BRUNO ROCHA COUTINHO

DESEMPENHO E DISPONIBILIDADE EM SISTEMAS DISTRIBUÍDOS EM LARGA ESCALA

Belo Horizonte, Minas Gerais
17 de outubro de 2005

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

DESEMPENHO E DISPONIBILIDADE EM SISTEMAS DISTRIBUÍDOS EM LARGA ESCALA

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.

BRUNO ROCHA COUTINHO

Belo Horizonte, Minas Gerais
17 de outubro de 2005

UNIVERSIDADE FEDERAL DE MINAS GERAIS

FOLHA DE APROVAÇÃO

Desempenho e Disponibilidade em Sistemas Distribuídos em Larga Escala

BRUNO ROCHA COUTINHO

Dissertação defendida e aprovada pela banca examinadora constituída por:

Ph. D. Wagner Meira Junior – Orientador
Universidade Federal de Minas Gerais

Ph. D. Dorgival Olavo Guedes Neto
Universidade Federal de Minas Gerais

Ph. D. Renato Antonio C. Ferreira
Universidade Federal de Minas Gerais

Ph. D. Lúcia Maria de A. Drummond
Universidade Federal Fluminense

Belo Horizonte, Minas Gerais, 17 de outubro de 2005

Resumo

Neste trabalho, implementamos um novo modelo de tolerância a faltas por checkpoint e recuperação, baseado em tarefas. Este modelo permite que a aplicação continue progredindo sua execução enquanto o sistema faz a cópia primária dos dados, paralelizando processamento e E/S. Nesse modelo, o sistema de checkpoint não bloqueia a execução da aplicação nem reduz suas oportunidades de paralelização impondo barreiras ou outros tipos de sincronização. Além disso, ele pode ser implementado usando checkpoints não coordenados sem correr o risco de que o padrão de troca de mensagens da aplicação gere dependências entre os intervalos de checkpoints que causariam um efeito "dominó" no momento da recuperação, obrigando a execução da aplicação voltar ao seu início.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Nas últimas décadas, avanços nas tecnologias de semicondutores e circuitos integrados têm permitido a construção de computadores cada vez mais rápidos e eficientes. Esse aumento contínuo da capacidade dos sistemas têm permitido a criação de aplicações ainda mais computacionalmente exigentes, que por sua vez intensificam ainda mais a demanda por máquinas mais poderosas [dRC94]. Isso gera uma reação em cadeia, cujo efeito é gerar demanda por maior capacidade de processamento não importando o quão alto a oferta de processamento atinja.

Entretanto, computadores de grande porte são construídos utilizando componentes especializados e projetados para trabalharem em conjunto [BMPS03]. E esses componentes não evoluem na mesma velocidade que os componentes genéricos utilizados na construção de computadores pessoais e pequenos servidores, o que faz com que o custo/benefício desses últimos seja muito melhor que os de computadores de grande porte.

Diante desse fato, vem surgindo um interesse crescente em substituir essas grandes máquinas por grupos de máquinas de baixo custo, conectadas por rede, também conhecidos como *clusters* de computadores [BMPS03]. Esse tipo de solução pode proporcionar a mesma capacidade de processamento que um computador de grande porte a um custo que pode chegar a ser uma ordem de grandeza menor [Tur00]. Outra vantagem é a capacidade de expansão desse sistema: para aumentar o poder de processamento de um *cluster* basta adicionar mais computadores [Bre01], o que normalmente não é tão simples em computadores de grande porte.

No entanto, um fator importante para o aumento das faltas[1] nos sistemas computacionais baseados em máquinas de baixo custo é que elas não apresentam uma confiabilidade muito alta, ao contrário dos grandes computadores que são cuidadosamente integrados, resultando em sistemas altamente robustos (isso é uma das razões

[1] Anomalia no sistema. Para uma explicação mais detalhada, veja a Seção 1.3.

do seu alto custo). Isso faz com que os *clusters* sofram uma quantidade de faltas muito maior que um computador de grande porte. Assim, para aproveitar a melhor relação custo-benefício dos *clusters* de computadores é essencial o desenvolvimento de aplicações capazes de lidar com faltas durante sua execução. A importância de se lidar com faltas em *clusters* pode ser ilustrada pelo fato de muitos pesquisadores considerarem a capacidade do serviço de procura Google de operar com *clusters* compostos por milhares de máquinas de baixo custo (e conseqüentemente baixa confiabilidade) mais valiosa que o algoritmo de pesquisa que deu origem ao serviço [New05].

Outro fator que aumenta o número de faltas nos sistemas computacionais advém do fato de que, quanto mais componentes tem um sistema, maior é a probabilidade de um deles falhar. Todos os sistemas atualmente, tanto *clusters* quanto grandes supercomputadores, tendem a ter um número cada vez maior de processadores, aumentando a possibilidade de faltas. Todas as quinze máquinas no topo da lista Top500 dos sistemas mais rápidos do mundo [MSDS05] têm mais de 2000 processadores e o BlueGene/L da IBM tem 32768 processadores, número maior que as 18.000 válvulas do ENIAC (várias delas queimavam todos os dias, fazendo com que ele ficasse indisponível durante metade do tempo [Wik]). Com o surgimento de malhas computacionais (*Grids*), onde as aplicações executam sobre ambientes distribuídos, compostos por um grande número de máquinas heterogêneas, administradas por diferentes instituições e conectadas pela Internet que é sujeita a vários tipos de faltas e ataques imprevisíveis, a taxa de faltas será muito maior.

Se por um lado a probabilidade de faltas é cada vez maior, por outro, muitos programas de computação científica estão sendo concebidos para executar por períodos cada vez maiores, podendo chegar a dias ou até meses de execução. Isso exigirá sistemas que consigam executar computações por longos períodos sem interrupção. Programas de certificação do *Advanced Simulation and Computing Program (ASC)* do Departamento de Energia dos Estado Unidos e de cálculo de enovelamento de proteínas *ab-initio* como os para o BlueGene/L da IBM foram feitos para executarem durante meses a fio [BMPS03].

Se considerarmos aplicações comerciais o cenário é ainda mais crítico, uma vez que a maioria delas vem sendo implementada na forma de serviços *Web*, exigindo que estejam 24 horas por dia e 7 dias por semana no ar. Dependendo do tipo de serviço qualquer parada, mesmo que programada, pode custar muito caro [PBB+02]. De acordo com o *National Institute of Standards and Technology (NIST)* dos Estados Unidos, quedas de sistemas custam à economia desse país 60 Bilhões de dólares por ano, representando 0.6% do PIB. Em sistemas de gerenciamento de registros

médicos, votação eletrônica ou controle de tráfego aéreo, falhas em sistemas de software podem causar conseqüências ainda piores em termos de segurança.

Entre os vários tipos de aplicações existentes, estamos interessados em implementar tolerância a faltas especificamente nos que utilizam o modelo de programação filtro-fluxo. Nesse modelo, a aplicação é representada como um *pipeline* de filtros que se comunicam através de fluxos. Esse modelo é muito semelhante aos *pipes* do *Unix*, *buffers* que podem ser escritos por um processo e lidos por outro, podendo ser usados na comunicação entre processos e para montar *pipelines* de processos.

Para isso criamos a abstração de tarefa. Uma tarefa é uma evento global com início e fim bem definidos em cada processador (através das chamadas createTask() e endTask() no código da aplicação) onde é permitida a troca de mensagens com outros processadores que estão executando a mesma tarefa.

1.1 Objetivo deste trabalho

Nosso problema é que para aproveitar o grande custo-benefício de sistemas de computação de alto desempenho montados usando hardware de prateleira é necessário que a aplicação seja implementada utilizando técnicas de tolerância a faltas. Porém, a implementação dessas técnicas é muito trabalhosa e existe um número muito pequeno de programadores com treinamento formal nessas técnicas.

Assim, o objetivo deste trabalho é adicionar tolerância a faltas a um ambiente de programação filtro-fluxo, abstraindo os detalhes de tolerância a faltas para que os programadores possam construir aplicações capazes de executar sobre sistemas compostos por componentes de baixo custo, que podem falhar a qualquer momento, mesmo sem treinamento formal em tolerância a faltas.

1.2 Contribuições do trabalho

A principal contribuição deste trabalho é um novo modelo de tolerância a faltas por *checkpoint* e recuperação baseado em tarefas. Uma tarefa é um evento global com início e fim bem definidos em cada processador (através das chamadas no código da aplicação) onde é permitida a troca de mensagens com outros processadores que estão executando a mesma tarefa.

Este modelo permite que a aplicação continue progredindo em sua execução enquanto o sistema faz a cópia dos dados para o armazenamento estável[2], paralelizando

[2]Armazenamento que sobreviverá mesmo que algum processo ou máquina falhe. Descrito na Seção 4.2.

processamento e E/S. Nesse modelo, o sistema de *checkpoint* não bloqueia a execução da aplicação nem reduz suas oportunidades de assincronia impondo barreiras ou outros tipos de sincronização.

Além disso, ele pode ser implementado usando *checkpoints* não coordenados sem correr o risco de que o padrão de troca de mensagens da aplicação gere dependências entre os intervalos de *checkpoints* que causariam um efeito dominó no momento da recuperação, obrigando a execução da aplicação a voltar ao seu início.

1.3 Conceitos

Existe muita confusão em relação à tradução para português dos termos técnicos básicos de tolerância a faltas em inglês: *fault* (condição anômala), *error* (manifestação dessa condição no sistema) e *failure* (o sistema não conseguir cumprir sua "missão"). Neste trabalho utilizaremos a terminologia proposta por Veríssimo e Lemos [VdL89], ou seja, utilizaremos falta→erro→falha como tradução de *fault*→*error*→*failure*. Mesmo que essa terminologia esteja mais longe do senso comum e existam trabalhos mais recentes que propõem metodologias alternativas, o autor desta dissertação considera essa a menos confusa. A seguir esses conceitos serão explicados mais detalhadamente:

Todo sistema possui uma especificação que descreve seu comportamento. Caso o sistema cumpra sua especificação, ele é considerado *correto*. Infelizmente é extremamente difícil construir sistemas complexos que se comportem da forma correta todo o tempo devido à grande quantidade de entradas que o sistema pode receber e de estados que ele pode ter. Assim um sistema pode receber uma entrada ou sofrer uma perturbação que o leve a se comportar de forma diferente da especificada. Quando isso acontece, consideramos que o sistema *falhou* [MS92, Cri91].

Uma *falta* é uma condição anômala, que pode causar a falha de um sistema. Faltas podem ser causadas por distúrbio externo (intencional ou não), fadiga, defeito de fabricação, erro de codificação ou evento não previsto no projeto (erro de projeto).

Algumas vezes ocorre alguma falta e os próprios componentes ou subsistemas conseguem tratá-la. Nesse caso ela se torna uma *falta mascarada* para o resto do sistema. Porém muitas vezes isso não acontece e o escopo da falta ultrapassa os limites de um componente, se manifestando em outras partes do sistema. Quando isso acontece temos um *erro*.

A tolerância a faltas consiste em tentar fazer com que uma falta não gere um erro (mascarar a falta) ou, se gerar, fazer o sistema se recuperar do erro sem realizar uma ação potencialmente perigosa nem deixar de fazer o que foi projetado para fazer.

Essas duas últimas propriedades são chamadas de vitalidade e segurança [Gär99]. Vitalidade (*liveness*) é o sistema continuar a responder requisições, ou seja, continuar a se comportar como descreve a sua especificação, realizando a atividade para a qual foi construído (sua *missão*). Segurança (*safety*) consiste em evitar que uma falta faça com que o sistema tome ações perigosas como entrar em um estado inválido, emitir uma saída errada, ou disparar comandos que podem comprometer outros sistemas ou ferir pessoas. Apesar do termo ser "segurança", não está nesse escopo (*safety*) lidar com faltas causadas intencionalmente, como invasões, roubo de dados ou ataques de negação de serviço (*denial of service*). A área de estudo que tenta evitar faltas intencionais é *security* (que também é traduzida como segurança) [VdL89].

