcionalidades do sistema descrito na seção 4.2. Essa API é mostrada na figura 5.1 como API SFTC (Sistema de Fluxo de Trabalho Científico), e o seu funcionamento é mostrado na seção 5.2.2.

Neste capítulo descrevemos primeiramente na seção 5.1 os dados armazenados pelos três componentes do sistema de fluxo de trabalho proposto nesse trabalho assim como as principais operações realizadas por eles. Na seção 5.2 descrevemos o modelo de faltas e o método de tolerância a faltas adotados neste trabalho assim

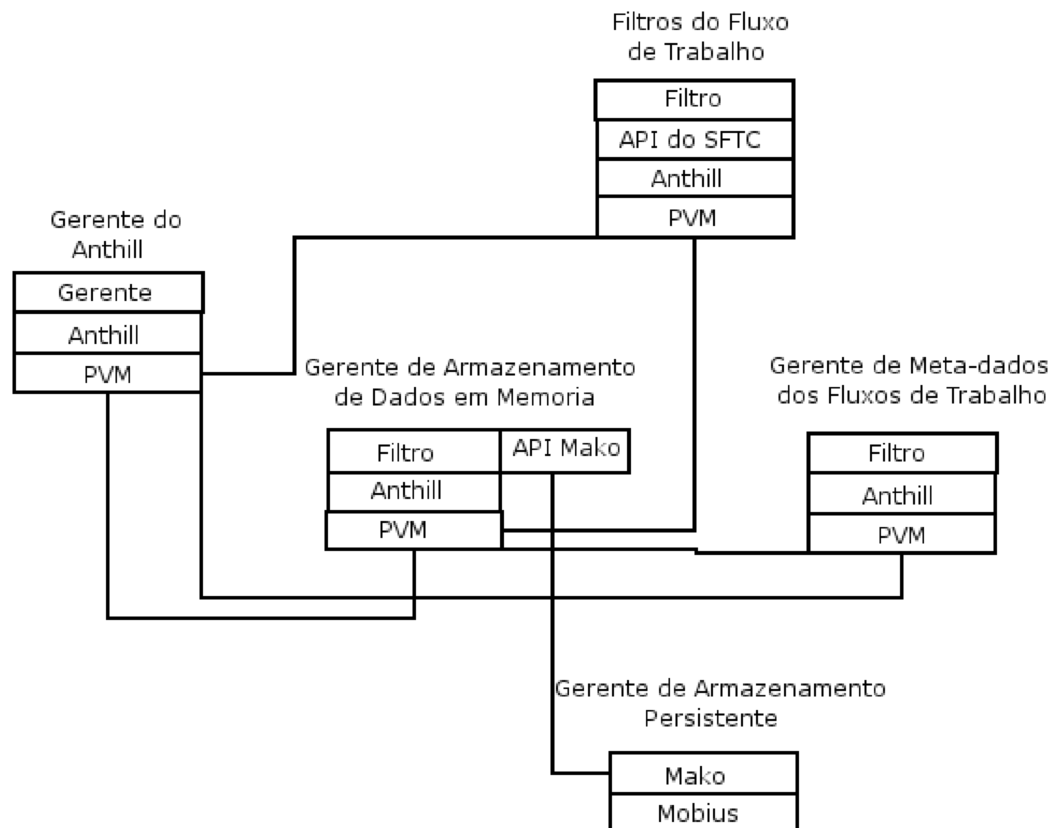


Figura 5.1: Camadas das entidades presentes na execução dos fluxos de trabalho.

como os protocolos utilizados pelos componentes do sistema para aumentar a sua disponibilidade para realizar a recuperação de uma falta.

5.1 Dados e Operações dos Componentes do Sistema

Nesta seção, descrevemos os dados armazenados por cada um dos componentes, assim como as operações por eles realizadas.

5.1.0.1 Gerente de Metadados dos Fluxos de Trabalho (GMFT)

Para realizar o monitoramento e o gerenciamento dos documentos recebidos e criados sob demanda pelos fluxos de trabalho, esse componente armazena uma lista de metadados dos documentos para cada tipo de *stream* de entrada dos filtros dos fluxos de trabalho. Os metadados desses documentos são:

- O tipo (nome) de *stream* de entrada relacionado a cada documento.

- Se o documento está armazenado no Gerente de Armazenamento de Dados em Memória (GADM).
- O identificador desse documento no GADM.
- O estado do documento (*não processado*, *processando* ou *processado*).
- Se o documento está armazenado no Gerente de Armazenamento Persistente (GAP).
- O identificador desse documento no GAP.
- Nome do nodo do GADM e do GAP em que o documento está armazenado.
- Nome da coleção do GAP em que o documento está armazenado.
- Dependências do documento.
- Nome dos filtros e número de sua instância que processou o documento.

Esses metadados são criados para cada documento presente na base de dados inicial, passada pelo usuário, para execução do fluxo de trabalho. Durante a execução dos fluxos de trabalho, quando um documento é criado e enviado por um *stream* de saída, esses mesmos metadados são criados com a ajuda do Gerente de Armazenamento de Dados em Memória, uma vez que este último é que recebe os documentos para serem armazenados.

O Gerente de Metadados dos Fluxos de Trabalho disponibiliza um protocolo de comunicação, utilizado pelo GADM, para realizar basicamente as seguintes operações:

- Retornar um identificador de um documento para ser lido, baseado no *stream* de entrada, nome e instância do filtro que vai processá-lo.
- Criar metadados de um documento que será armazenado no GADM.
- Atualizar os metadados de um documento que será armazenado no GAP.
- Atualizar o estado de um documento para *processado*.
- Retornar os estados dos documentos.
- Realizar *rollback*, se uma falta ocorreu em uma instância do GADM (mais detalhes serão descritos na seção 5.2.3.3).

5.1.0.2 Gerente de Armazenamento de Dados em Memória (GADM)

Esse componente possui uma lista de documentos acessada a partir do identificador único do documento. Ele armazena em memória tanto os documentos lidos quanto os documentos enviados pelos *streams* de saída dos fluxos de trabalho. Para cada documento são armazenados metadados que o descrevem. Além dos mesmos metadados armazenados no Gerente de Metadados dos Fluxos de Trabalho, esse componente também armazena o XML fornecido pelo usuário que descreve o documento que é armazenado no Gerente de Armazenamento Persistente.

O protocolo disponibilizado por esse componente é utilizado pelo Anthill quando ele está o suporte a sistema de fluxo de trabalho científico é usado. Basicamente, as operações disponíveis são:

- Ler documento. Nessa operação, esse componente pede ao Gerente de Metadados dos Fluxos de Trabalho um documento que ainda não foi processado. Se esse documento não estiver armazenado no GADM, ele lê esse documento do Gerente de Armazenamento Persistente e retorna a instância do filtro que pediu.

- Escrever documento. Recebe o documento que está sendo gerado pelo *stream* do fluxo de trabalho. Avisa ao GMFT que um novo documento está sendo criado e recebe o identificador criado por ele. Coloca como dependência desse documento os documentos que foram lidos anteriormente e foram utilizados para o criar. E, por fim, armazena esse documento em memória.

- Retornar às dependências de um documento.

5.1.0.3 Gerente de Armazenamento Persistente (GAP)

O Gerente de Armazenamento Persistente utiliza o serviço Mako do Mobius para realizar o armazenamento persistente dos documentos criados pela execução dos fluxos de trabalho. O Mako, como foi descrito na seção 2.2.1, fornece um protocolo para submeter, atualizar, remover e retornar documentos. O Gerente de Armazenamento de Dados em Memória utiliza uma API (mostrada na figura 5.1 como API Mako) desenvolvida na linguagem C utilizando *sockets* para preparar os pacotes de acordo com o protocolo Mako, e enviá-los para o nodo necessário do ambiente distribuído que esteja executando esse serviço.

5.2 Aumento da Disponibilidade do Sistema

Primeiramente, na seção 5.2.1, apresentamos o modelo de faltas e o método de tolerância a faltas utilizados para aumentar a disponibilidade do sistema. Na seção 5.2.2 descrevemos o protocolo de controle utilizado para criar um estado consistente do sistema, mesmo na presença de falhas. O estado consistente do sistema para esse trabalho em específico é o conjunto de todos os documentos processados pela aplicação, de tal forma que esse conjunto deve ser o mesmo no final da execução da aplicação quando ocorrem ou não faltas no sistema. Por fim, na seção 5.2.3, apresentamos os protocolos de recuperação de cada componente.

Para garantirmos a transparência do mecanismo que aumenta a disponibilidade do sistema para os fluxos de trabalho, assumimos que todo documento enviado em um *stream* de saída é dependente dos documentos que acabaram de ser recebidos antes de ele ser gerado. Essa dependência é controlada de forma que um filtro não pode enviar, a partir de um mesmo documento de entrada, vários documentos para um ou mais *streams* de saída. Dessa forma, essa implementação não é capaz de realizar a recuperação de filtros que apresentam dependência de dados (descrita na seção 2.1.1) do tipo ***n para m*** e ***1 para n***.

5.2.1 Modelo de Faltas e Método de Tolerância a Faltas Adotados

O sistema é constituído de um conjunto de nodos de processamento que se comunicam utilizando uma rede de alta velocidade. Nesses nodos, os processos (tanto os dos fluxos de trabalho quanto os do sistema de fluxo de trabalho) são executados, e é assumido que eles são processos resilientes cuja execução, quando repetida gerará o mesmo resultado. Já quanto aos canais de comunicação, assumimos que são canais FIFO confiáveis, os quais garantem a entrega de mensagens entre os nodos.