Todas essas técnicas consistem em replicar componentes do sistema que podem falhar, causando a falta no sistema, ou fazer replicação temporal (caso um componente falhe, reexecuta-se a computação que ele estava fazendo). A primeira opção geralmente é implementada usando-se hardware redundante, o que tende a se tornar muito caro. A segunda é implementada utilizando-se técnicas de redundância por software, tais como camadas de armazenamento de estado com replicação [LF03, Gri01], pontos de controle (*checkpoints*) onde o estado da computação é replicado para ser recuperado na ocorrência caso de uma falta [EAWJ02, BMPS03]; reinício de subsistemas faltosos para trazê-los de volta a um estado bem conhecido (o estado inicial), técnica conhecida pelo nome de *microreboots* recursivos [CF01, CF03, CCF04, PBB+02] ou simples reexecução de computações que foram realizadas por componentes que falharam [DG04].

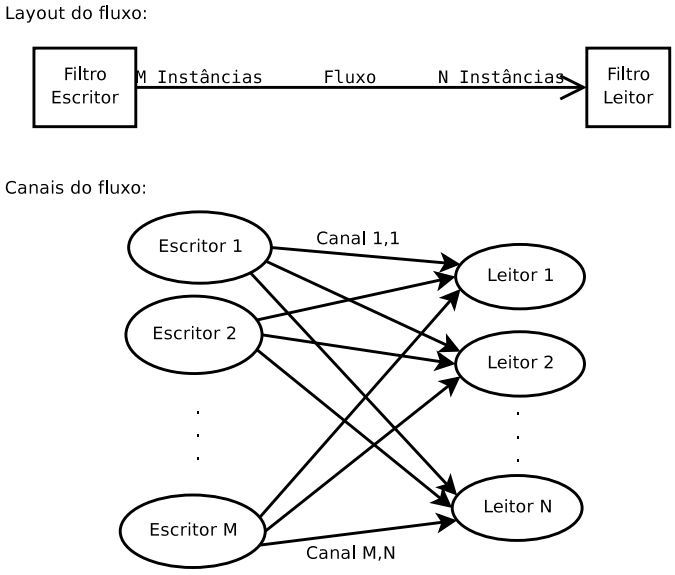
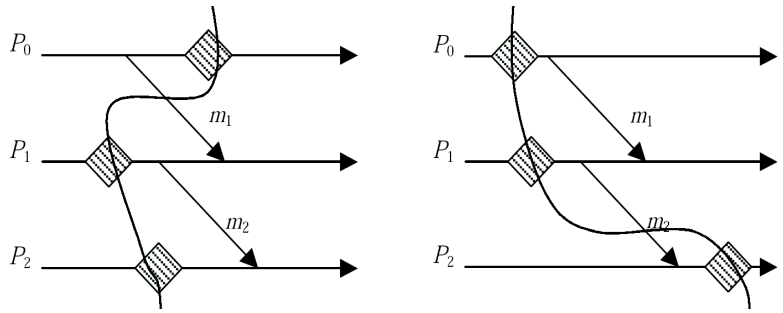


Figura 1.1: Canais compondo um fluxo

Técnicas de pontos de controle e recuperação por *rollback* consistem em recuperar o sistema de uma falta, trazendo-o de volta a um estado anterior a ela. Elas tratam um sistema distribuído como um conjunto de processos que se comunicam trocando mensagens por uma rede. A tolerância a faltas é conseguida salvando periodicamente o estado dos processos durante uma execução normal. Na ocorrência de uma falta, os processos afetados são reiniciados de um de seus estados salvos, reduzindo a quantidade de computação perdida. Cada estado salvo é chamado de *ponto de controle*.

O objetivo deste trabalho é implementar pontos de controle e recuperação por *rollback* para ser utilizado por aplicações construídas sobre o Anthill. Uma aplicação executando sobre o Anthill é um sistema distribuído, onde cada instância de cada filtro é um processo e um fluxo entre dois filtros é um conjunto de canais unidirecionais, cuja direção é a direção do fluxo: de instâncias do filtro que escreve dados no fluxo para instâncias do filtro que lê dados do fluxo (Figura 1.1).



(a) *Checkpoint* consistente com mensagem em trânsito (b) *Checkpoint* inconsistente

Figura 1.2: Linhas de *checkpoint*

O estado de um sistema distribuído é constituído do estado de todos os processos que o constituem mais o estado dos canais de comunicação entre eles. O estado dos canais de comunicação é constituído pelas mensagens em trânsito (Figura 1.2(a)). Como o próprio nome diz, *mensagens em trânsito* são mensagens que foram enviadas mas ainda não chegaram ao destinatário.

Para um estado global de um sistema distribuído ser *consistente*, todas as mensagens devem estar registradas da mesma forma em ambas as pontas do canal. Caso uma mensagem esteja registrada como recebida no receptor, ela deve estar registrada como enviada no emissor. Porém, existe um caso especial: caso a mensagem não esteja registrada como recebida, ela pode ela pode estar registrada como enviada no emissor, o que caracteriza uma mensagem em trânsito, como a da Figura 1.2(a). Nesse caso, o conteúdo da mensagem faz parte do estado do canal.

Sistemas distribuídos complicam a recuperação por *rollback* devido ao fato de gerarem dependências entre processos durante uma execução normal. Quando algum processo sofre uma falta, essas dependências podem forçar processos que não sofreram faltas a voltar atrás, criando o que é chamado de *propagação de rollback*. No pior caso, a propagação de *rollback* pode chegar até o estado inicial da computação, perdendo todo o trabalho realizado antes da falta. Esta situação é conhecida como *efeito dominó* [Ran75, EAWJ02].

Para ver por que a propagação de *rollback* ocorre, considere uma situação onde o remetente de uma mensagem m volta atrás para um estado anterior ao envio de m. Isso força o destinatário a voltar também para um estado anterior à recepção de m, caso contrário os estados dos dois processos formariam um estado global *inconsistente* (Figura 1.2(b)) pois, segundo eles, a mensagem m foi recebida sem ter sido enviada, o que é impossível em uma execução sem faltas. Para isso não acontecer, os estados dos processos devem formar um estado global *consistente*, que poderia existir em uma execução sem faltas.

Uma vez que as dependências nem sempre permitem que os estados mais recentes de todos os processos sejam aproveitados na recuperação, foi criado o conceito de linha de recuperação. A *linha de recuperação* é o conjunto de estados locais de processos que serão utilizados na recuperação do estado do sistema. Se um processo grava um ponto de controle que depende de uma mensagem cujo envio anda não foi registrado, ele não poderá fazer parte da linha de recuperação.

1.4 Organização do texto

Este trabalho está dividido em 7 capítulos. O resto dele está organizado da seguinte forma:

Capítulo 2. [Conceitos e Trabalhos Relacionados] Neste capítulo, serão introduzidos os conceitos utilizados na literatura de tolerância a faltas e discutidos alguns trabalhos relacionados.

Capítulo 3. [Modelo de programação do Anthill] Neste capítulo serão apresentados o modelo de programação do ambiente Anthill e as extensões criadas nesse modelo para que o ambiente possa lidar com tolerância a faltas.

Capítulo 4. [Modelos de tolerância a faltas] São discutidos diversos modelos de faltas e tolerância a faltas presentes na literatura, bem como os motivos que nos levaram a utilizá-los ou não.

Capítulo 5. [Implementação] Este capítulo detalha os algoritmos, protocolos e as estruturas de dados utilizados na implementação do sistema de tolerância a faltas do Anthill.

Capítulo 6. [Experimentos] Aqui são descritos os experimentos de medição de desempenho e injeção de faltas, bem como a metodologia utilizada na avaliação do desempenho e da disponibilidade do sistema de tolerância a faltas implementado.

Capítulo 7. [Conclusão e trabalhos futuros] Mostra as conclusões do trabalho e apresenta sugestões de continuação do mesmo.

Capítulo 2

Trabalhos Relacionados

Neste capítulo apresentaremos os trabalhos relacionados à tolerância a faltas no Anthill. Este capítulo está dividido em duas partes: na Seção 2.1 analisaremos *middlewares* com modelo de programação parecido com o do Anthill e na Seção 2.2 apresentaremos mecanismos de tolerância a faltas que foram implementados em outros sistemas de computação distribuída existentes na literatura.

2.1 Trabalhos relacionados com o Anthill

Nessa seção apresentaremos ambientes de programação distribuída com modelo de programação semelhante ao do Anthill. Na Seção 2.1.1 descreveremos o DataCutter, um *middleware* cujo modelo de programação, o modelo filtro-fluxo, foi estendido pelo Anthill [FJG+05, VJF+04], o sistema sobre o qual este trabalho implementa tolerância a faltas. Na Seção 2.1.2 será apresentado o MapReduce, um ambiente de computação distribuída em larga escala amplamente utilizado no Google e finalmente será apresentado o Borealis, um sistema de processamento de fluxos de dados distribuído em tempo real.

2.1.1 DataCutter

O DataCutter [BKC+01, BCC+02, AKSS01, NCK+03, NKCS03, SFB+02] é um *middleware* para exploração e análise de bases de dados científicas em ambientes distribuídos e heterogêneos. Seu modelo de programação, chamado de programação filtro-fluxo, representa a aplicação como um *pipeline* de filtros. Cada filtro pode ser executado em várias máquinas em uma rede. A troca de dados entre dois filtros é feita mediante o uso de fluxos, que são canais uni-direcionais que enviam dados em *buffers* de tamanho fixo.

O DataCutter provê um conjunto básico de serviços que podem ser utilizados para implementar outros serviços mais específicos da aplicação ou ser combinados com serviços de malhas computacionais (*grids*) como gerenciamento de metadados, gerenciamento de recursos e serviços de autenticação. Porém, o DataCutter ainda não tem um sistema de tolerância a faltas.

2.1.2 MapReduce

O *MapReduce* [DG04] é um ambiente de programação para aplicações distribuídas para processamento de dados em larga escala assim como o DataCutter e o Anthill. Os três são bastante parecidos em termos de modo de operação: ambos possuem um gerente que faz a distribuição e escalonamento da aplicação em várias máquinas além de monitorar o status da computação e fazer detecção de faltas.

Ambos fazem paralelização automática utilizando o conceito de cópias transparentes. O desenvolvedor da aplicação programa como se estivesse fazendo um código seqüencial, porém o sistema de execução iniciará várias cópias do programa (que trabalharão em paralelo) e lidará com os detalhes de troca de mensagens entre os processos e sincronização (mesmo que implícita).

As similaridades não acabam por aí: uma inovação implementada no Anthill em relação ao DataCutter é o fluxo rotulado. No fluxo rotulado o programador da aplicação indica uma função que extrai um rótulo da mensagem que será usado como chave em uma função *hash* que irá roteá-la para algum processador da mesma forma que no *MapReduce* o programador da aplicação gera valores intermediários na forma de pares chave/valor que são roteados pela chave.

No *MapReduce*, os processos da aplicação são divididos em dois tipos: os de mapeamento e os de redução. Os valores intermediários são da forma chave/valor e os todos os valores com a mesma chave produzidos por todos processos de mapeamento vão para o mesmo processo redutor. Isso é o equivalente a uma aplicação Anthill para fazer redução. Essa aplicação teria 2 filtros: o mapeador e o redutor, o mapeador lê os dados de entrada e os dados intermediários são enviados na forma de mensagens em um fluxo rotulado para o filtro redutor que gera a resposta final.

Uma diferença no código em relação ao feito para o *MapReduce* é que o *MapReduce* exige que cada processo redutor ordene os dados que recolheu de todos os processos mapeadores antes de enviá-los para a função de redução. Por outro lado, o Anthill não exige essa ordenação, assim o filtro redutor tem que fazer a ordenação por conta própria ou ser capaz de trabalhar com os dados em ordem aleatória. Se a aplicação feita no Anthill não necessitar dessa ordenação, provavelmente seria muito mais rápida, pois além de poupar o custo de uma ordenação gigantesca, poderá re-

alizar o mapeamento em paralelo com a redução (uma das formas de paralelização utilizadas pelo Anthill). Além disso, o *MapReduce* realiza apenas a operação de redução distribuída, já o DataCutter e o Anthill são mais gerais, permitindo mais operações além da composição de *pipelines* mais elaborados que dois filtros ligados por um fluxo.

A tolerância a faltas no *MapReduce* é feita principalmente através de reexecução das processos que estavam nas máquinas que falharam. Caso o processo seja de mapeamento e já tenha terminado, o mestre notificará os redutores que estavam lendo dados no disco da máquina que falhou para lerem da máquina que está reexecutando o processo.

A conseqüente facilidade de programação e de tolerância a faltas no *MapReduce* tem o custo de impor uma sincronia muito alta à aplicação, o que acaba diminuindo a sua eficiência. Um dos requisitos que tivemos muito cuidado em garantir quando projetamos esse protocolo de tolerância a faltas foi evitar que ele limite as oportunidades de assincronia da aplicação ou atrase seu progresso.

2.1.3 Borealis

Borealis [AAB+05, BBMS05] é um sistema de processamento de fluxos de dados (*Stream Processing Engine* ou SPE) distribuído em tempo real. Ele é voltado para o processamento de dados originados diretamente de fontes geradores como redes de sensores, sistemas de processamento de transações, sistemas de monitoramento, entre outros. Essas aplicações definem um conjunto de operadores parecidos com operadores SQL (seleção, projeção, união, junção etc.) que trabalham em janelas de dados deslizantes. Os dados são produzidos e devem ser processados continuamente com baixa latência[1]. Por isso esses sistemas podem ser considerados de tempo real flexível.

Cada operador é implementado por um nodo que é um processo similar a um filtro do Anthill, porém não existe o conceito de múltiplas cópias transparentes ou múltiplas instâncias de um filtro, assim cada nó possui apenas uma cópia no pipeline, também chamado de *query network*. Os nodos se comunicam através de canais unidirecionais denominados fluxos, por onde passam mensagens na forma de tuplas. Quando um nó está ligado a outro por um fluxo, eles são considerados vizinhos, o que envia dados para um nó é chamado de vizinho *upstream* o que recebe dados gerados por um nó é o vizinho *downstream*. Ao contrário do Anthill cujos fluxos

[1]Baixa latência nesse contexto, é algo da ordem de segundos ou décimos de segundo

podem conectar M instâncias de filtro a N instâncias, os fluxos do Borealis são de 1 pra 1. As diferenças entre o Anthill e o Borealis podem ser vistas na Figura 2.1.

Para ter alta disponibilidade, o Borealis executa múltiplas cópias da *query network*. Como todas as cópias da *query network* devem produzir os mesmos resultados, os fluxos garantem a entrega das mensagens em ordem e a execução de todos os operadores deve determinista. Ainda assim existe uma fonte de não determinismo: se um operador possui vários fluxos de entrada, apesar de existir uma ordem entre as mensagens dentro de cada fluxo, não existe uma ordem entre as mensagens de um fluxo e de outro. Para garantir o determinismo nesse caso, foi criado o operador *SUnion*. Ele deve ser inserido à frente de todo operador O que possui mais de um fluxo de entrada para repassar todos os dados que receber de qualquer fluxo de maneira determinista para o operador O através de um único fluxo. Assim, como o estado e a saída dos operadores depende apenas das entradas recebidas e todas as cópias de cada operador recebem as mesmas entradas na mesma ordem, todas as cópias dos operadores terão os mesmos estados e emitirão as mesmas mensagens.

Caso um nó pare de receber dados (ou *heartbeats*) de um de seu fluxos de entrada causada pela falha de um vizinho *upstream*, por exemplo, inicialmente esse nó tentará obter a informação de uma cópia desse vizinho (utilizando os fluxos alternativos da Figura 2.1). Como os operadores têm o mesmo estado em todas as cópias da *query network*, a saída de uma cópia de um operador pode ser utilizada no lugar da saída de outra cópia que falhou. Caso nenhuma cópia de um vizinho *upstream* esteja funcionando, o Borealis ainda permite fazer um compromisso entre consistência e disponibilidade. O usuário pode especificar um limite de tempo no qual os operadores devem produzir alguma resposta: caso esse limite seja atingido e o operador não tenha recebido todas as entradas necessárias, ele é obrigado a produzir tuplas tentativas, que são tuplas geradas utilizando-se apenas informação parcial. As mensagens com essas tuplas tem um campo indicando que são tuplas tentativas que só devem ser consideradas em último caso e toda tupla que for gerada utilizando dados de tuplas tentativas também é marcada como tentativa. Os operadores tentam produzir o mínimo possível de tuplas tentativas para evitar que algum vizinho *downstream* tenha que tomar decisões utilizando dados incompletos.

Por fim, quando o sistema se recuperar da falha, os nós devem fazer uma reconciliação, ou seja, reexecutar suas computações com as entradas corretas. A reconciliação faz os operadores corrigirem seu estado interno e também estabilizar suas saídas substituindo tuplas candidatas por tuplas corretas, permitindo que os vizinhos *downstream* também reconciliem. Assim, o Borealis fornece resultados aproximados caso ocorram faltas de maior duração, que para esse tipo de aplicação é um compor-

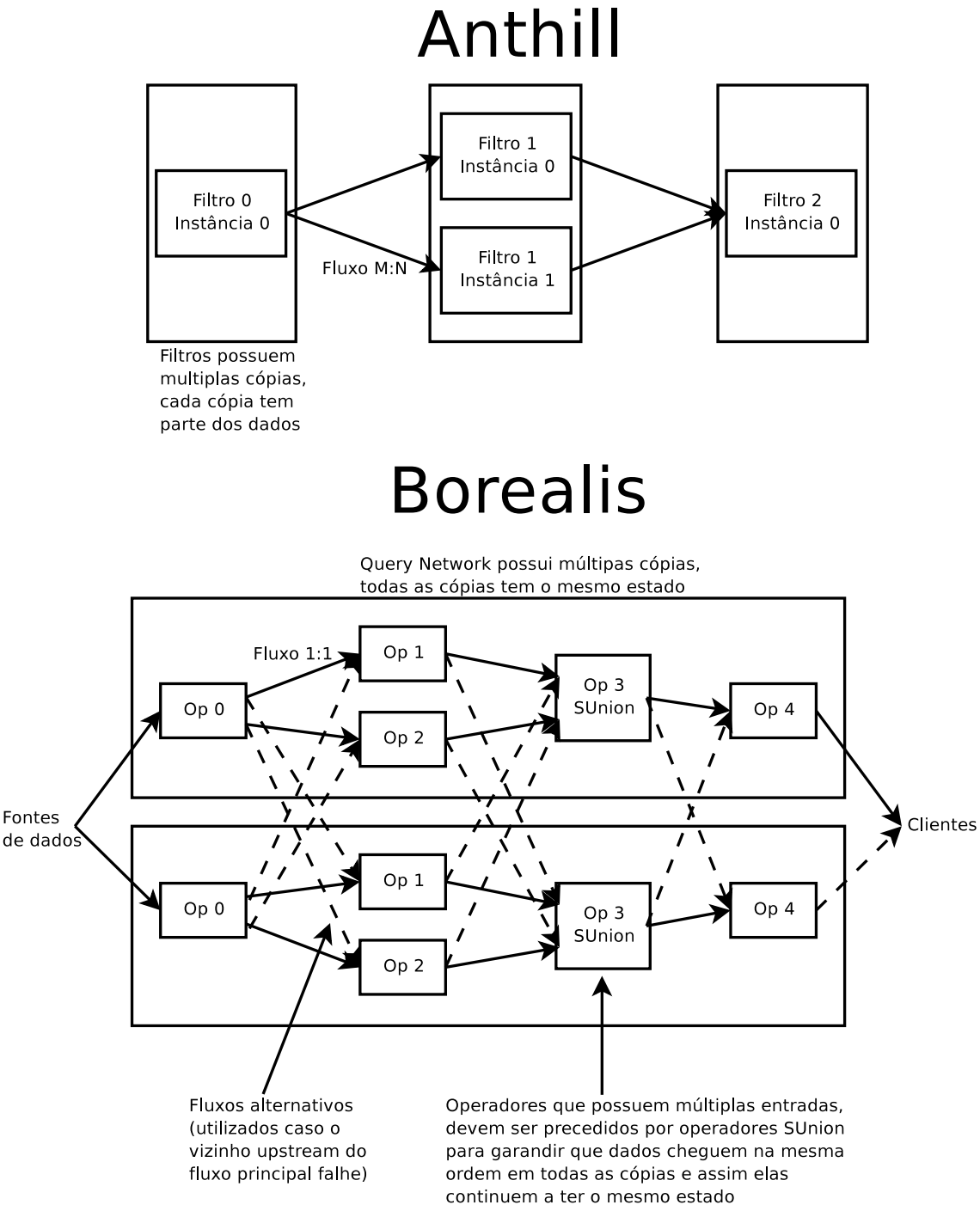


Figura 2.1: Diferenças entre o Anthill e o Borealis

tamento melhor que longas esperas, além de garantir que o sistema eventualmente irá gerar um resultado consistente.

2.2 Sistemas de tolerância a faltas

Por simplicidade e para garantir maior generalidade, quando se projeta um algoritmo de *checkpoint* distribuído, assume-se apenas que o sistema é um conjunto de processos que se comunicam em canais ponto a ponto confiáveis com preservação de ordem das mensagens. Apesar dos ambientes sobre os quais são apresentados os algoritmos nessa seção, PVM e MPI, terem modelos de programação muito diferentes do modelo do Anthill, todos esses sistemas no fundo são sistemas distribuídos que fazem comunicação em canais ponto a ponto. Como esse é o único pré-requisito para os algoritmos de *checkpoint* que serão apresentados, todos eles podem ser portados para o Anthill.

2.2.1 Sistemas de *Checkpoint* em MPI

A *Message Passing Interface* (MPI) é um padrão de computação paralela por passagem de mensagens em sistemas distribuídos em larga escala [GGHL+96, GHLL+98, GLS94, WGT99, MPI94, SOHL+96, SSB+03, BMPS03] definido pelo MPI Forum [MPI94]. Essa padrão é o mais utilizado atualmente para passagem de mensagens e suas implementações fazem parte do *middleware* de várias aplicações de alto desempenho [BDV94, GLDS96, GL96, Tea98].

Apesar de o padrão MPI propriamente dito não especificar nenhum tipo particular de comportamento de tolerância a faltas e as implementações de MPI mais usadas não forem projetadas para serem tolerantes a faltas, nos últimos anos MPI tem sido utilizada em *clusters* cada vez maiores, chegando a milhares de computadores. Por exemplo, o *cluster* Thunderbird dos laboratórios Sandia nos EUA tem 8000 processadores [MSDS05]. Com essa grande quantidade de processadores, a chance de algum deles falhar é muito grande, assim como a quantidade de computação perdida caso isso aconteça. Assim surgiu uma demanda muito forte por mecanismos de tolerância a faltas nas implementações MPI, que gerou vários trabalhos [BMPS03, SSB+03].

Uma das abordagens para fazer *checkpoint* em aplicações MPI é fazê-lo a nível de aplicação [BMPS03]. Para fazê-lo de forma transparente (sem modificação do código fonte da aplicação), utiliza-se uma biblioteca de *checkpoint* e um pré-compilador para inserir chamadas para ela. Para salvar os dados da aplicação são reimplementadas as funções de manipulação de memória: malloc(), realloc() e free().

Essa abordagem tem o problema de que o pré-compilador tem que inserir uma chamada no código para registrar o valor de cada uma das variáveis utilizadas pela aplicação. Para aplicações MPI mais simples, cuja execução passa a maior parte do tempo manipulando matrizes em *loops*, como as usadas nos experimentos em [BMPS03], não há uma penalidade muito grande. Entretanto, aplicações mais complexas, com muitas chamadas de função, podem ter seu desempenho prejudicado.

Outra abordagem (*checkpoint* a nível de sistema), foi utilizada pelo sistema de *Checkpoint*/Reinício do LAM/MPI [SSB+03], um sistema de *checkpoint* transparente para aplicações MPI, executando no ambiente LAM. Para fazer o *checkpoint* individual de cada processo foi utilizado o sistema de *checkpoint* de processos do Berkeley Lab, BLCR [DHR02], que é implementado como um módulo do núcleo do Linux. Ele possibilita o registro de *callbacks* que são disparadas no momento que o *checkpoint* ocorrer e que continuam sua execução após o *checkpoint* ser gravado ou quando o processo reiniciar. Essas *callbacks* permitem a aplicação parar temporariamente qualquer atividade que não possa ser gravada no *checkpoint*.

Para evitar inconsistências entre os *checkpoints* de processos diferentes, foi utilizada um protocolo coordenado bloqueante[2]. Para realizar o *checkpoint*, os processos se comunicam através de canais de comunicação alternativos, trocando informação sobre a quantidade de dados enviada para os outros processos e recebida deles. Em seguida, com base nessa informação, cada processo recebe o restante dos dados dos canais de comunicação MPI, drenando todos os dados em trânsito.