Nesse sistema, tanto os nodos quanto os canais podem falhar a qualquer momento. Eles seguem o modelo de faltas de queda. Devido ao fato de o PVM [Sun90] ser utilizado como base para o Anthill, e conseqüentemente para o sistema apresentado neste trabalho, resolvemos utilizar os mecanismos de detecção de faltas do PVM para detectar faltas que ocorrem nas instâncias dos filtros da aplicação, no Gerente de Meta Dados dos Fluxos de Trabalho e no Gerente de Armazenamento de Dados em Memória. Os canais de comunicação do PVM possuem retransmissão de mensagens e também garantem a ordem FIFO na entrega das mensagens, garantindo um comportamento segundo o modelo de faltas de parada. Quando uma falta ocorre em um canal de comunicação de algum nodo do sistema, de tal forma

que ele não responde mais a requisições, através de temporizadores, assumimos que houve falha no nodo, garantindo o modelo de faltas de queda.

A escolha do modelo de faltas de queda para os componentes do sistema é devido ao fato de esse modelo abranger grande parte das faltas encontradas nos sistemas distribuídos da atualidade. Ele também apresenta faltas que podem ser detectadas sem exigir um esforço quase que fora da realidade como é o caso do modelo bizantino, uma vez que a detecção de faltas não é o objetivo deste trabalho. Quando um processo pára de responder, pode-se assumir que ele falhou. Nesse caso, falsos positivos podem ocorrer, mas o sistema estará preparado para se recuperar. As faltas de queda de nodos são mapeadas para faltas de processo, ou seja, quando um nodo falhar, é assumido que todos os processos que estavam ali executando falharam.

Como o Gerente de Armazenamento Persistente não é construído sob o PVM, sua detecção de faltas é realizada utilizando-se uma técnica popular de monitoramento de *heartbeat*. Nessa técnica, os processos são organizados em um estrutura particular, e periodicamente eles enviam mensagens para o processo para o qual eles apontam. Se um processo não receber n mensagens consecutivas do predecessor, ele assume que o predecessor falhou [NKB+03]. Atualmente, o Gerente de Armazenamento de Dados em Memória fica responsável por fazer essa detecção durante a sua comunicação com o Gerente de Armazenamento Persistente (GAP). Uma vez detectada uma falha no GAP, esse componente é reiniciado.

Uma primeira restrição imposta ao projeto deste trabalho é que o nodo onde o gerente do Anthill executa não pode falhar. Essa restrição é devido ao fato de o gerente ser o responsável por dar início ao processo de recuperação do sistema. Se compararmos o número de processos (instâncias dos filtros dos fluxos de trabalho) que estão presentes na execução dos fluxos de trabalho que podem ser da ordem de centenas ou milhares, vamos perceber que a probabilidade de eles falharem é bem maior do que a do gerente. Uma forma de acabar com essa restrição seria a utilização de redundância de hardware [Tre04], entretanto detalhes serão omitidos por estarem fora do escopo deste trabalho.

Quando o sistema de fluxo de trabalho sofre uma falta durante a recuperação de uma anterior, o sistema todo é reiniciado e a sua execução anterior à primeira falha não é aproveitada. Dessa forma, assumimos que uma falta não vai acontecer durante a recuperação de uma outra anterior, ou seja, o mecanismo que aumenta a disponibilidade do sistema de fluxo de trabalho não consegue se recuperar de uma segunda falha, enquanto ele já estiver se recuperando de uma primeira. Se considerarmos que o período da execução dos fluxos de trabalho é de ordem de grandeza maior do que o período da recuperação da falta, raramente o reinício do

sistema vai acontecer, e o desempenho dos fluxos de trabalho será atrapalhado.

O método para minimizar a reexecução dos fluxos de trabalho após uma falta, incorporado ao sistema de fluxo de trabalho científico, foi escolhido baseado nas características desse tipo de sistema. Se empregássemos métodos como *checkpoint* ou replicação ativa separadamente, sem levar em consideração suas características, poderíamos introduzir um *overhead* muito grande no sistema. O método empregado pode ser dividido em duas partes: a primeira relacionada aos fluxos de trabalho (Parte 1); e a segunda, aos componentes do sistema de fluxo de trabalho (Parte 2):

- Parte 1: Para garantir que os dados necessários para a recuperação dos fluxos de trabalho fossem armazenados, resolvemos utilizar um método baseado em dependência de dados que emprega alguns dos conceitos de *checkpoint* e *message logging*. Os documentos intermediários dos fluxos de trabalho, os mesmos utilizados para inspeção pelos usuários, são armazenados assim como os metadados da execução que o descrevem, por exemplo, seus documentos dependentes, seus locais de execução etc. Baseado nesses metadados, podemos criar um estado consistente do sistema e, quando necessário recuperá-lo. Na recuperação os documentos armazenados, quando não foram processados totalmente, podem ser novamente enviados para os fluxos de trabalho para serem processados, ou então, quando suas dependências forem perdidas, para serem utilizados para recriá-las.

- Parte 2: Para garantir que uma falta que ocorre no Gerente de Armazenamento de Dados em Memória (GADM) ou no Gerente de Metadados dos Fluxos de Trabalho (GMFT) não ocasione a perda dos metadados desses componentes, um mecanismo de replicação desses dados é utilizado. Quando os metadados dos documentos são criados em um deles, o outro será notificado e terá esses mesmos dados armazenados. Dessa forma, quando uma falta ocorrer em algum deles, o outro estará preparado para o ajudar a reconstruir o seu estado.

Uma característica interessante desse método proposto é que ele pode ser um primeiro passo para expandir o modelo de faltas para tratar faltas bizantinas. Hoje em dia os dados intermediários são salvos para aumentar a confiança do sistema e para a inspeção manual pelos usuários. Se considerarmos que essa inspeção pode ser realizada automaticamente, através de testes de aceitação, faltas causadas por corrupção de dados, defeitos de *hardware* ou *software* poderiam ser tratadas.

Uma premissa do atual modelo de programação é que não consideramos os estados criados durante a execução dos filtros como necessários para processar diferentes documentos. Hoje em dia estamos preocupados em conseguirmos recuperar o sistema

de tal forma que todos os documentos que tenham que ser processados realmente o sejam. Dessa forma, uma segunda extensão interessante seria a realização de *check-points* do estado da aplicação. Essa extensão poderia ser feita enviando, junto com os documentos de saída enviados pelos *streams*, o próprio estado da aplicação. Dessa forma, quando fôssemos recuperar o sistema, recuperaríamos também o seu estado.

Como foi dito anteriormente, uma vez detectada uma falha no GAP, esse componente é reiniciado. Entretanto não existe nenhum mecanismo capaz de detectar se houve ou não perda de dados, uma vez que o Mobius, utilizado como base para o GAP, não possui nenhum mecanismo de replicação de dados e recuperação. Esses mecanismos poderiam ser conseguidos, se o Mobius utilizasse, por exemplo, banco de dados distribuídos confiáveis, o que está fora do contexto deste trabalho. Se tais mecanismos existissem, uma conseqüência direta seria o aumento da confiança do Gerente de Armazenamento Persistente.

5.2.2 Protocolo de Controle

Usualmente, aplicações que utilizam o paradigma *filter-stream* são implementadas como um conjunto de filtros os quais possuem um formato do tipo descrito no algoritmo 1. Eles normalmente são alimentados por uma fonte de dados, por exemplo, uma base de dados para o primeiro filtro no *pipeline* e um *stream* de entrada para os demais filtros. Eles possuem uma função de processamento principal que realiza algum tipo de computação sobre os dados recebidos. Quando necessário, eles enviam dados de saída para o próximo filtro no *pipeline* ou criam bases de dados de saída.

```
while existir dado para ser processado do
    dado ← Leia();
    Processe(dato);
    if existir algum dado de saída then
        Escreva(dado);

end
```

Algoritmo 1: Filtro da Aplicação

Uma vez que o usuário quer utilizar o sistema de fluxo de trabalho científico, ele precisa implementar sua aplicação utilizando as ferramentas do Anthill, e toda a comunicação com o sistema é realizada transparentemente. Nas figuras 5.2 e 5.3, são mostrados exemplos do terceiro estágio do fluxo de trabalho da figura 4.2, nos quais uma instância do filtro Classificação das Cores lê dados do sistema e envia o resultado

para outra instância do filtro Segmentação do Tecido, respectivamente. Enquanto estivermos explicando esse exemplo em particular, sempre que citarmos tanto o Gerente de Armazenamento de Dados em Memória (GADM) quanto o Gerente de Metadados dos Fluxos de Trabalho (GMFT) estaremos falando em nível de instância. Nesse exemplo, como podemos ver em ambas as figuras 5.2 e 5.3, as instâncias 1 e 2 do GADM estão responsáveis, respectivamente, pelas instâncias dos filtros Classificação das Cores e Segmentação do Tecido. Do mesmo modo, as instâncias 1 e 2 do GMFT estão responsáveis, respectivamente, pelas instâncias 1 e 2 do GADM.

Quando a instância do primeiro filtro do fluxo de trabalho está pronta para processar um documento, o sistema é notificado por uma mensagem do tipo LER_DOC (mostrada na figura 5.2), e o GADM irá tentar retornar um documento. Esse componente pede, através da mensagem RET_DOC_NAO_PROCESSADO, ao GMFT por um documento relacionado ao fluxo de entrada do filtro Classificação das Cores. O GMFT vai procurar por um documento "*não processado*" e retornará (mensagem DOC_ID) o meta-dado desse documento para o GADM. Utilizando esse meta-dado, o GADM é capaz de retornar esse documento do Gerente de Armazenamento Persistente (GAP), e então enviá-lo (mensagem DOC_DISPONIVEL) para o filtro.

Depois que a instância do filtro Classificação das Cores processar o documento que acabou de ler, ela pode enviar o resultado desse processamento para o filtro Segmentação do Tecido. Com esse documento de saída, suas dependências também serão enviadas. Elas serão necessárias mais tarde para atualizar os documentos dependentes para "*processado*", o que significa que eles foram processados e a saída relacionada a eles foi gerada. Então as mensagens 1 (PEGA_DEPS) e 2 (RET_DEPS_DOC) na figura 5.3 são responsáveis por pedir ao GADM as dependências do documento e recebê-las, respectivamente. O GADM cria uma lista de documentos lidos para cada instância dos filtros da aplicação que ele está tratando. Sempre que uma instância do filtro Classificação das Cores pedir pelas dependências de um documento, o GADM retornará essa lista armazenada e ao mesmo tempo "apagará" essa lista. Quando um novo documento for lido, uma nova lista será criada. Dessa forma, os documentos aparecem na lista de dependência de apenas um outro documento.

Quando a instância do filtro Segmentação do Tecido, na figura 5.3, recebe um documento através do seu fluxo de entrada, o sistema cuida de armazenar esse documento e também de atualizar suas dependências para "*processado*". Com a mensagem ESCREVE_DOC, o documento e suas dependências são enviadas para o GADM. A primeira ação tomada por esse componente é armazenar o documento na memória. Se não existir espaço para armazenar esse documento, ele

envia alguns documentos para o GAP, seguindo uma ordem FIFO, e então o armazena. Para realizar o armazenamento do documento, o GADM precisa de um identificador único no sistema, e para isso há necessidade de pedi-lo ao GMFT. Para garantir que esse identificador seja criado em ambos os gerentes (GADM e GMFT), mesmo na presença de faltas, foi utilizado o protocolo *Two Phase Commit* [Gra78, Lam80]. Esse protocolo se dá pelas mensagens de 4 a 7 na figura 5.3. Na mensagem INICIO_NOVO_DOC_NO_GADM, o GADM notifica ao GMFT desse novo documento. Um identificador é criado para ele e é retornado na mensagem NOVO_ID_DOC. Para completar o protocolo, as mensagens 6 e 7 também são trocadas. Dessa forma, tanto o GADM quanto o GMFT sabem que o documento foi armazenado no GADM com o identificador correto, mesmo na ocorrência de faltas.

A próxima ação a ser tomada pelo GADM é atualizar as dependências do novo documento para "*processado*" no GMFT. Essa ação é realizada na mensagem 8 (DOC_PROCESSADO). A mensagem 9 (DOC_PROCESSADO_ACK) é enviada de volta para o GADM para ter certeza de que o GMFT recebeu e atualizou o estado dos documentos.

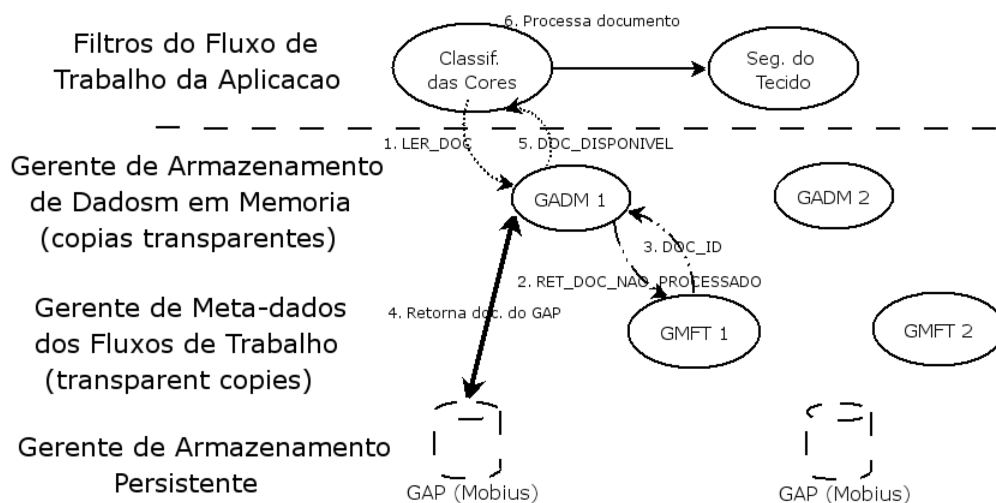


Figura 5.2: Comunicação entre os componente quando o filtro Classificação das Cores lê um documento.

5.2.3 Mecanismo de Recuperação

Agora serão descritos os algoritmos utilizados para recuperar os três diferentes pontos de falhas do sistema tratados nesse trabalho: filtros dos fluxos de trabalho dos usuários, Gerente de Armazenamento de Dados em Memória e Gerente de Metadados

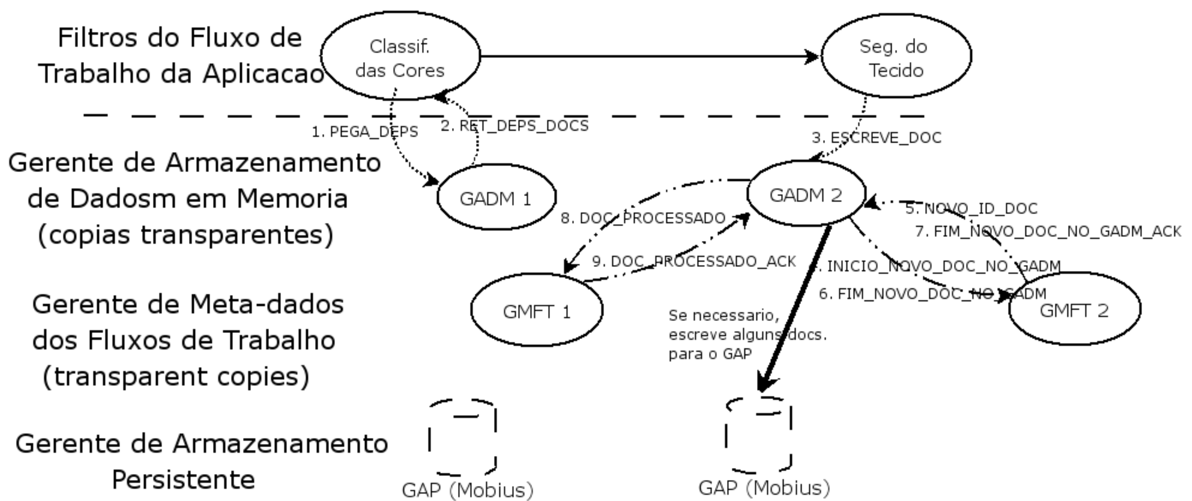


Figura 5.3: Comunicação entre os componentes quando o filtro Classificação das Cores envia um documento para o filtro Segmentação do Tecido.

dos Fluxos de Trabalho. É interessante observar que nenhum desses algoritmos são centralizadores, ou seja, todos são distribuídos.

5.2.3.1 Recuperação dos Filtros dos Fluxos de Trabalho

Quando uma falta ocorre em algum dos filtros da aplicação, o Gerente do Anthill cuida de interromper a execução de todas as instâncias dos filtros dos fluxos de trabalho que estavam executando, e reiniciá-las. É possível realizar essa situação porque assumimos que a execução dos filtros pode ser reiniciada utilizando-se os documentos armazenados e suas dependências. O dano no tempo de execução dos fluxos causado pela falta vai depender de quantos documentos terão que ser reexecutados porque suas execuções foram interrompidas ou eles não foram processados (tiveram seus estados como "processados") antes da falha.

A figura 5.4 apresenta uma situação em que K e V documentos foram gerados pelos filtros A e B, respectivamente, mas não foram completamente processados pelos filtros B e C, antes da falta. Na recuperação, esses documentos estão armazenados no sistema (no Gerente de Armazenamento de Dados em Memória ou no Gerente de Armazenamento Persistente) e precisam ser processados antes que a execução do fluxo acabe. Durante a recuperação, três possibilidades poderiam ser tomadas para reexecutar esses documentos. Uma possibilidade seria a reexecução dos documentos no fim da execução do fluxo. Por exemplo, na figura 5.4, os M documentos da base de dados inicial seriam executados antes dos outros K e V. Uma segunda abordagem seria executar primeiro os K e V documentos, e então

começar executando os M documentos. Por fim, uma última seria executar todos os documentos ao mesmo tempo. Essa última abordagem foi escolhida para ser implementada porque, assumindo que o *pipeline* esteja bem balanceado, o processamento de K e V não causaria o travamento do filtro A quando ele começasse a processar os M documentos e enviasse os resultados para B.

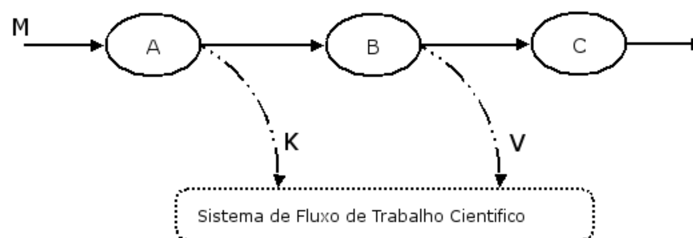


Figura 5.4: Exemplo da recuperação do sistema após uma falha. Existem M documentos para serem processados da base de dados inicial, e K e V documentos dos fluxos intermediários.

Quando os filtros dos fluxos de trabalho são reiniciados após a falta, os primeiros filtros do *pipeline* vão agir como eles costumavam fazer anteriormente, pedindo ao sistema documentos disponíveis, ou seja, com estado "não processado". Entretanto, os outros filtros terão que perguntar ao sistema se existe algum documento que não foi processado antes da falha e está armazenado. Se existirem N documentos, por exemplo, eles vão pegar os próximos N documentos do sistema. Na figura 5.5 é mostrado um exemplo de quando essa situação acontece. Nesse exemplo, inicialmente, o filtro Classificação das Cores lê o documento 1 e envia o resultado do seu processamento (documento $1'$) para o filtro Segmentação do Tecido. Depois essa mesma situação acontece com um documento diferente. O documento 2 é lido, processado, e o seu resultado (documento $2'$) é enviado para o próximo filtro. Quando esse filtro recebe o documento $2'$, uma falta ocorre em alguma instância do filtro da aplicação. Nessa situação, os documentos 1, 2 e $1'$ estão processados, e o documento $2'$ não está, mas está armazenado no sistema, em particular no GADM. Dessa forma, quando os filtros são reiniciados, não existe necessidade de processar os documentos 1 e 2 novamente. O filtro Segmentação do Tecido vai primeiramente ler o documento $2'$ do sistema, processá-lo, e então continuar lendo do seu fluxo de entrada.