O algoritmo de *checkpoint* está sumarizado na Figura 2.2. Nele o `mpirun`, o programa executado pelo usuário para disparar a aplicação MPI, age como um gerente dos processos da aplicação MPI e ele é o processo para o qual o usuário enviará um sinal (sinal Unix) quando quiser iniciar um *checkpoint*. No caso de uma recuperação, ocorrem os eventos mostrados na Figura 2.3.

Recentemente, a implementação de MPI LAM foi remodelada como um *framework*, conhecido como System Services Interface (SSI), para os serviços providos pela infra-estrutura LAM. Ele é constituído de vários componentes que podem ser escolhidos em tempo de execução. Cada componente provê um serviço para o MPI e o ambiente de execução LAM (*daemons* que ajudam a disparar/monitorar processos MPI, assim como os *daemons* do PVM). Existem interfaces para módulos de comunicação ponto-a-ponto dependentes de interconexão, algoritmos de comunicação coletiva (MPI_gather, MPI_scatter, MPI_alltoall) e sistemas de *checkpoint*.

[2]Protocolo no qual os processos trocam mensagem e podem ser bloqueados para garantir que o estado gravado em todo *checkpoint* seja globalmente consistente, descrito na seção 4.2.2.

1. **gerente:** recebe uma requisição de *checkpoint*
2. **gerente:** propaga a requisição para cada processo MPI
3. **gerente:** indica para os processos MPI que está pronto para o *checkpoint*
4. **processos MPI:** comunicam-se através de canais de comunicação alternativos e esvaziam os canais de comunicação da aplicação
5. **processos MPI:** indicam (uns para os outros e para o gerente) que estão prontos para o *checkpoint*
6. **sistema de *checkpoint* de processos:** executa em todos os processos, dentro do MPI. Em cada processo, salva o estado do processo no armazenamento estável.
7. **processos MPI:** continuam sua execução

Figura 2.2: Protocolo de *checkpoint* do LAM/MPI

1. **gerente:** reinicia todos os processos a partir de suas imagens armazenadas
2. **processos MPI:** enviam suas novas informações de processo para o gerente
3. **gerente:** atualiza a tabela de processos e a reenvia para todos os processos
4. **processos MPI:** recebem a nova tabela de processos do gerente
5. **processos MPI:** reconstroem seus canais de comunicação
6. **processos MPI:** continuam a execução do estado armazenado

Figura 2.3: Protocolo de recuperação do LAM/MPI

A interface do serviço de *checkpoint* (CR SSI) consiste em:

initialize: usada para iniciar o sistema de *checkpoint* e registrar *callbacks* que serão invocadas no *checkpoint*

suspend: usada pela *thread* da aplicação MPI para suspender sua própria execução e liberar a trava das chamadas de sistema para a *thread* de *checkpoint*

disable checkpoint: usada quando o gerente entra em uma seção crítica onde não pode ser interrompido por uma requisição de *checkpoint*

enable checkpoint: usada quando gerente sai de uma seção crítica e pode voltar a receber requisições de *checkpoint*

finalize: usada para finalizar o sistema de *checkpoint*

Para implementar o *checkpoint*, foi necessário expandir a interface de comunicação ponto-a-ponto (Request Progression Interface, RPI). As seguintes funções foram adicionadas:

checkpoint: invocada quando chega uma requisição de *checkpoint*, geralmente para consumir mensagens em trânsito

continue: invocada após a conclusão do *checkpoint*, para o processo continuar sua execução

restart: invocada para restabelecer conexões e qualquer outra operação após a recuperação de uma falta

Embora a implementação de MPI LAM, tenha criado uma interface modular para abstrair detalhes de baixo nível do mecanismo de *checkpoint*, foi utilizado um mecanismo de *checkpoint* de processos que funciona dentro do núcleo do sistema operacional (um mecanismo implementado como biblioteca executando no nível de usuário é muito mais portável) e a estratégia utilizada na implementação do sistema de *checkpoint* não é escalável, pois bloqueia todos os processos da aplicação para realizar o *checkpoint* e exige que eles esvaziem seus canais de comunicação. Essa estratégia é suficiente para um ambiente estável e controlado, onde os *checkpoints* seriam obtidos a cada hora, por exemplo. Por outro lado em um ambiente mais caótico como a infra-estrutura do Google [DG04] com milhares de máquinas, onde a cada minuto algumas máquinas caem, essa estratégia de *checkpoint* tem um custo muito alto.

2.2.2 FTOP

O FTOP (*Fault Tolerant PVM*) [BGS02] é um sistema de *checkpoints* e recuperação por *rollback* transparente integrado ao PVM. Ele é implementado na forma de uma biblioteca totalmente em modo usuário, não necessitando mudanças no núcleo do sistema operacional.

Ele assume um sistema cujos componentes falham segundo o modelo de falhas de parada[3]. Esse sistema é constituído por:

- Um grupo de máquinas conectadas por uma rede de alta velocidade, que podem sofrer faltas e travar, rodando PVM.

[3]Modelo no qual sempre que um componente falha, ele para de funcionar imediatamente e isso é detectado. Descrito na seção 4.1 .

- Uma máquina que roda o coordenador do PVM, ou *Global Resource Manager* (GRM). Assume-se que ela não sofrerá faltas.
- Uma das máquinas (pode ser a mesma do GRM) será configurada como armazenamento estável (irá armazenar os arquivos de *checkpoint*, logs de mensagens, logs de arquivos etc). Assume-se que essa máquina também não sofrerá faltas.

Para gravar o estado dos processos no disco não é utilizado nenhum pacote de gravação de processos individuais. Ao invés disso, o FTOP lida diretamente com detalhes de baixo nível relacionados à gravação de estado de um processo no disco e sua restauração. Com isso ele funciona totalmente em modo usuário (assim como alguns pacotes), e pode ser facilmente portado para outros sistemas operacionais.

O FTOP utiliza um protocolo de *checkpoints* coordenado não-bloqueante [4], assim ele não sofre o chamado "efeito dominó" como o protocolo não coordenado e não tem um custo tão alto quanto o protocolo coordenado bloqueante.

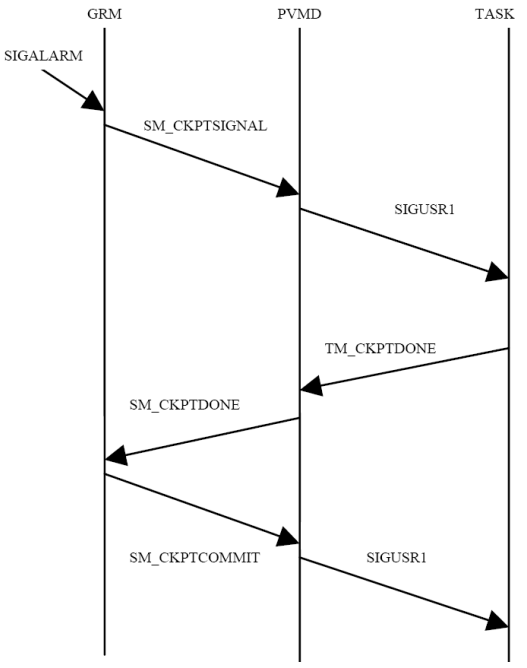


Figura 2.4: Protocolo de pontos de controle no FTOP

O protocolo de *checkpoint* (Figura 2.4) é iniciado quando o GRM recebe um sinal SIGALRM. O GRM envia uma mensagem SM_CKPTSIGNAL para todos os *daemons* da máquina virtual PVM. Quando o *daemon* PVM de uma máquina recebe

[4]Protocolo no qual os processos trocam mensagem para garantir que o estado gravado em todo *checkpoint* seja globalmente consistente, porém nunca é necessário bloquear um processo. Descrito na seção 4.2.2.

a mensagem SM_CKPTSIGNAL, ele envia o sinal SIGUSR1 para cada processo local gravar seu ponto de controle. Após terminar de gravá-lo, o processo ainda não pode enviar mensagens pois elas podem ser incluídas no ponto de controle local do outro, gerando um ponto de controle global inconsistente. Assim, deve-se garantir que todos os processos tenham gravado seus pontos de controle antes de deixá-los enviar mensagens novamente. Isso é feito da seguinte forma:

- Após cada processo gravar seu ponto de controle local, ele envia uma mensagem de reconhecimento (TM_CKPTDONE) para o `pvmd` da sua máquina.
- Quando o `pvmd` de uma máquina receber as mensagens de reconhecimento de todos os processos locais, ele envia uma mensagem SM_CKPTDONE para o GRM.
- Quando o GRM receber as mensagens SM_CKPTDONE de todos os `pvmds`, sabe-se que todos os processos gravaram seus pontos de controle e não há mais risco de ser gerado um ponto de controle global inconsistente. Então o GRM envia uma mensagem de confirmação (SM_CKPTCOMMIT) para todos os `pvmds`.
- Assim que os `pvmds` recebem a mensagem de confirmação, eles enviam um sinal SIGUSR1 para todos os processos locais, avisando-os que podem voltar a enviar mensagens.

Como pode haver mensagens em trânsito no momento do ponto de controle, os processos geram números de seqüência e gravam cada mensagem no envio, além de gravarem o número de seqüência da última mensagem recebida para poderem reenviar essas mensagens em trânsito na recuperação. Além disso, o FTOP tem um coletor de lixo para eliminar mensagens cujo reenvio não será mais necessário, liberando espaço no armazenamento estável.

Para detecção de faltas é utilizado o próprio mecanismo de detecção de faltas do PVM. Seus *daemons* periodicamente trocam mensagens para verificarem se os outros estão executando normalmente. Caso alguma máquina do sistema falhe (trave ou fique sem comunicação, parando de enviar mensagens; ver modelo de faltas de queda, seção 4.1), os `pvmds` não receberão mais mensagens dessa máquina e, após um *timeout*, avisarão o GRM. Quando o GRM é notificado de uma falta, ele identifica a máquina que caiu e quais processos estavam executando nela. Como preparação para a recuperação, são criados novos processos para substituir os que estavam na máquina perdida e todos os processos iniciados após a linha de recuperação são finalizados.

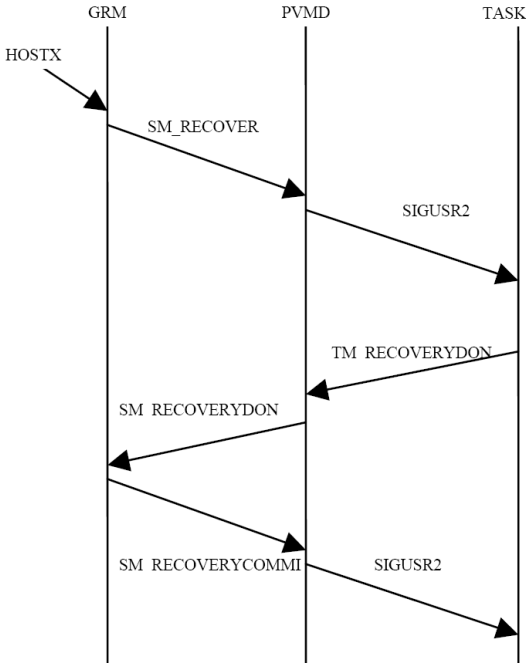


Figura 2.5: Protocolo de recuperação do FTOP

A recuperação da falta consiste na restauração de um estado anterior à falta. Para isso o FTOP utiliza um protocolo de confirmação de 2 fases (figura 2.5). Na 1ª fase, o GRM informa cada processo para voltar para o seu último ponto de controle. Como foi dito anteriormente o FTOP tem seu próprio sistema para lidar com detalhes de baixo nível da restauração do estado do processo. Esse sistema utiliza sinais Unix e assim que o processo sai da função de tratamento do sinal, ele terá voltado para o ponto da execução anterior ao ponto de controle. Quando todos processos terminarem de voltar ao último ponto de controle, o GRM envia uma mensagem de confirmação para os processos, liberando sua execução. Não é permitida a troca de mensagens entre processos durante a recuperação.

Para fazer o sistema voltar para o estado anterior à falta, não basta apenas recuperar os estados dos processos: é necessário também restaurar o estado dos canais de comunicação. O FTOP faz isso verificando os números de seqüência da última mensagem enviada e da última recebida e, finalmente, reenviando as mensagens que faltam.