5.2.3.2 Recuperação do Gerente de Armazenamento de Dados em Memória

Quando uma falta ocorre no sistema, o Gerente do Anthill é responsável por detectá-la e reiniciar, se existir, a instância do Gerente de Armazenamento de Dados

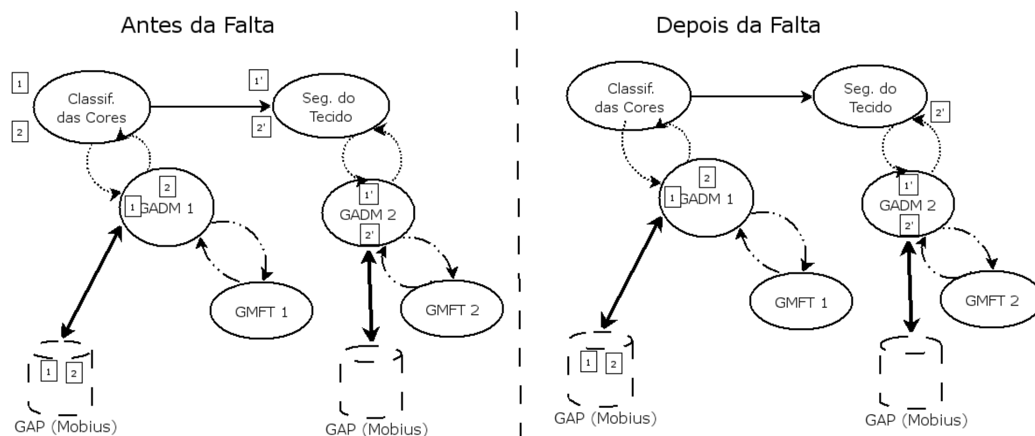


Figura 5.5: Exemplo de recuperação dos filtros do fluxo de trabalho da aplicação biomédica.

em Memória (GADM) que falhou. Para as demais instâncias que não falharam, ele envia uma mensagem notificando a falta. Quando elas recebem essa notificação, elas param imediatamente de realizar o seu trabalho, gravam o estado em que elas estavam quando receberam a mensagem, e esperam que o sistema seja reiniciado.

Após o reinicio do sistema, as instâncias do GADM precisam executar o seu protocolo de recuperação antes de voltar a atender as requisições dos filtros dos fluxos de trabalho. Quando elas se recuperam de uma falta, sua recuperação vai depender do fato de a falta ter acontecido ou não nela, ou seja, do fato de os documentos que estavam nela anteriormente armazenados terem sido perdidos ou não.

Quando as instâncias do GADM que não falharam estão executando o seu algoritmo de recuperação, elas primeiramente têm que ter certeza de que o trabalho que elas estavam realizando anteriormente à falta foi completado. Através do estado salvo logo após o recebimento da notificação da falta, elas conseguem verificar se algumas "transações" como as 5, 6 ou 7 do protocolo de consenso ou as 8 e 9, todas apresentadas na figura 5.3, não foram completadas e precisam ser executadas.

Durante a execução dos fluxos de trabalho, não é garantido que o estado dos documentos (*"não processado"*, *"processando"* e *"processado"*) está armazenado na mesma instância em que os próprios documentos realmente foram armazenados. Por exemplo, vamos supor que nas figuras 5.3 e 5.2, um documento A fosse lido pelo sistema e fosse armazenado na instância 1 do GADM. Nessa instância, ele possui seu estado criado como *"processando"*. Quando esse documento for processado e o filtro gerar uma saída, vamos supor o documento A', essa saída é enviada para o próximo filtro do *pipeline*. Antes de esse filtro processar A', ele é armazenado na instância 2 do GADM, a qual cuida de criar um estado para o documento A

e atualizá-lo para *"processado"*. Dessa forma, depois que as transações pendentes são executadas, as instâncias do GADM perguntam ao Gerente de Metadados dos Fluxos de Trabalho (GMFT) pelos estados dos documentos armazenados nele. Isso é feito para garantir que, no início da execução, depois de uma falta ou não, todos os documentos armazenados no GADM terão o estado correto.

Se a falta ocorrer em alguma instância do filtro GADM, quando ela é reiniciada, ela tem que notificar ao GMFT que os documentos que estavam armazenados somente nela foram perdidos e que ele terá que realizar um *rollback* baseado nas dependências dos documentos. Dessa forma, para todos os documentos perdidos, o GMFT atualiza suas dependências para *"não processada"*, de tal forma que elas ficarão disponíveis para que os filtros dos fluxos de trabalho as processem.

Como podemos perceber, quando uma falta ocorre no sistema e o GADM está realizando a leitura de um documento, ou seja, está executando o protocolo descrito na figura 5.2, não existe nenhum caso especial na sua recuperação que deva ser tratado. Nesse caso, como o GMFT na sua recuperação vai atualizar o estado de todos os documentos que estavam "processando"para "não processado", os documentos que estavam sendo lidos no momento da falta serão lidos novamente depois da recuperação. Caso alguma instância do GADM tenha esse documento armazenado em memória, não haverá necessidade de o ler novamente do GAP, como no último exemplo mostrado na seção 5.2.3.1.

5.2.3.3 Recuperação do Gerente de Metadados dos Fluxos de Trabalho

Assim como ocorre com o Gerente de Armazenamento de Dados em Memória (GADM), o Gerente do Anthill é responsável por reiniciar, se existir, a instância do Gerente de Metadados dos Fluxos de Trabalho (GMFT) que falhou. O seu mecanismo de recuperação também vai depender de a falta ter ocorrido ou não em alguma das suas instâncias.

Quando uma instância do GMFT que falhou é reiniciada, ela primeiramente vai reconstruir o seu estado. A primeira ação tomada é criar o estado para os documentos presentes na consulta de entrada por que o usuário passou como base de dados de entrada. A partir desse estado inicial, ela pergunta às instâncias do GADM, relacionadas a ela, pelos estados dos documentos que já foram armazenados por elas ou que ainda estão armazenados. Por fim, ela pergunta a todas as instâncias do GADM no sistema pelos documentos que estão *"processados"*. Dessa forma ela consegue ter seu estado recriado.

Os próximos passos da recuperação das instâncias do GMFT são similares tanto para as instâncias que falharam quanto para as que não falharam. O próximo passo

em particular está relacionado a transações que podem não ter sido executadas no momento da falta, como foi descrito na seção 5.2.3.2. Dessa forma, se houver necessidade, as instâncias do GADM vão requisitar que essas transações pendentes sejam executadas.

Caso a falta tenha afetado alguma instância do GADM, no reinicio do sistema a instância que falhou vai avisar ao GMFT que os documentos que estavam armazenados nela foram perdidos. Dessa forma o GMFT vai realizar um *rollback* baseado nas dependências desses documentos perdidos. Para todos os documentos perdidos, o estado de suas dependências é atualizado para "não processado", de tal forma que elas ficarão disponíveis para serem processadas mais tarde, criando novamente os documentos perdidos.

Por fim, o último passo da recuperação das instâncias do GMFT é atualizar o estado dos documentos que estavam como "processando" para "não processado", de tal forma que eles possam ser retornados novamente para a aplicação para serem processados. Esses documentos são os documentos que estavam sendo processados no momento da falta, ou que tinham acabado de ser lidos pelo GADM, ou que até mesmo foram processados mas não tiveram seu estado atualizado para "processado" até o momento da falta.

5.2.4 Sumário

Uma vez detalhado como o sistema de fluxo de trabalho científico proposto neste trabalho funciona, temos que avaliar sua execução para termos certeza de que as decisões tomadas e as soluções propostas foram realmente boas. No próximo capítulo apresentamos os experimentos realizados e os resultados encontrados. Procuramos avaliar o sistema tanto em execuções normais quanto em execuções com faltas ocorrendo em seus componentes.

Capítulo 6

Experimentos

6.1 Ambiente Experimental

Os experimentos foram realizados utilizando-se a aplicação biomédica, descrita na seção 4.3, como base para avaliação do impacto dos mecanismos empregados no sistema de fluxo de trabalho científico que aumentam a sua disponibilidade. Eles foram executados em um aglomerado de computadores com 20 nodos conectados através de dois switches Gigabit Ethernet. Cada nodo tem um processador 1.4GHz AMD Athlon(tm) 64 Processor 3200+ com 2 GB de memória primária, executando Linux 2.6.15.

Nos experimentos foi utilizada uma base de dados real composta por 866 imagens microscópicas obtidas a partir de slides de uma placenta de rato. O tamanho total dessa base de dados é de aproximadamente 24 GB.

Como descrito na seção 4.3, o fluxo de trabalho resultante dessa aplicação deve ser executado em três estágios diferentes: o primeiro é composto pelo filtro PP/PF, o segundo pelo filtro de normalização do histograma, e o último pelos filtros de classificação das cores e segmentação do tecido. Dessa forma, os resultados apresentados estão separados por estágio.