Finalmente, após tudo estar restaurado, resta um problema: os processos continuarão enviando mensagens para aqueles que estavam na máquina que caiu e não para os novos processos criados para substituí-los. Para lidar com isso de forma transparente, o FTOP tem uma tabela de roteamento e redireciona mensagens destinadas aos processos que não existem mais para aqueles que os substituíram.

Capítulo 3

Modelo de programação

Neste capítulo apresentaremos o modelo de programação do Anthill, que será descrito a seguir. Na Seção 3.1 apresentaremos as tarefas e os espaços de dados, as extensões criadas para a aplicação ficar tolerante a faltas e finalmente, na Seção 3.2 será apresentada uma aplicação construída sobre o Anthill que foi adaptada para se tornar tolerante a faltas utilizando a API descrita na Seção 3.1.

O Anthill é um ambiente de programação, implementado em C, para o desenvolvimento de aplicações de processamento de dados em larga escala em ambientes distribuídos. Seu modelo de programação é o "filtro-fluxo rotulado", uma extensão do modelo de programação do DataCutter [BKC+01, BCC+02, AKSS01, NCK+03, NKCS03, SFB+02], chamado de programação filtro-fluxo. O modelo filtro-fluxo representa a aplicação como um *pipeline* de filtros onde um filtro pode ser ligado a outro através de um fluxo, como pode ser visto na Figura 3.1. O modelo "filtro-fluxo rotulado" se diferencia do filtro-fluxo por possuir uma política de envio chamada fluxo-rotulado, discutida no final dessa seção.

Cada filtro pode ser executado em várias máquinas e cada execução de um filtro em uma máquina é uma instância daquele filtro. Durante a execução da aplicação, cada instância de filtro repetidamente lê dados dos fluxos de entrada, executa o processamento sobre esses dados e então escreve os resultados pertinentes em um fluxo de saída.

O Paradigma filtro-fluxo oferece duas visões da aplicação, a utilizada pelo programador da aplicação e a visão do ambiente (Figura 3.1). Na visão utilizada pelo programador da aplicação, mais conhecida como disposição (*layout*), abstrai-se o número de instâncias de cada filtro e onde elas serão executadas. Já na visão do ambiente (utilizada pelo usuário da aplicação e escalonadores), também conhecida como colocação (*placement*) esse detalhes devem ser levados em conta para garantir o desempenho da aplicação.

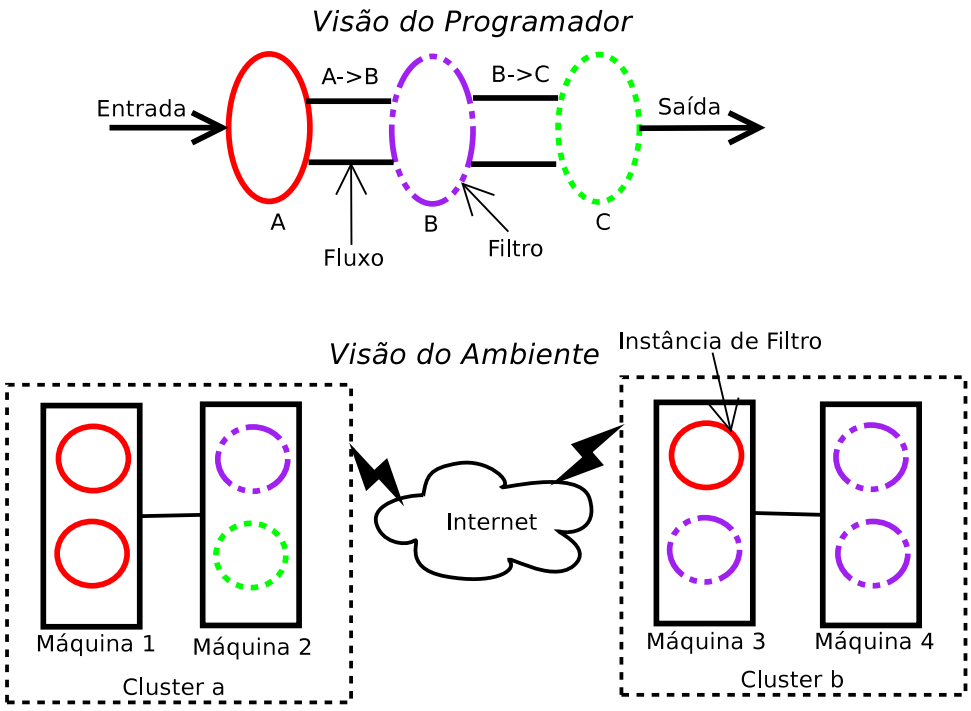


Figura 3.1: Visões oferecidas pelo paradigma Filtro-Fluxo

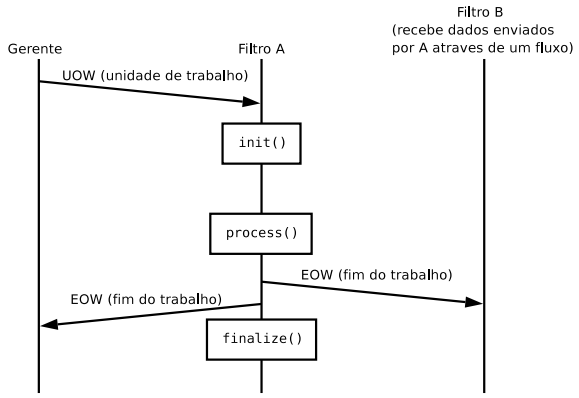


Figura 3.2: Modelo de programação do Anthill em alto nível

O código de um filtro é empacotado em uma biblioteca dinâmica e deve implementar as funções init(), process() e finalize(). Essas funções são executadas para cada unidade de trabalho (*Unit of Work ou UOW*) recebida pelo pipeline da aplicação (Figura 3.2). A função init() contém os procedimentos de iniciação do filtro. Nela, não é permitida a criação de tarefas, a recepção ou o envio de mensagens (ler ou escrever dados nos fluxos). A função process() implementa o código do filtro propriamente dito e a função finalize() é utilizada na liberação de recursos utilizados durante o processamento.

Um fluxo (*stream*) liga uma porta de saída de um filtro a uma porta de entrada de outro. Como ambos os filtros podem ter múltiplas instâncias, o fluxo é responsável por rotear as mensagens para as instâncias certas do filtro receptor. Para isso o fluxo possui várias políticas de envio: aleatória, revezamento (*round robin*), *broadcast* (envia mensagem para todas instâncias do filtro receptor), fluxo rotulado (o programador da aplicação implementa uma função que extrai um rótulo do conteúdo da mensagem e uma que recebe o rótulo e determina para qual instância a mensagem deve ser enviada) e multicast seletivo (semelhante a fluxo rotulado só que a mensagem pode ser enviada para mais de uma instância). As políticas de fluxo rotulado e multicast seletivo são extensões criadas no Anthill e não existiam anteriormente no DataCutter.

Quando um filtro A sair da função process(), o próprio Anthill irá avisar o gerente e quaisquer filtros que recebam dados de A, através de uma mensagem de fim de trabalho (*End of Work ou EOW*) que é enviada para o gerente e para todas as portas de saída de A (Figura 3.2). Quando um filtro chama ahReadBuffer() para ler dados de um fluxo e o Anthill recebe uma mensagem de EOW, ele não repassa essa mensagem para o filtro. Quando todas as instâncias do filtro escritor enviarem EOW, readBuffer() retorna EOW no filtro leitor, simulando o comportamento das funções de leitura de arquivo quando chegam ao fim do mesmo.

3.1 Tarefas e Espaços de Dados

A API de tarefas é uma extensão do modelo de programação "filtro-fluxo rotulado", que permite a implementação de blocos de código cuja execução é atômica sob o ponto de vista de tolerância a faltas. O bloco de código nunca será executado parcialmente: se ocorrer uma falta e ele não foi executado por completo em todos os processadores, pode-se considerar que ele não foi executado.

Uma tarefa é um bloco de código com início e fim bem definidos em cada processador, embora execute de forma assíncrona entre os processadores. Ela é identificada por um número (taskId), cuja unicidade deve ser garantida pela aplicação. Sua execução inicia após ter sido criada (sua tarefa mãe ter chamado ahCreateTask()) e todas as tarefas das quais ela depende estiverem terminado. A tarefa termina quando ela chama a função ahEndTask().

Apesar de ter um escopo bem definido, uma tarefa não é implementada como uma função para que uma instância de filtro seja capaz de executar várias tarefas simultaneamente (desde que não exista dependência de dados entre elas). Porém como a instância do filtro não possui várias *threads*, mesmo que várias tarefas possam

ser executadas em um dado instante, só uma o será. Essa tarefa que está em execução no momento é chamada de *tarefa corrente* e sempre só existe uma tarefa corrente por instância. Assim, sempre que a tarefa corrente ficar bloqueada esperando alguma mensagem, a instância do filtro poderá alternar para outra tarefa, tornando-a a nova tarefa corrente, diminuindo a possibilidade de ficar parada.

Durante sua execução, a tarefa pode ler dados de tarefas anteriores e, quando terminar, pode deixar dados para serem lidos por outras tarefas. Não são permitidas mensagens entre tarefas, a única forma de comunicação entre uma tarefa e outra é deixando dados no espaço de dados para a outra acessar.

Quando uma instância de um filtro chama ahCreateTask(), o Anthill cria essa tarefa nos outros filtros imediatamente, assim como o Anthill irá terminar a tarefa imediatamente em todos os outros filtros quando for chamada ahEndTask(). Devido a isso, a aplicação deve ter certeza de que não há mais nada naquela tarefa em todo o pipeline quando chamar ahEndTask(). Para permitir isso, assumimos que apenas um filtro inicia e termina as tarefas. Assim, mensagens da aplicação relacionadas à uma tarefa terão que passar por todo o pipeline e voltar para o filtro criador, que terá certeza de que não há mais nada a ser feito na tarefa em todo o pipeline ao disparar o seu término.

A tarefa mãe é a tarefa que está sendo executada quando é feita a chamada ahCreateTask(). Se nenhuma tarefa estiver executando, então a tarefa criada é uma tarefa órfã. A tarefa mãe passa explicitamente, na chamada de ahCreateTask(), quais tarefas cujos espaços de dados a tarefa criada pode acessar. Quando a tarefa mãe faz isso, a tarefa filha passa a ter dependência de dados em relação a todas tarefas que ela pode acessar. Uma tarefa só pode autorizar sua filhas a acessar dados de tarefas que ela própria tem acesso ou de tarefas filhas dessas tarefas. Essa propriedade permite ao Anthill ter completo conhecimento das dependências de dados entre as tarefas. O Anthill utiliza isso para descartar espaços de dados de tarefas que não estão mais sendo utilizados por nenhuma outra tarefa.

A tarefa filha começa quando todas as tarefas das quais ela depende terminarem, pois isso garante que os dados que ela lerá não podem mais ser modificados. Uma tarefa termina quando chama ahEndTask(). A partir desse momento, seu espaço de dados não pode ser mais alterado. A API de criação e término de tarefas é a seguinte:

- int ahUseTasks()
  Chamada em init() para avisar ao gerente que o filtro irá utilizar a API de tarefas.

- int ahCreateTask(int taskId, int *deps, int depSize, char *metadata, int metaSize)
  Cria a tarefa *taskId*, dando a ela permissão para acessar os espaços de dados das tarefas cujos identificadores estão no vetor *deps*, de tamanho *depSize*. Pode-se passar para a tarefa dados sobre o que ela terá de fazer no vetor *metadata* de tamanho *metaSize*. Não é permitida a criação de tarefas em init(). Retorna erro se a tarefa *taskId* já foi criada.

- int *ahGetTaskDeps(int taskId, int *depsSz)
  Retorna um vetor com as dependências da tarefa *taskId* e passa o tamanho do vetor em **depsSz*.

- int ahEndTask(int taskId)
  Termina a tarefa *taskId*.