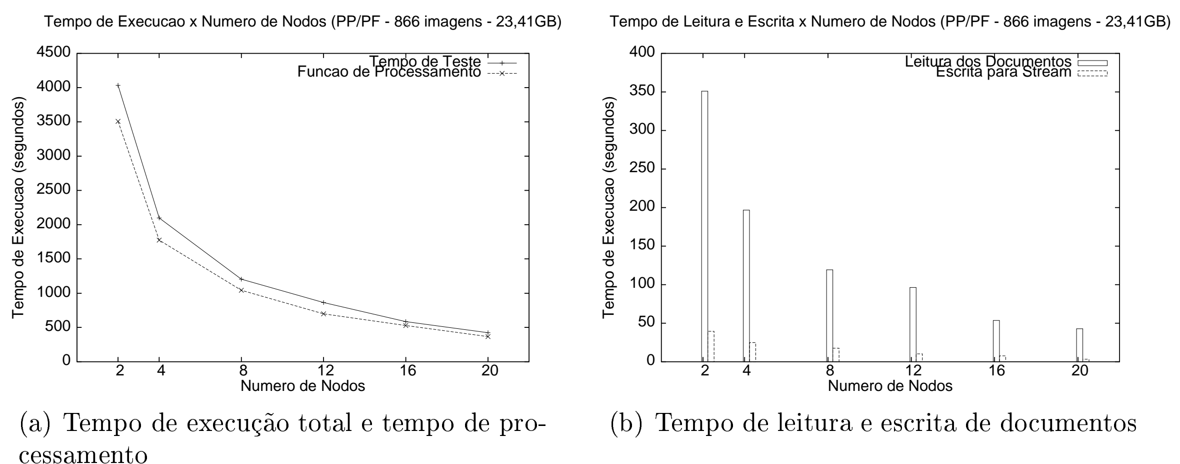
Em cada um dos estágios, como podemos observar na figura 4.2, um filtro *escritor* é utilizado para armazenar os dados de saída no sistema. Dessa forma, existe um *stream* de saída dos filtros PP/PF, normalização do histograma e classificação das cores para o filtro escritor que é utilizado para trafegar os documentos de saída. Durante a execução, é gerada uma instância de cada filtro da aplicação por nodo de processamento assim como um filtro escritor, um servidor Mako e uma instância do GADM. Apenas uma instância do GMFT foi utilizada, uma vez que em mais de 99% do seu tempo ela se encontra ociosa, como mostramos nos resultados.

6.2 Resultados

Esta seção de resultados é dividida em duas partes. A primeira, apresentada na seção 6.2.1, é para avaliar o desempenho do sistema, utilizando-se o mecanismo que aumenta a sua disponibilidade sem inserção de faltas. E a segunda (seção 6.2.2) é para avaliar o desempenho desse mesmo sistema quando faltas são inseridas durante a sua execução.

6.2.1 Resultados sem Inserção de Faltas

Os primeiros resultados apresentados são do primeiro estágio da execução da aplicação que é composto pelo filtro PP/PF. A figura 6.1 apresenta dois gráficos da execução desses filtros sem inserção de faltas. Como podemos perceber na figura 6.1(a), o tempo de execução total desse filtro é dominado por sua função de processamento, ou seja, pela função que realiza a separação do primeiro plano do plano de fundo. Na figura 6.1(b) apresentamos os tempos de leitura dos documentos armazenados no Gerente de Armazenamento Persistente, e de escrita para o filtro escritor do sistema. Esse tempo de escrita foi o tempo durante o qual o filtro ficou "esperando" o sistema enviar os documentos processados para serem armazenados no sistema.



(a) Tempo de execução total e tempo de processamento

(b) Tempo de leitura e escrita de documentos

Figura 6.1: Tempo de execução do filtro PP/PF sem inserção de faltas.

Para analisarmos o *overhead* introduzido pelo sistema na execução dos fluxos de trabalho, não consideramos como *overhead* o tempo de leitura dos dados, que é uma tarefa inevitável para executar a aplicação. Levamos em consideração tempos tais como de instanciação dos processos, escrita dos dados para o filtro escritor e espera pelo término de todos os processos. Dessa forma, pudemos perceber que o

overhead na execução desse primeiro estágio da aplicação variou de 2 a 5% do tempo de execução total, ou seja, ele foi muito pequeno.

Ainda analisando o primeiro estágio do pipeline, podemos verificar o speed-up da execução desse estágio na figura 6.2. Como podemos perceber ele está próximo ao linear. Dessa forma podemos observar que o sistema apresentou uma boa escalabilidade.

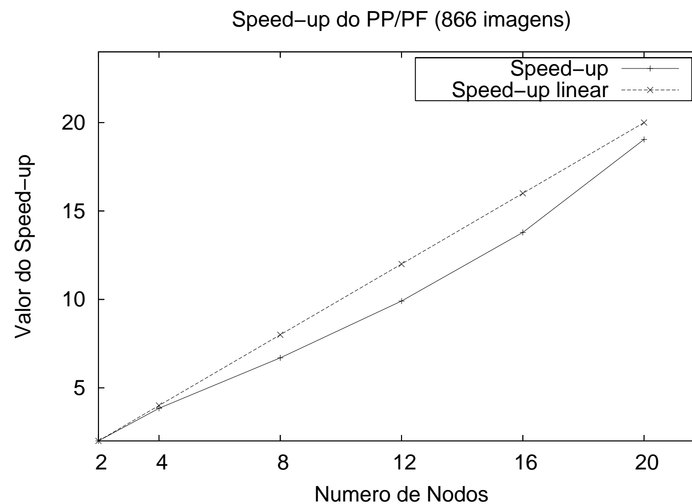


Figura 6.2: Speed-up.

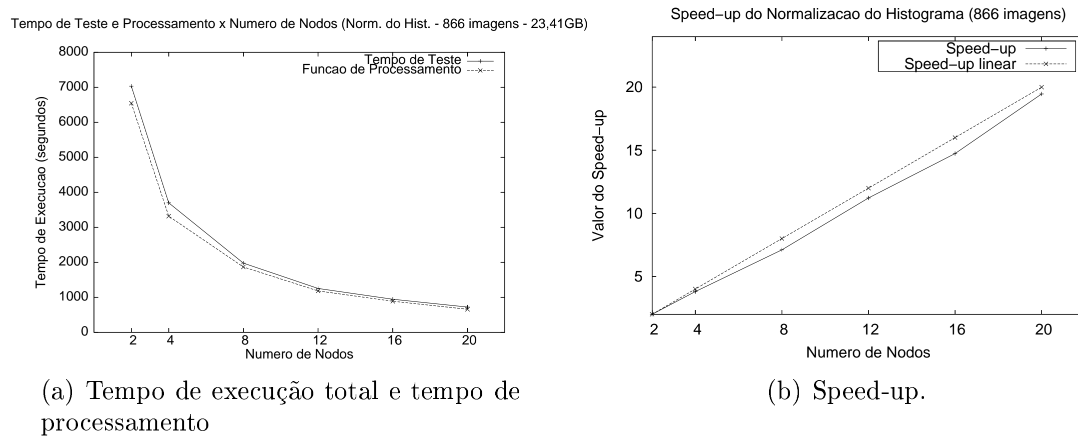
A figura 6.3 apresenta os resultados da execução do segundo estágio da aplicação biomédica formada pelo filtro Normalização do Histograma. Como podemos perceber na figura 6.3(a), a maior parte do tempo de execução, assim como o filtro PP/PF, está relacionada à função de processamento. O speed-up apresentado na figura 6.3(b) está ainda mais próximo ao linear, confirmando a boa escalabilidade do sistema.

Como podemos perceber na tabela 6.1, o tempo gasto pelo GMFT para atender as requisições das instâncias do GADM é muito pouco na grande maior parte do tempo - mais de 99% ele está ocioso. Por essa razão, não houve necessidade de utilizar mais de uma instância desse componente. Essa mesma situação ocorre para os três estágios da aplicação.

A figura 6.4 apresenta os resultados para o último estágio do fluxo de trabalho da aplicação biomédica, formado pelos filtros Classificação das Cores e Segmentação do Tecido, quando nenhuma falta é inserida. A figura 6.4(a) apresenta o speed-up desse estágio. Como podemos perceber, esse resultado está bem próximo do speed-up linear.

Tabela 6.1: Percentual do tempo, durante a execução do segundo estágio da aplicação, em que o GMFT fica ocioso, esperando por requisições.

Número de nodos	Percentual
2	99.98%
8	99.80%
16	99.61%



(a) Tempo de execução total e tempo de processamento

(b) Speed-up.

Figura 6.3: Execução do filtro Normalização do Histograma sem inserção de faltas.

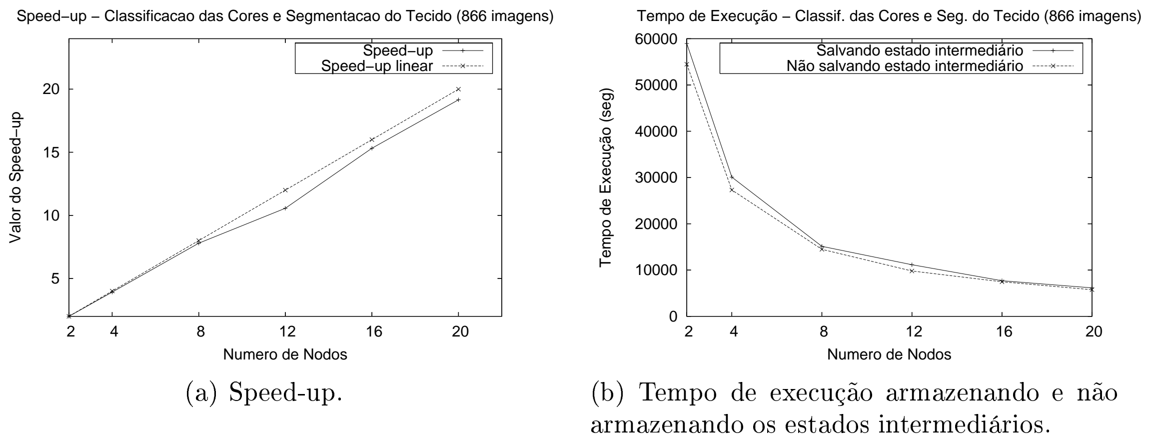
Uma outra análise importante que fizemos foi verificar o impacto do armazenamento dos resultados intermediários na execução da aplicação. Para realizar essa análise, resolvemos comparar duas situações: quando os resultados são e não são armazenados. A figura 6.4(b) apresenta essa comparação para o último estágio da aplicação. Nesse caso os documentos intermediários trocados entre os filtros Classificação das Cores e Segmentação do Tecido não são armazenados no sistema de fluxo de trabalho. Como podemos notar, o *overhead* introduzido é muito pequeno, confirmando a eficiência dos mecanismos introduzidos no sistema.

Por fim, resolvemos construir um gráfico para mostrar a eficiência total do sistema para executar a aplicação biomédica. A figura 6.5 mostra o speedup total do sistema quando os tempos de execução dos três estágios são somados. Dessa forma, podemos perceber que a escalabilidade do sistema de modo geral foi muito boa.

6.2.2 Resultados com Inserção de Faltas

Para avaliar a execução do fluxo de trabalho da aplicação biomédica, quando faltas são ou não inseridas, foram utilizados 6 diferentes cenários:

1. Sem inserção de falta.



(a) Speed-up.

(b) Tempo de execução armazenando e não armazenando os estados intermediários.

Figura 6.4: Execução dos filtros Classificação das Cores e Segmentação do Tecido sem inserção de faltas.

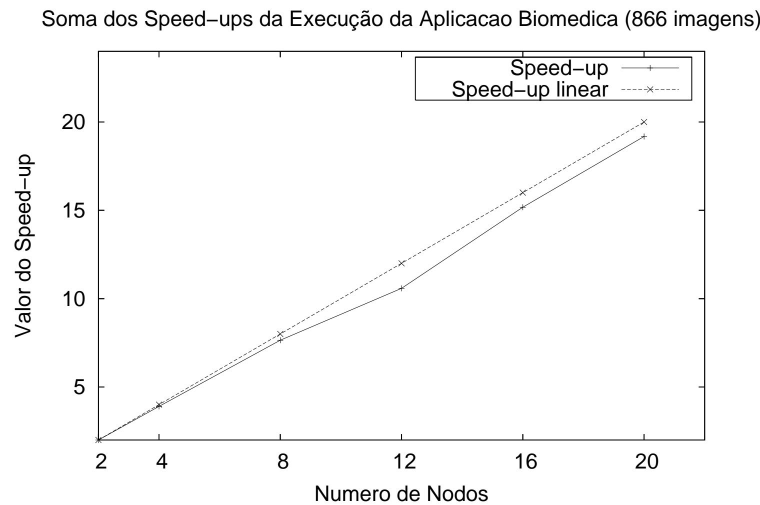


Figura 6.5: Speeup da soma dos tempos de execução dos três estágios da aplicação.

2. Uma falta inserida em uma instância de filtro do fluxo de trabalho.
3. Uma falta inserida no Gerente de Armazenamento de Dados em Memória (GADM).
4. Uma falta inserida no Gerente de Metadados dos Fluxos de Trabalho (GMFT).
5. Duas faltas inseridas em uma instância de filtro do fluxo de trabalho.
6. Três faltas inseridas em uma instância de filtro do fluxo de trabalho.

Cada experimento foi realizado três vezes, sendo que as faltas foram inseridas em tempos diferentes da execução (início, meio e final). O tempo total da execução

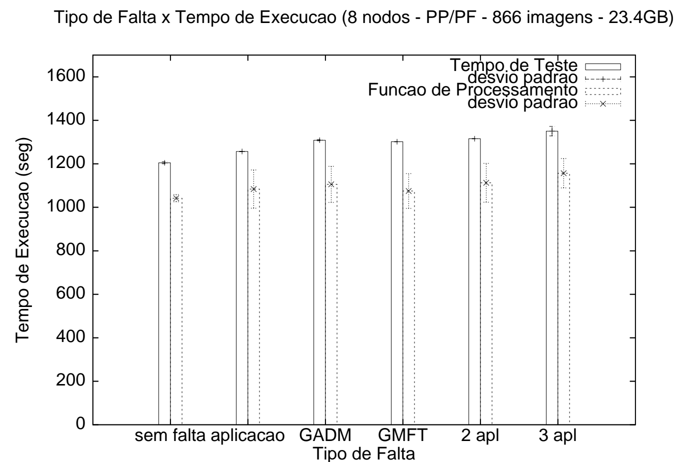


Figura 6.6: Tempo de execução do filtro PP/PF, rodando em 8 máquinas, em 6 diferentes cenários: 1 - sem falta; 2 - uma falta no filtro da aplicação; 3 - uma falta no GADM; 4 - uma falta no GMFT; 5 - duas faltas em filtros da aplicação; 6 - três faltas em filtros da aplicação

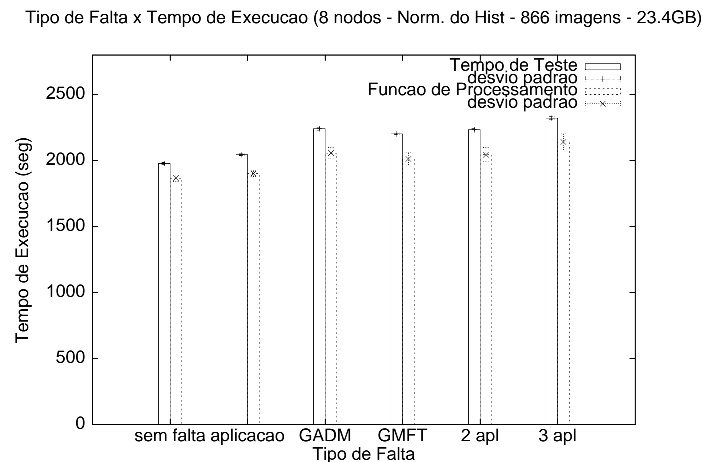


Figura 6.7: Tempo de execução do filtro Normalização do Histograma, rodando em 8 máquinas, em 6 cenários diferentes: 1 - sem falta; 2 - uma falta no filtro da aplicação; 3 - uma falta no GADM; 4 - uma falta no WMFT; 5 - duas faltas em filtros da aplicação; 6 - três faltas em filtros da aplicação

do estágio sem falta foi divido em três, e, na primeira execução, a falta era inserida no meio do primeiro terço desse tempo; a segunda, no meio do segundo terço; e a terceira, no meio do terceiro terço. Uma vez que a aplicação biomédica possui dependência de dados *1 para 1*, o número de documentos a serem reprocessados é geralmente o número de instâncias dos filtros que estavam executando. Dessa forma, não houve variação muito grande no tempo de execução dos três experimentos.

Quando mais de uma falta foi inserida por execução, tomamos o cuidado de não inserí-la durante a recuperação de uma falta anterior. Os tempos de inserção de faltas também foram calculados baseados no tempo de execução do estágio sem falta. Esse tempo foi dividido pelo número de faltas que iam ser inseridas por experimento e as faltas eram inseridas no meio de cada parte resultante da divisão. Dependendo do tipo de cenário tratado, uma instância era escolhida aleatoriamente e tinha sua execução terminada.

As figuras 6.6, 6.7 e 6.8 apresentam o tempo de execução do primeiro, segundo e terceiro estágios, respectivamente, para esses 6 cenários descritos rodando em 8 nodos do aglomerado de computadores. Como podemos perceber, os cenários 2, 3 e 4, com apenas uma falta inserida, apresentam tempo de execução bem próximo ao do cenário sem inserção de faltas. É interessante observar que o cenário 3 apresenta um tempo de execução um pouco maior do que o 2 e 4, uma vez que a falta ocorre no GADM, e documentos que estavam armazenados apenas nele foram perdidos e tiveram que ser recriados, com base em suas dependências.

Quando comparamos os cenários 1, 5 e 6, podemos observar que, quanto maior o número de faltas inseridas durante a execução da aplicação, maior será o tempo total da execução, e, conseqüentemente, maior será o tempo de processamento. Vale lembrar que, na reconfiguração do sistema na presença de falta, todas as instâncias dos filtros da aplicação tinham sua execução terminada e eram reiniciados. Dessa forma, os documentos que estavam sendo processados no momento da falha terão que ser processados novamente depois da recuperação.

O tempo de recuperação apresentado em cada um dos seis cenários dos três estágios da aplicação variou de 10 a 30 segundos, o que chega a representar cerca de 0.1% do tempo de execução total desse fluxo sem falta. Esse tempo é o gasto pelo Gerente do Anthill para reiniciar todas as instâncias dos filtros da aplicação, para notificar as instâncias do GADM e do GMFT da ocorrência da falta assim como o tempo gasto pelo GADM e pelo GMFT para executarem seu mecanismo de recuperação. O resto da diferença do tempo de execução sem falta para execução com falta é o tempo gasto para reprocessar os documentos que estavam no estado *"processando"* no momento da falta.

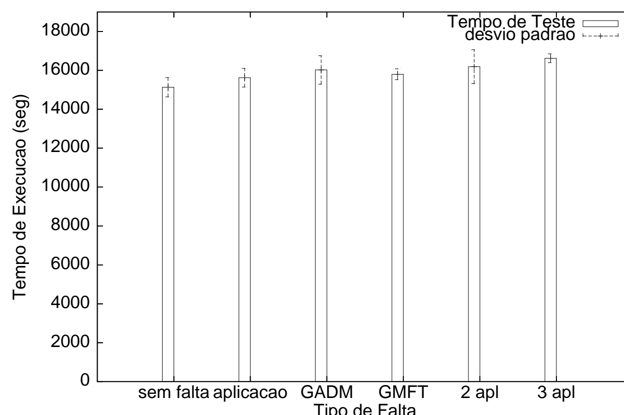


Figura 6.8: Tempo de execução dos filtros Classificação das Cores e Segmentação do Tecido, rodando em 6 máquinas, em 6 cenários diferentes: 1 - sem falta; 2 - uma falta no filtro da aplicação; 3 - uma falta no GADM; 4 - uma falta no GMFT; 5 - duas faltas em filtros da aplicação; 6 - três faltas em filtros da aplicação

6.2.3 Outras Aplicações

Como podemos perceber pelos resultados apresentados nas seções 6.2.1 e 6.2.2, o sistema obteve um ótimo desempenho tanto na ausência quanto na preseça de faltas, quando executa aplicações intensivas em dados que demandam um longo tempo de processamento. O *overhead* introduzido foi muito baixo, obteve uma ótima escalabilidade até 20 processadores e foi capaz de recuperar faltas durante a sua execução. Nessa seção iremos discutir como o sistema se comportaria para outros tipos de aplicação.