O grafo de tarefas é o grafo onde cada tarefa é um nodo e as tarefas são conectadas por arestas às suas tarefas mães. A **borda do grafo de tarefas** é formada pelas tarefas que não estavam replicadas em todas instâncias dos filtros quando houve uma falta e têm que ser reexecutadas. Para cada tarefa na borda do grafo de tarefas, o Anthill executa uma *callback* para reiniciar a execução da tarefa. Apesar de não ser usado pelo programador da aplicação, o grafo de tarefas é armazenado como parte do estado das tarefas e é utilizado na recuperação, para re-executar as tarefas que estão na sua borda.

Sob o ponto de vista da tolerância a faltas, a execução da tarefa é atômica. Isso irá permitir que a unidade mínima de tolerância a faltas seja uma tarefa inteira e como não são permitidas mensagens entre tarefas, o sistema de tolerância a faltas não precisará lidar com mensagens. Durante sua execução (e mesmo após), a tarefa passa por quatro estados, são eles:

Criada: Para criar uma nova tarefa, a aplicação chama ahCreateTask(). Essa operação pode ser feita mesmo que não exista nenhuma tarefa executando no momento (como no início da execução do filtro). A nova tarefa começa no estado *criada*. Nesse estado, não é permitido que ela seja executada nem que seu espaço de dados seja alterado. A tarefa permanecerá nesse estado (criada) até que todas as tarefas das quais ela depende (dependência de dados) terminem. Os identificadores das tarefas às quais uma nova tarefa depende são passados pela tarefa mãe através do parâmetro *deps* da chamada ahCreateTask().

Executando: A tarefa passa para o estado *executando* quando todas as tarefas das quais ela depende terminam, permitindo que o Anthill inicie sua execução.

Várias tarefas podem estar nesse estado ao mesmo tempo, porém só uma delas será a tarefa corrente e terá seu código executado pela instância do filtro. Neste estado a tarefa será executada sempre que for a tarefa corrente e será permitido que ela leia dados deixados pelas tarefas que já terminaram, que altere os dados no seu espaço de dados e que crie tarefas filhas. As tarefas filhas começarão a executar quando todas as tarefas das quais elas dependem terminarem.

Finalizada: Quando a tarefa chama ahEndTask(), ela passa para o estado *finalizada*, terminando sua execução. Uma vez que a tarefa está terminada, os dados deixados no seu espaço de dados se tornam disponíveis para outras tarefas. Nesse estado não é mais possível a alteração do espaço de dados. Uma cópia do espaço de dados da tarefa e dos argumentos passados cada vez que a tarefa chamou *ahCreateTask()* é feita para o armazenamento estável (assim o sistema pode reexecutar as tarefas filhas se necessário). Essa replicação é realizada em segundo plano para que ela fique fora do caminho crítico da aplicação, liberando-a para executar outras tarefas (e não adicionar sincronização ao algoritmo, atrasando-o).

Replicada: Uma vez terminada a cópia do espaço de dados da tarefa e do grafo de tarefas para o armazenamento estável, a tarefa passa para o estado *replicada*. Uma vez que a tarefa passe para esse estado em todas as instâncias dos filtros, caso ocorra uma falta, não será necessária sua reexecução. Além disso, o conteúdo do espaço de dados se torna persistente, não sendo perdido caso ocorra uma falta.

Caso haja alguma falta, o Anthill reiniciará todos os filtros (Mais detalhes desse processo na seção 5.1.2). A recuperação de tarefas acontece entre as chamadas init() e process(). Para não ter que lidar com a possibilidade de reinício de tarefas no meio do código da aplicação, poluindo o mesmo, o Anthill permite que a aplicação registre uma função *callback* que será chamada após a recuperação de uma falha para reiniciar a execução das tarefas que estão na borda do grafo de tarefas (Figura 3.3). Essa função pode reiniciar a tarefa, enviar mensagens para outros filtros como se a tarefa estivesse executando normalmente e permitir que a aplicação volte ao seu *loop* principal, onde ficará esperando uma resposta dos outros filtros. A função *callback* é registrada em init(). Se a aplicação não a registrar, será executada a função padrão, que irá apenas recriar a tarefa.

- int ahRegisterRecoveryCallback(RecoveryCallback_t *callback)
  Chamada pelo filtro na função init() para informar o Anthill qual função é a *callback* de recuperação.

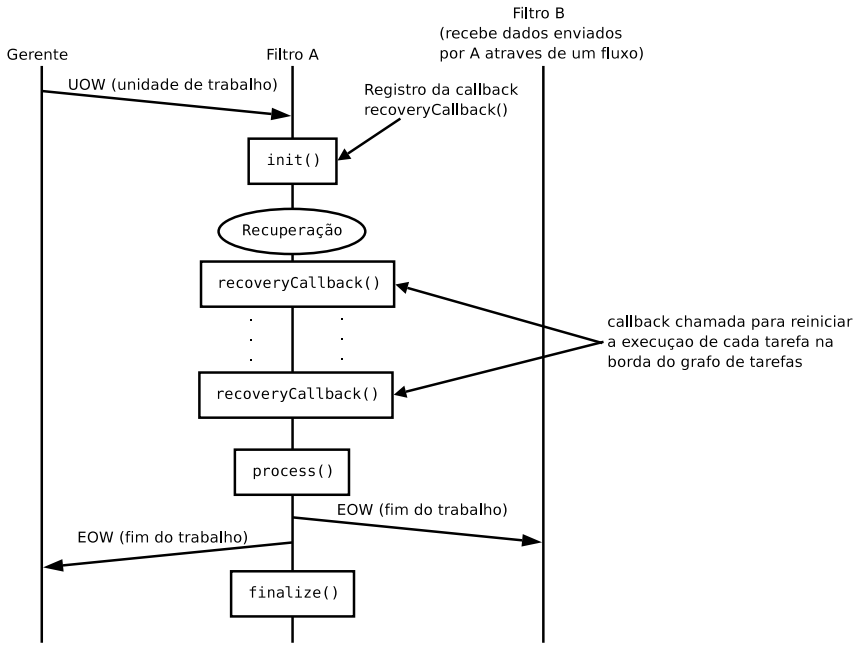


Figura 3.3: Recuperação de uma falta

- int RecoveryCallback_t(int taskId, int *deps, int depSize, char *metadata, int metaSize)
  Protótipo da *callback* de recuperação. Quando ocorrer uma falta, o Anthill chamará essa função para reexecutar as tarefas que estarão na borda do grafo de tarefas. Os argumentos passados para a *callback* serão os mesmos que forem passados para a função ahCreateTask(), quando a tarefa for criada (antes da falta). A *callback* deve chamar ahCreateTask() com os argumentos recebidos para recriar a tarefa, enviar mensagens para outros filtros e fazer o processamento necessário para que o filtro possa continuar a execução da tarefa no seu *loop* principal.

Cada tarefa possui um espaço de dados associado a ela. Esse espaço de dados está dividido entre as instâncias de cada filtro. Uma instância não pode acessar o espaço residente em outra (se isso fosse permitido, teríamos que sincronizar os espaços de dados das instâncias do filtro e isso teria um custo muito alto). A aplicação deve utilizar o fluxo rotulado para rotear cada mensagem para a instância que possui os dados necessário para processá-la.

A tarefa que está sendo executada no momento também é chamada de tarefa corrente. É nela que são feitas as alterações quando a aplicação chama alguma função de manipulação do espaço de dados. No início da execução do filtro, ou quando todas as tarefas terminarem, não existirá nenhuma tarefa corrente, mas

assim que a primeira tarefa começar a executar, ela se torna a tarefa corrente e existirá uma tarefa corrente enquanto alguma tarefa estiver executando. Quando existirem várias tarefas em execução, pode-se mudar de tarefa corrente usando a função setCurrentTask(), descrita a seguir.

Para manipulação dos espaços de dados de uma tarefa, o Anthill fornece uma API semelhante às de manipulação de dicionários. A chave é um string e o valor é um vetor de bytes que pode ter qualquer tamanho. A API é a seguinte:

- int ahSetCurrentTask(int taskId)
  Faz a tarefa corrente ser *taskId*, assim operações de alteração sobre o espaço de dados serão feitas sobre ela. A tarefa *taskId* deve estar em execução.

- int ahGetCurrentTask()
  Retorna a tarefa corrente ou -1 se não existir uma tarefa corrente.

- int ahPutData(char *key, void *val, int valSize)
  Insere o valor *val* de tamanho *valSize* na chave *key* dentro do espaço de dados da tarefa corrente. Caso esta chave já esteja sendo usada, o valor anterior será sobrescrito.

- void *ahGetData(int taskId, char *key, int *dataSz)
  Faz a tarefa corrente acessar o valor associado à chave *key* no espaço de dados da tarefa *taskId*. Retorna um ponteiro apontando para uma área de memória com uma cópia do valor e passa o tamanho da área em **dataSz*.

- int ahRemoveData(char *key)
  Apaga o valor associado á chave *key* do espaço de dados da tarefa corrente.

Fizemos as instâncias do filtro verem suas partes do espaço de dados como um *hash* pensando nas aplicações de mineração de dados, que são as aplicações iniciais do Anthill. Não sabemos se isso vai funcionar bem com outros tipos de aplicação, como as que trabalham com matrizes, que são a grande maioria das aplicações de computação de alto desempenho. Caso seja necessário podemos criar outra forma de acesso ao espaço de dados.

3.2 Aplicação Exemplo: Paralelização do Algoritmo Eclat

Uma das aplicações implementadas no Anthill é uma paralelização do algoritmo Eclat [VJF+04]. O algoritmo Eclat [ZPOL97] realiza mineração de regras de asso-

ciação assim como seu antecessor, o algoritmo Apriori [AIS93, AS94], um dos mais conhecidos algoritmos de mineração de dados.

Regras de associação são compostas de 3 partes: o antecedente, o conseqüente e métricas como suporte e confiança. O antecedente é um conjunto de itens que se comprados, com uma probabilidade determinada pela confiança, também será comprado o item que está no conseqüente. Dada um base, que consiste de transações, onde cada transação é uma lista com os produtos adquiridos em cada compra. Você quer saber do total de pessoas que compraram "Coca-Cola", quantas também compraram "Mentos". Essa informação é dada pela seguinte regra de associação:

"Coca-Cola" $\longrightarrow$ "Mentos". Com suporte 1235 e confiança 20%.

Essa regra diz que das pessoas que compraram "Coca-Cola", 20% também compraram "Mentos" e que 1235 pessoas compraram "Coca-Cola" e "Mentos". Nessa regra o antecedente é "Coca-Cola", o conseqüente é "Mentos" e as métricas são o suporte e a confiança (existem outras métricas, mas essas são as mais utilizadas).

3.2.1 Algoritmo Eclat

A tarefa de mineração de regras de associação foi introduzida em [AIS93]. Essa tarefa pode ser dividida em dois passos. O primeiro passo consiste em encontrar todos os *itemsets* freqüentes, onde um *itemset* é um conjunto de itens e um *itemset* freqüente é um *itemset* que ocorre na base com uma freqüência maior que a freqüência mínima especificada pelo usuário. Não são gerados todos os *itemsets* possíveis pois se a base de dados possuir m itens, o número de *itemsets* possíveis é o conjunto potência do conjunto de itens da base, assim existiriam 2^m *itemsets* possíveis. A poda pela freqüência mínima elimina a maioria dos 2^m *itemsets*, onde praticamente todos os eliminados não seriam interessantes, pois ocorrem poucas vezes na base de dados. O segundo passo é gerar as regras de associação a partir dos *itemsets* freqüentes.

No algoritmo Eclat a geração de *itemsets* é feita da seguinte forma: No início da sua execução, ele lê a base de dados, que está no formato horizontal (uma transação por linha), e a converte para o formato vertical (listas invertidas de *itemsets*). Após essa transformação, os itens infrequentes são eliminados.

Os itens dentro dos *itemsets* são armazenados de forma ordenada. Assim não existe o *itemset* "BA", só o "AB". O prefixo de um *itemset* é um conjunto ordenado formado por todos os itens do mesmo, exceto o último. Por exemplo, o prefixo do

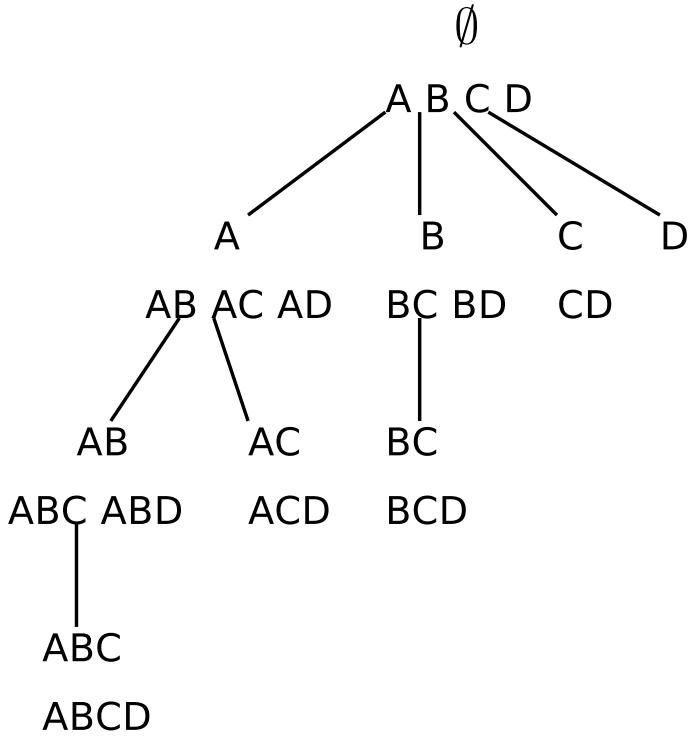


Figura 3.4: Árvore de prefixos de uma execução do algoritmo Eclat

itemset "AB" é "A" e o do *itemset* "DEF" é "DE". Caso o *itemset* possua apenas um item, seu prefixo será $\emptyset$.

Durante a execução do algoritmo, *itemsets* de tamanho i, com um mesmo prefixo (pX e pY por exemplo onde p é o prefixo comum, X e Y são itens, sendo que $X < Y$), são combinados para formar *itemsets* de tamanho $i + 1$ (pXY no exemplo, cujo prefixo é pX). As listas invertidas desses novos *itemsets*, contendo as transações onde eles aparecem, são obtidas realizando interseções das listas invertidas dos *itemsets* combinados. Após ser gerada a lista invertida de um novo *itemset*, o algoritmo verifica se ele é freqüente e o descarta se não for.

Assim, todos os *itemsets* são combinados, produzindo *itemsets* com prefixos maiores. Isso irá gerar uma árvore de prefixos como a da Figura 3.4 que será estendida combinando-se *itemsets* com o mesmo prefixo até que não existam mais pares de *itemsets* de mesmo prefixo para serem combinados.

A partir dos *itemsets* freqüentes, a geração de regras de associação é relativamente fácil. Praticamente todos os algoritmos utilizam algum procedimento parecido com o do Algoritmo 1.

```
forall itemset y do
    forall item I ∈ y do
        x = y − I;
        confiança = suporte(y)/suporte(x);
        if confiança > min_conf then
            printf("%s ⟶ %s; suporte: %f, confiança: %f\n", x, y,
            suporte(y), confiança);
        end
    end
end
```

Algorithm 1: Procedimento para geração de regras de associação

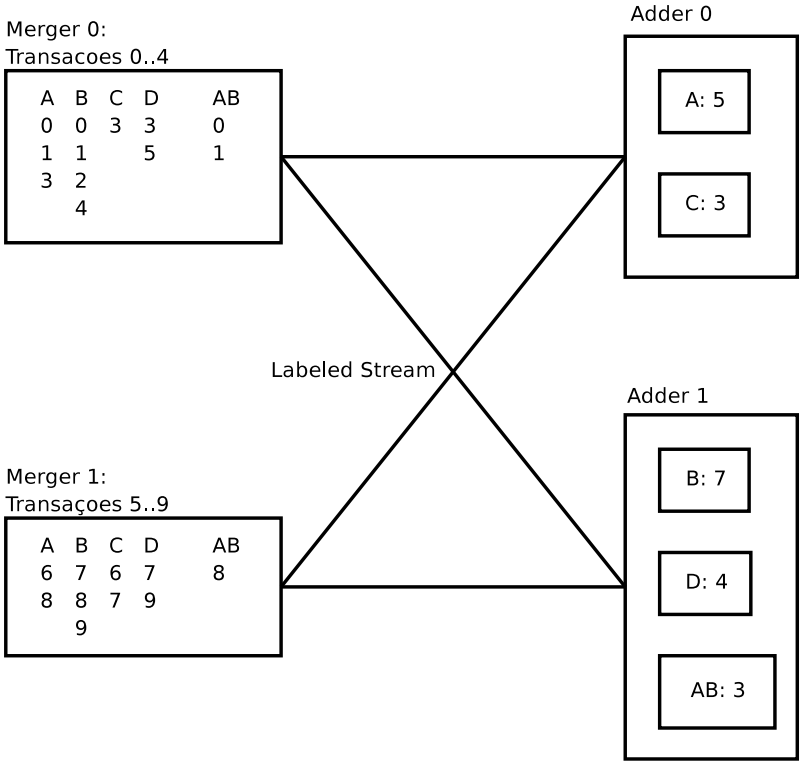


Figura 3.5: Distribuição dos *itemsets* nos filtros

3.2.2 Paralelização do Eclat no Anthill

A paralelização do Eclat no Anthill [VJF+04] possui dois filtros: Merger e Adder. Cada instância do Merger armazena uma parte das listas invertidas dos *itemsets*, correspondente à parte da base de dados pertencente a aquela instância. O algoritmo assume que a base (entrada do algoritmo) já está distribuída em partes iguais e no início da execução, cada instância do filtro Merger irá ler justamente a parte que é de sua responsabilidade. Por exemplo, na Figura 3.5, o arquivo de entrada do Merger 0 terá as transações 0 a 4 e o arquivo do Merger 1 terá as transações 5 a 9.

Para combinar dois *itemsets* pA e pB, de tamanho i, e gerar um *itemset* pAB, de tamanho $i+1$, cada instância do Merger faz a interseção da sua parte da sua parte das listas de pA e pB e envia o tamanho da lista gerada para o filtro Adder. O Adder recebe as contagens locais de cada Merger, verifica se o *itemset* é freqüente e envia essa informação para todas as instâncias do Merger, que irão fazer novas combinações utilizando o *itemset*, se ele for freqüente, ou descartá-lo caso contrário. Assim é feita a contagem de ocorrências de forma distribuída.

O filtro Merger pode ter várias instâncias, já o Adder pode ser multiplamente instanciado desde que tenha suporte de um algoritmo de terminação global. Isso ocorre porque cada instância só sabe se ela própria está ociosa e não tem informação suficiente para prever se outras instâncias ainda podem declarar que algum *itemset* é freqüente, iniciando o processamento de várias combinações [VJF+04].

3.2.3 Tolerância a faltas na implementação paralela

No Eclat, a tarefa (unidade mínima de tolerância a faltas) foi modelada como o procedimento de geração da lista invertida de um *itemset* e verificação de freqüência, descrito na seção anterior. Uma vez que para gerar a lista invertida de um *itemset* é necessário fazer a interseção das listas de outros dois *itemsets* que possuem o mesmo prefixo, a tarefa que gera um *itemset* depende das tarefas que geram os dois *itemsets* que serão utilizados como entrada.

Quando a tarefa termina, o Anthill grava seu estado no armazenamento estável, evitando que uma falta posterior obrigue o algoritmo a reprocessar o *itemset*. No estado da tarefa também são armazenados os parâmetros passados para a chamada *ahCreateTask()* na criação da tarefas filhas. Na recuperação, o Anthill chama a *callback* definida pelo filtro Merger do Eclat para redisparar as tarefas na borda do grafo de tarefas. Isso faz com que tempo de execução das *callbacks* durante a recuperação seja proporcional ao tamanho da borda do grafo de tarefas.

A leitura da base de dados e sua conversão do formato horizontal para o vertical são feitos em uma única tarefa. Como esse processo envolve todos os itens da base de uma vez, a única divisão possível seria dividir a leitura da base da conversão, mas isso implicaria no armazenamento de um grande volume de dados no armazenamento estável, cujo custo é maior que o de reler a base caso ocorra uma falta. Assim, se ocorrer uma falta durante a leitura da base e sua conversão para o formato vertical (que são feitas em uma só passo, pois o algoritmo não armazena mais que uma linha da base em formato horizontal) toda a base terá que ser lida novamente.

Após a leitura da base, o Eclat percorre seu espaço de busca, utilizando busca em largura. Ele verifica se os *itemsets* de tamanho i (os próprios itens da base

na primeira iteração) têm a freqüência mínima, em caso afirmativo, combina-os de modo a gerar novos *itemsets* de tamanho $i+1$ para serem verificados. Cada *itemset* é processado por uma tarefa e sua lista invertida fica armazenada no espaço de dados dessa tarefa. Como existe esse mapeamento de um pra um entre os *itemsets* e as tarefas, o grafo de tarefas será o mesmo que o grafo do espaço de busca percorrido pelo algoritmo, com os *itemsets* gerados durante sua execução.

Caso ocorra alguma falta, o tempo de recuperação é proporcional à largura da borda do espaço de busca no momento. Quanto mais larga a borda no momento da falta, mais tarefas serão perdidas e terão que ser reexecutadas, aumentando o tempo de recuperação (veja Seção 6.2). No início da execução, a borda do espaço de busca se alarga muito rápido pois quanto maior o tamanho dos *itemsets* sendo processados, maior é o número de combinações de itens que podem ser feitas para gerar um novo *itemset*. Após um certo ponto, a maioria dos *itemsets* gerados passa a ser infrequente, diminuindo as possibilidades de combinações de *itemsets* e conseqüentemente estreitando a borda do espaço de busca até que a execução acabe.

3.3 Sumário

Nesse capítulo foi apresentada a API que os filtros devem seguir para utilizar o sistema de tolerância a faltas. Apesar dessa API facilitar a implementação de uma aplicação tolerante a faltas, o programador ainda tem muito trabalho em adaptar sua aplicação para a utilização da API, principalmente pela necessidade de dividir a aplicação em tarefas. Uma solução para esse problema é a utilizar um compilador para traduzir o código da aplicação em um código já dividido em tarefas que utiliza a API.

Capítulo 4

Modelos de tolerância a faltas

Para discutirmos o modelo de tolerância a faltas utilizado no nosso sistema, temos que apresentar as opções existentes na literatura e, para discutir algumas desses opções, temos que conhecer os modelos de faltas existentes e quais tipos de falta cada modelo consegue suportar. Assim, na seção 4.1 descreveremos os modelos de faltas existentes, já na seção 4.2 descreveremos modelos de sistemas de tolerância a faltas e, finalmente, na seção 4.3 discutiremos os modelos de faltas e de tolerância utilizados no nosso sistema e as razões das nossas escolhas.

4.1 Modelos de faltas

Todo componente de *hardware* ou *software* tem uma especificação de como ele se comporta, para que sejamos capazes de usá-lo. A especificação pode determinar tanto a resposta do componente para algum estado inicial e seqüência de entrada, quanto o intervalo de tempo que a resposta deve ocorrer. Se o componente se comporta de acordo com a especificação, dizemos que ele está *correto*, caso contrário, ele *falhou* [Cri91].

Um sistema é composto por vários componentes, sendo que a falha de um deles é vista pelo sistema como uma falta. Um sistema tolerante a faltas prevê em sua especificação a possibilidade de um determinado número de seus componentes falharem e continuará funcionando corretamente mesmo que isso ocorra.

Para tolerar um tipo de falta, primeiramente o sistema deve ser capaz de detectá-la. Para permitir isso, foram criados modelos de faltas, que especificam como um componente se comporta quando falha e como essa falta (a falha do componente) pode ser detectada pelo sistema. Apesar desses modelos muitas vezes serem descritos em termos de componentes de *hardware*, eles podem ser aplicados em praticamente qualquer sistema computacional.