Um primeiro cenário a ser analisado é se a aplicação não fosse intensiva em processamento, ou seja, se o tempo gasto para processar cada documento fosse muito pequeno. Nesse caso, a execução do sistema poderia ser impactada pelo limite da largura de banda tanto do disco quanto da rede. O tempo gasto lendo/escrevendo os documentos do/no Gerente de Armazenamento Persistente (GAP) ou enviando os documentos processados para os próximos filtros seria bem maior do que o tempo gasto processando os documentos. Sujeito a esses limites, o sistema pode apresentar baixo desempenho, e os resultados encontrados nas duas seções anteriores não estariam de acordo.

Uma característica das aplicações que resultam nos fluxos de trabalho científicos é que elas são intensas tanto em dados quanto em processamento. Entretanto, caso fosse necessário adaptar o sistema para suportar aplicações que não fossem intensivas em processamento, a solução - entre outras - seria utilizar *prefetching* na leitura dos

dados e agrupamento dos documentos para armazená-los no GAP.

Na arquitetura de computadores, por exemplo, técnicas de *prefetching* de instruções são utilizadas para agilizar a execução de programas, reduzindo os estados de espera. Aplicando técnicas de *prefetching* em sistemas de fluxo de trabalho científico, poderiamos prever quais documentos seriam lidos no futuro. A leitura poderia ser feita não só nos momentos em que os filtros estivessem ociosos e pedissem documentos, mas também a qualquer momento. Os documentos lidos poderiam ser armazenados em memória para posterior utilização, o que diminuiria o tempo de leitura.

Para reduzir o *overhead* - como número de mensagens trocadas - na escrita dos documentos intermediários no Gerente de Armazenamento Persistente, uma técnica que poderia ser utilizada é o agrupamento de documentos para posterior armazenamento. Uma desvantagem dessa técnica é que o custo de recuperação do sistema na presença de faltas pode ser maior, uma vez que um número maior de documentos não estariam armazenados persistentemente e teriam que ser recriados.

Um outro cenário a ser analisado é se a aplicação apresentasse filtros com dependência *n para 1*, e o número de dependências de um documento fosse muito grande. Nesse caso, se houvesse uma falta no Gerente de Armazenamento de Dados em Memória em que esse documento estivesse armazenado, ele teria que ser recriado, e todas as suas dependências teriam que ser reprocessadas. O desempenho do sistema nesse cenário pode apresentar uma piora em relação ao desempenho apresentado na seção anterior, uma vez que um número grande de documentos teriam que ser reprocessados.

Capítulo 7

Conclusão e Trabalhos Futuros

Neste trabalho foi apresentado um mecanismo para aumentar transparentemente a disponibilidade de sistemas de fluxo de trabalho científico. Esse mecanismo tenta minimizar o trabalho perdido por uma falta, ou seja, quando o sistema é recuperado ele é capaz de retomar sua execução em um momento anterior à falta. Componentes para um eficiente gerenciamento e armazenamento de grandes quantidades de dados foram propostos e desenvolvidos buscando-se não causar um grande impacto na execução dos fluxos de trabalho. Métodos de tolerância a faltas, baseados em *checkpointing* e *message logging*, foram propostos e introduzidos a fim de recuperar o sistema após faltas.

Os resultados obtidos demonstraram um bom desempenho do sistema de fluxo de trabalho proposto neste trabalho. Os testes realizados com a aplicação biomédica apresentou um ótimo *speedup* e fomos capazes de perceber que o *overhead* introduzido com o armazenamento de dados intermediários na sua execução é muito pequeno.

Para avaliar o desempenho do sistema na presença de faltas, foram realizados experimentos com 5 tipos de cenários diferentes. Percebemos que quanto mais faltas acontecem no decorrer de um experimento, maior o tempo gasto. Quando a falta ocorre no Gerente de Armazenamento de Dados em Memória, o impacto da recuperação é grande pois há necessidade de realizar *rollback* dos documentos que estavam armazenados apenas em memória e foram perdidos. No geral os resultados apresentados com inserção de faltas estavam de acordo com o esperado.

7.1 Trabalhos Futuros

Várias são as extensões que podem ser agregadas ao sistema de fluxo de trabalho científico apresentado neste trabalho. Nessa seção descrevemos as principais:

- Expandir o modelo de programação. Na implementação atual, conseguimos controlar a dependência de dados e realizar a recuperação apenas de filtros que possuem dependência do tipo “1 para 1” ou “n para 1”. Uma possível melhora seria suportar dependências dos tipos “1 para 1” e “n para m”.

- Implementar um coletor de lixo. Hoje em dia assumimos que todos os resultados intermediários gerados pela execução dos fluxos de trabalho serão utilizados pelos seus desenvolvedores para inspeção. Essa suposição acaba deixando um volume de dados muito grande armazenado nos discos dos nodos onde o sistema é executado. Uma possível melhora a ser implementada é um coletor de lixo que remove dados a partir de regras dadas pelos usuários, ou seja, os usuários poderiam escolher os dados que deveriam permanecer armazenados ou não. Esse coletor teria que ser inteligente o suficiente para apagar os dados indesejáveis apenas quando eles realmente não seriam necessários para recuperar o sistema de uma falta.

- Incorporar o armazenamento do estado das aplicações. O mecanismo utilizado para recuperar o sistema de faltas não leva em consideração o estado criado pelos filtros. A incorporação de mecanismos que armazenem esse estado e utilize o na recuperação do sistema é uma extensão para o sistema.

- Implementar um mecanismo para reuso de dados. Durante a execução de diversos fluxos de trabalho, uma característica que pode acontecer é a de os fluxos de trabalho possuírem sitarefas (filtros) comuns entre e. A implementação de um mecanismo que fosse capaz de fazer essa detecção e otimizar o trabalho que deve ser realizado para executar esses fluxos é uma extensão bastante interessante e desafiadora.

- Implementar um escalonador de fluxos de trabalho. Não necessariamente todos os fluxos de trabalho que vão executar no sistema precisam ter a mesma prioridade para executar. Implementar um escalonador capaz de introduzir prioridades nas execuções dos fluxos também é uma extensão interessante.

Referências Bibliográficas

[ABJ+04] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludscher, and S. Mock. Kepler: An extensible system for design and execution of scientific workflows, 2004.

[AD99] R. H. Arpaci-Dusseau. Performance availability for networks of workstations - phd thesis, university of california, 1999.

[ADAD01] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. Fail-stutter fault tolerance. In *HOTOS '01: Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, page 33, Washington, DC, USA, 2001. IEEE Computer Society.

[ADV95] R. H. Arpaci, A. C. Dusseau, and A. M. Vahdat. Towards process management on a network of workstations, May 1995.

[AE02] Ken Ashcraft and Dawson Engler. Using programmer-written compiler extensions to catch security holes. In *SP '02: Proceedings of the 2002 IEEE Symposium on Security and Privacy*, Washington, DC, USA, 2002.

[AL81] T. Anderson and P. A. Lee. *Fault tolerance -principles and practice.* 1981.

[AL86] A. Avizienis and J. Laprie. Dependable computing: From concepts to design diversity. In *Proceedings of the IEEE, volume 74*, pages 629–638, May 1986.

[ALR01] A. Avizienis, J.-C. Laprie, and B. Randell. Fundamental concepts of dependability. Technical report, April 2001.

[AM98] L. Alvisi and K. Marzullo. Waft: Support for fault-tolerance in wide-area object oriented systems, 1998.

[BGS02] R. Badrinath, R. Gupta, and N. Shrivastava. Ftop: A library for fault tolerance in a cluster. In *14th IASTED Intl. Conf. on Parallel and Distributed Computing and Systems*, November 2002.

[BJ87] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. In *SOSP '87: Proceedings of the eleventh ACM Symposium on Operating systems principles*, pages 123–138, New York, NY, USA, 1987. ACM Press.

[BLNC06] S. Bowers, B. Ludascher, A. H.H. Ngu, and T. Critchlow. Enabling scientific workflow reuse through structured composition of dataflow and control-flow. In *Proceedings of IEEE Workshop on Workflow and Data Flow for Scientific Applications (SciFlow 2006)*, Atlanta, GA, April 2006.

[BWC+03] A. Berglund, S. B. X. WG., D. Chamberlin, M. F. Fernndez, M. Kay, J. Robie, and J Simon. *XML Path Language (XPath)*. 1st edition, August 2003.

[CAI04] CAIS. Vulnerabilidade no kernel do linux, Junho 2004.

[Car86] W. C. Carter. Hardware fault tolerance. pages 11–63, 1986.

[CASD85] F. Cristian, H. Aghali, R. Strong, and D. Dolev. Atomic broadcast: From simple message diffusion to byzantine agreement. In *Proc. 15th Int. Symp. on Fault-Tolerant Computing (FTCS-15)*, pages 200–206, Ann Arbor, MI, USA, 1985. IEEE Computer Society Press.

[CB93] J. B. Chen and B. N. Bershad. The impact of operating system structure on memory system performance. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 120–133, December 1993.

[CER] CERN. Large hadron collider (lhc) - http://www.interactions.org/lhc/.

[CFK+99] A. Chervenak, I. Foster, C. Kesselman, C. Salisbury, and S. Tuecke. The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets, 1999.

[CG87] George F. Clement and Paul K. Giloth. Evolution of fault tolerant switching systems in atet. In *The Evolution of Fault-Tolerant Computing, edited by A. AviYzienis, H. Kopetz, and J.C. Laprie, volume*

1 of Dependable Computing and Fault-Tolerant Systems, pages 37–54, Springer-Verlag, New York, 1987.

[CL99] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *Proc. 3rd USENIX Symposium on Operating Systems Design and Implementation*, New Orleans, LA, 1999.

[Cou05] Bruno Rocha Coutinho. Desempenho e disponibilidade em sistemas distribuídos em larga escala - tese de doutorado, 2005.

[DBG+03] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, A. Lazzarini, A. Arbree, R. Cavanaugh, and S. Koranda. Mapping abstract complex workflows onto grid environments. *In Journal of Grid Computing*, pages 25–39, 2003.

[DG04] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[DHSZ03] K. Driscoll, B. Hall, H. Sivencrona, and P. Zumsteg. Byzantine fault tolerance, from theory to reality. In *Proc. 22nd International Conference on Computer Safety, Reliability and Security (SAFECOMP03)*, pages 235–248, Edinburgh, Scotland, UK, October 2003.

[Di87] Z. Di. Eliminating domino effect in backward error recovery in distributed systems. In *Proceedings of the 2nd International Conference on Computers and Applications*, pages pp. 243– 248, 1987.

[DNR02] E. Dupont, M. Nicolaidis, and P. Rohr. Embedded robustness ips for transient-error-free ics. In *Design and Test of Computers - IEEE*, pages 56–70, 2002.

[EAWJ96] Mootaz Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. In *ACM Press*, New York, NY, USA, 1996.

[ES86] Paul D. Ezhilchelvan and Santosh K. Shrivastava. A characterisation of faults in systems. In *Proceedings of the Fifth Symposium on Reliability in Distributed Software and Database Systems*, pages 215–222, January 1986.

[FJG+05] Renato A. Ferreira, Wagner Meira Jr., Dorgival Guedes, Lúcia M. A. Drummond, Bruno Coutinho, George Teodoro, Túlio Tavares, Renata

Araújo, and Guilherme T. Ferreira. Anthill: A scalable run-time environment for data mining applications. In *17th International Symposium on Computer Architecture and High Peformance Computing*, Rio de Janeiro, RJ, 2005.

[FTFT01] J. Frey, T. Tannenbaum, I. Foster, and S. Tuecke. Condor-g: A computation management agent for multi-institutional grid. In *Tenth IEEE Symposium on High Performance Distributed Computing*, San Francisco, CA, August 2001.

[FVWZ02] I. Foster, J. Voeckler, M. Wilde, and Y. Zhao. Chimera: A virtual data system for representing, querying, and automating data derivation. In *Proceedings of 14th Conference on Scientific and Statistical Database Management*, 2002.

[GR93] J. Gray and A. Reuter. *Transaction processing: concepts and techniques.* Morgan Kaufmann, San Francisco, CA, 1993.

[Gra78] Jim Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, pages 393–481, London, UK, 1978. Springer-Verlag.

[HLOS04] S. Hastings, S. Langella, S. Oster, and J. Saltz. Distributed data management and integration: The mobius project. In *Global Grid Forum Semantic Grid Applications Wkshp*, Jun. 2004.

[HRL+05] S. Hastings, M. Ribeiro, S. Langella, S. Oster, U. Catalyurek, T. Pan, K. Huang, R. Ferreira, J. Saltz, and T. Kurc. Xml database support for distributed execution of data-intensive scientific workflows. *SIGMOD Record*, 34, 2005.

[htt] MySql Database http://www.mysql.com.

[Joh89] David B. Johnson. Distributed system fault tolerance using message logging and checkpointing. Technical Report TR89-101, 2, 1989.

[KEM+78] D. Katsuki, E.S. Elsam, W.F. Mann, E.S. Roberts, J.G. Robinson, F.S. Skowronski, and E.W. Wolf. Pluribus: An operational fault tolerant multiprocessor. In *Proceedings of the IEEE, 66(10):1146-1159*, October 1978.

[KF98] Carl Kesselman and Ian Foster. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, November 1998.

[KKF+04] George Kola, Tevfik Kosar, Jaime Frey, Miron Livny, Robert J. Brunner, and Michael Remijan. Disc: A system for distributed data intensive scientific computing. In *Proceeding of the First Workshop on Real, Large Distributed Systems (WORLDS'04)*, San Francisco, CA, December 2004.

[KKL05] George Kola, Tevfik Kosar, and Miron Livny. Faults in large distributed systems and what we can do about them. In *Proceedings of 11th European Conference on Parallel Processing (Euro-Par 2005)*, Lisbon, Portugal, August 2005.

[LAB+05] B. Ludascher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger-Frank, M. Jones, E. Lee, and Y. Zhao J. Tao. Scientific workflow management and the kepler system, to appear, 2005.

[Lam80] Butler W. Lampson. Atomic transactions. In *Advanced Course: Distributed Systems*, pages 246–265, 1980.

[Lap85] J. Laprie. Dependable computing and fault-tolerance: Concepts and terminology. In *Proceedings of 15th IEEE International Fault-Tolerant Computing Symposium*, pages 2–11, Ann Arbor, 1985.

[LKL04] G. Lola, T. Kosar, and M. Livny. Phoenix: Making data-intensive grid applications fault-tolerant. In *Proceedings of 5th IEEE/ACM International Workshop on Grid Computing*, 2004.

[LN] Inc. Linux Networkx. Icebox cluster management appliance.

[L.P] Hewlett-Packard Development Company L.P. Hp integrated lights-out standard.

[LP95] Edward A. Lee and Thomas M. Parks. Dataflow process networks. In *Proceedings of the IEEE*, pages 773–799, may 1995.

[LSP82] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. In *ACM Transactions on Programming Languages and Systems, 4 (3)*, pages 382–401, July 1982.

[MM88] A. Mahmood and E. J. McCluskey. Concurrent error detection using watchdog processors-a survey, February 1988.

[NKB+03] K. Nagaraja, N. Krishnan, R. Bianchini, R. Martin, and T. Nguyen. Quantifying and improving the availability of cooperative cluster-based services, 2003.

[PFT+05] Rui Pinho, Yuri Faria, Thiago Teixeira, Tulio Tavares, George Teodoro, and Wagner Meira Jr. Análise e entendimento de desempenho com a ferramenta antfarm. In *VI Workshop em Sistemas Computacionais de Alto Desempenho*, Rio de Janeiro, RJ, 2005.

[PH05] Tony C. Pan and Kun Huang. Virtual mouse placenta: Tissue layer segmentation. In *Proceedings of the 27th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC2005)*, 2005.

[PVB+88] D. Powell, P. Verissimo, G. Bonn, F. Waeselynck, and D. Seaton. The delta-4 approach to dependability in open distributed computing systems. In *Digest of Papers, The Eighteenth International Symposium on Fault-Tolerant Computing*, pages 246–251, June 1988.

[RK06] Leonid Ryzhyk and Ihor Kuz. Towards operating system support for application-specific fault-tolerance protocols. In *Proceedings of the 2nd International Workshop on Object Systems and Software Architectures*, Victor Harbor, South Australia, Australia, January, 2006.

[SH82] John F. Shoch and Jon A. Hupp. The worm programs: early experience with a distributed computation. *Commun. ACM*, 25(3):172–180, 1982.

[Sie86] D . Siewiorek. Architecture of fault-tolerant computers. In *FaultTolerant Computing: Theory and Techniques*, pages 417–466, Prentice-Hall, 1986. Dhiraj K. Pradhan editor.

[SM96] Alexander Schill and Christian Mittasch. Workflow management systems on top of osf dce and omg corba. *Distributed Systems Engineering*, 3(4):250–262, 1996.

[SRC84] Jerome H. Saltzer, David P. Reed, and David D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.

[SS83] Richard D. Schlichting and Fred B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *Computer Systems*, 1(3):222–238, 1983.

[Sun90] V. S. Sunderam. PVM: a framework for parallel distributed computing. *Concurrency, Practice and Experience*, 2(4):315–340, 1990.

[TL02] Douglas Thain and Miron Livny. Error scope on a computational grid: Theory and practice. In *Proceedings of the 11th IEEE Symposium on High Performance Distributed Computing (HPDC-11)*, Edinburgh, Scotland, July 2002.

[TMM+02] O. Tatebe, Y. Morita, S. Matsuoka, N. Soda, and S. Sekiguchi. Grid datafarm architecture for petascale data intensive computing. In *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid)*, 2002.

[Tre04] Michael Treaster. A survey of fault-tolerance and fault-recovery techniques in parallel systems, 2004.

[TTF+06] G. Teodoro, T. Tavares, R. Ferreira, T. Kurc, W. Meira Jr., D. Guedes, T. Pan, and J. Saltz. Run-time support for efficient execution of scientific workflows on distributed environments. In *Proc. of The 18th Symposium on Computer Architecture and High Performance Computing SBAC-PAD 06*, Ouro Preto, Brazil, October 2006.

[VJF+04] A. Veloso, W. Meira Jr., R. Ferreira, D. Guedes, and S. Parthasarathy. Asynchronous and anticipatory filter-stream based parallel algorithm for frequent itemset mining. In *European Conference on Principles of Data Mining and Knowledge Discovery (PKDD 2004)*, 2004.

[VR01] P. Verissimo and L. Rodrigues. *Distributed Systems for System Architects*. Kluwer Academic Publishers, Boston, USA, January 2001.

[YTEM04] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using model checking to find serious file system errors. In *Processing of the Sixth Symposium on Operating Systems Design and Implementation (OSDI '04)*, page 273 288, San Francisco, CA, December 2004.