No modelo de falhas de parada (*fail-stop*), um componente pode falhar a qualquer momento. Quando ele falha, ele pára de produzir qualquer saída e de interagir com o resto do sistema, e os outros componentes detectam a falha automaticamente. Além disso, o sistema não faz nenhuma transição de estado incorreta [SS83]. Este modelo de faltas compreende faltas mais simples como:

Queda de processos: Um processo que estava executando na máquina é terminado prematuramente devido a um erro de programação, exceção não tratada ou falta de algum recurso como memória ou espaço em disco. Quando isso acontece, qualquer software de monitoração que esteja executando na mesma máquina detectará a queda da aplicação imediatamente. Aplicações em outras máquinas serão notificadas pelo sistema operacional da máquina onde ocorreu a falta, que fecha todas conexões com a aplicação que caiu. Existem várias alternativas para evitar esse problema, que permeia áreas inteiras da computação como engenharia de software e verificação formal. Algumas delas são ferramentas de análise estática de código [AE02, Fos02, EL02, Hol02, BV, NMW02, KA05, CW02, HL02] e verificação formal de programas (limitada ou não) [YTEM04, Hol00, BPHV00, HJMS03, HDR04, CKL04, CCG+04]. Além disso há técnicas para recuperação de módulos que falharam [CCF04, PBB+02, CF01], impedindo que isso comprometa o sistema inteiro. Essas técnicas também podem ser utilizadas para reiniciar módulos periodicamente, ou sob suspeita de mal funcionamento, sendo capaz de lidar com faltas que não seguem o modelo de parada, mesmo sem possuir um detector apropriado.

No modelo de queda (*crash*), o componente falha do mesmo jeito que no modelo de parada, mas sem garantia de detecção. Esse modelo também é chamado de falha silenciosa (*fail-silent*) [PBS+88]. Geralmente esse tipo de falta é tratada definindo um prazo máximo para uma operação e se esse prazo for atingido, considera-se que houve uma falta. São exemplos de faltas silenciosas:

Queda de máquina: Causado por falta de energia, queima de fonte ou cabos de força rachados ou desconectados. Faz a máquina ser desligada abruptamente (podendo causar perda ou corrupção de dados). Outras máquinas detectam essa falta através de *timeouts* nas conexões TCP. Essa é uma falta transiente, pois eventualmente a energia voltará, a máquina será religada, voltando a responder requisições. Para evitar essa falta geralmente são usadas máquinas com fonte redundante, sendo que cada fonte recebe energia de uma rede elétrica diferente.

Travamento de máquina: Geralmente causado por falha de *chipset*, erro em *drivers* de dispositivo ou no sistema operacional. Faz a máquina parar de processar, enviar ou receber pacotes de rede. Para as outras máquinas é como se ela tivesse caído, exceto pelo fato que necessitará de intervenção externa para voltar a funcionar. Existem sistemas para automatizar essa operação, desligando definitivamente e religando a máquina como o HP Integrated Lights-Out Standard (iLO) [Hew], o Linux Networks Icebox^{TM} Cluster Management Appliance [Lin]. Além de *watchdog timers* [MM88] que requerem alteração periódica de seu registrador por parte do sistema operacional: caso ele fique muito tempo sem alterar esse registrador (quando a máquina trava, por exemplo) ele a reiniciará automaticamente.

Travamento da aplicação: Geralmente causado por erro de programação na própria aplicação, fazendo-a ficar retida em um *loop*, ou causando *deadlock* em aplicações paralelas. Como o processo da aplicação não termina, até os processos que estão rodando na mesma máquina não conseguem distinguir se a aplicação travou ou apenas está demorando muito para responder. Geralmente os erros que só aparecem em aplicações distribuídas, como *deadlock*, são mais complexos de serem descobertos pois não são deterministas e o número de iterações possíveis (e conseqüentemente erros) entre os processos e módulos é muito grande, o que dificulta sua reprodução e a inspeção manual de cada iteração. Devido a toda essa complexidade, as técnicas de verificação automática de código citadas anteriormente (ferramentas de análise estática de código e verificação formal), em particular o projeto *Meta-Level Compilation* [AE02, YTEM04], têm ganhado destaque ultimamente. Já as técnicas de *microreboots* recursivos atacam falhas silenciosas de maneira diferente: uma vez que o reinício de um módulo tem baixo custo e não afeta o resto do sistema, o módulo suspeito de travamento é sempre reiniciado.

SCSI timeout: Algumas falhas em discos SCSI simplesmente fazem com que o disco pare de responder e quando o tempo máximo da operação estipulado pelo sistema operacional (que pode chegar a ser uma ordem de grandeza maior que o tempo normalmente gasto na operação) é atingido, ele retorna erro.

O modelo de falha por omissão assume que o componente falha deixando de responder a algumas entradas [CASD85]. Se o componente permanece inativo após uma falha (modelo de queda), então essa falha é um caso especial de falha por omissão no qual o componente não responde mais a entrada alguma após a primeira falha. São exemplos de faltas por omissão:

Switch descarta pacotes: Quando o *buffer* de um *switch* ou roteador excede sua capacidade, ele começa a descartar pacotes. Como o protocolo TCP retransmite pacotes perdidos e a própria perda de um pacote é utilizada no controle de transmissão, roteadores mais modernos têm políticas de descarte de pacotes para controlar congestionamento [MR91].

Perda de pacotes: Erros de roteamento em *switches* e roteadores podem enviar pacotes para lugares errados, causando sua perda.

Descarte de mensagens: É um erro comum que servidores de email mal configurados classifiquem mensagens legítimas como *spam* e as descartem silenciosamente. Nesse caso não existe mecanismo de retransmissão.

Cancelamento de requisições: Se durante uma operação de leitura ou escrita do servidor Web Apache houver algum erro de E/S, ele trata isso cancelando a requisição [PBB+02].

No modelo de falha de temporização, um componente falha comportando de forma funcionalmente correta, mas fora do intervalo de tempo permitido [CASD85]. Falhas de temporização podem ser classificadas como falhas de temporização por adiantamento e falhas de temporização por atraso, que fazem o componente continuar a produzir a resposta certa, mas apresentar uma taxa de resposta muito mais baixa que o esperado. Essas últimas também são chamadas de falhas de desempenho (*fail-stutter*) [ADAD01] ou falhas de lentidão (*slow-down failures*) [ST90].

Em sistemas de grande escala, mesmo utilizando hardware homogêneo, é muito grande a probabilidade de algum componente ter desempenho ligeiramente inferior ao esperado. Sistemas que usam esse modelo de faltas diferenciam faltas absolutas (de correção) de faltas de desempenho. Faltas absolutas são tratadas segundo o modelo de parada ou de queda. Já faltas de desempenho são tratadas não assumindo que todos os componentes apresentam exatamente o mesmo desempenho, não utilizando paralelismo estático, monitorando o desempenho dos componentes e realizando balanceamento de carga em tempo de execução. Exemplos de faltas de desempenho:

Congestionamento na rede: Caso a rede seja chaveada, o *switch* passa a apresentar latência muito alta e pode até descartar pacotes. Mesmo que a camada de transporte os reenvie, o usuário pode notar uma redução de desempenho do sistema. Em redes não chaveadas, por exemplo, que usam *hubs* Ethernet, a situação pode se tornar muito pior. Nessas redes, à medida que o tráfego aumenta, aumentam as colisões, fazendo as máquinas retransmitirem os pacotes,

gerando ainda mais tráfego. No pior caso, todas as máquinas ficam o tempo todo retransmitindo pacotes e ninguém consegue transmitir nada.

Erros de leitura corrigíveis: Uma máquina com um disco ruim pode sofrer erros corrigíveis freqüentes que diminuem sua velocidade de leitura de 30MB/s para 1MB/s. Segundo [DG04], esse tipo de falta é muito comum na infraestrutura do Google.

Desativação de otimizações: A maioria do hardware existente hoje em dia possui várias otimizações: caches, modos de transferência mais rápidos ou que dispensam intervenção da CPU. Algumas dessas otimizações foram projetadas para contornar gargalos extremamente restritivos. Quando o dispositivo apresenta mal comportamento ou, devido a erro no software de gerenciamento do mesmo, essas otimizações podem ser desativadas, reduzindo drasticamente o desempenho do dispositivo. Recentemente, o servidor de arquivos (que serve os arquivos pessoais de cada usuário) do nosso laboratório desligou o DMA do seu disco, fazendo a velocidade de leitura dele cair de 30MB/s para 3.5MB/s. Como praticamente todos os programas no Linux têm arquivos de configuração que ficam na pasta pessoal, tornou-se impraticável trabalhar no nosso laboratório. Segundo [DG04], um erro no código de inicialização da máquina fazia as caches do processador serem desativadas. Isso fazia a máquina ficar mais de cem vezes mais lenta.

Uma falha é chamada de arbitrária ou bizantina se não podemos assumir nada sobre o comportamento de falha do componente [LSP82]. Um componente que falhou pode tomar ações desconhecidas, inconsistentes ou até maliciosas.

Hardware com Defeito de Fabricação: O erro mais famoso dos processadores da Intel, conhecido como Pentium FDIV bug [Pri95], era um erro em 5 posições de uma tabela de 1066 posições, que poderia causar imprecisão em resultados de divisões a partir da 4^a casa decimal.

Inversão de bits de memória: A inversão de bits é um erro de memória muito comum. Até processadores de baixo custo como o AMD Semprom têm proteção por paridade ou ECC em todas suas caches [Inc04]. Esse tipo de erro pode ser utilizado para fins maliciosos, como violação de *smart cards* [GA03] ou execução de código arbitrário em sistemas que utilizam verificação de tipo como mecanismo de segurança.

Quanto mais genérico é um modelo de faltas, mais difícil é a sua detecção, pois maior é o número de dados que devem ser verificados, mais intrusivos são os testes,

menos premissas pode-se assumir e maior a redundância necessária para recuperar das faltas. Isso faz com que quanto mais genérico seja o modelo de faltas maior o custo da sua detecção e recuperação.

Quanto à possibilidade de recuperação após a falha, os modelos podem ser divididos em dois grandes grupos: o primeiro assume que componentes que falham serão capazes de se recuperar em um prazo aceitável, também conhecido como *cash-recovery* [BG05] e um que não assume isso, ou seja, um componente que falhou pode exigir intervenção humana por período prolongado para se recuperar ou a falha pode ser irrecuperável, é necessário substituir o componente.

4.2 Modelos de tolerância a faltas

Existem duas grandes abordagens para a tolerância a faltas em sistemas distribuídos: a abordagem da máquina de estados, que utiliza replicação do sistema e a abordagem de pontos de controle (*checkpoints*) e recuperação por volta ao passado (*rollback-recovery*), que utiliza replicação no tempo, periodicamente gravando o estado da aplicação em um armazenamento estável e, em caso de falta, volta para o último estado gravado e re-executa a computação.

Na abordagem da máquina de estados, a aplicação é estruturada como um serviço, que é implementado por múltiplos processos, para obter tolerância a faltas [Sch90]. O serviço é caracterizado como uma máquina de estados que mantém variáveis que são modificadas em resposta aos comandos recebidos. O estado dessa máquina e sua saída dependem apenas da seqüência de comandos recebidos recebida, independente de tempo e de qualquer outra atividade do sistema.

A tolerância a faltas é obtida executando réplicas dessa máquina de estados em vários processadores. Como todas as réplicas começarão no mesmo estado e receberão os comandos na mesma ordem, todas terminarão no mesmo estado e produzirão a mesmo saída. Caso um dos processadores falhe, basta copiar o estado de um que está funcionando para que ele fique consistente com o resto do sistema novamente.

Caso o sistema possa sofrer faltas bizantinas, a saída produzida pelos processadores só é utilizada após votação. Caso a saída de um ou mais processadores seja diferente da dos demais será utilizada a saída produzida pela maioria. Essa configuração precisa de $2t + 1$ processadores para suportar até t faltas simultâneas. Caso o sistema só sofra faltas de parada, $t + 1$ processadores são suficientes para suportar t faltas simultâneas.

A abordagem da máquina de estados é muito indicada para sistemas de tempo real, pois nesses sistemas, a resposta deve ser gerada dentro de uma restrição de

tempo e caso ultrapasse essa restrição considera-se que o sistema falhou. Assim nesses sistemas não é admissível que uma falta cause qualquer parada ou interrupção. A abordagem da máquina de estados consegue atender as restrições dos sistemas de tempo real replicando o estado do sistema em várias máquinas e caso uma delas caia, as outras continuam a assumir a operação sem que haja nenhum atraso.

Essa abordagem tem um custo muito alto, principalmente tratando-se de sistemas de computação distribuída. Ela exige o uso de protocolos de *multicast* confiável que têm um custo alto para garantir a entrega de mensagens, bem como sua ordem. Além disso, exige a sincronização e ordenação de todos os eventos, impedindo a utilização eficiente dos recursos e tornando-se especialmente vulneráveis a faltas de desempenho. Por fim, exige replicação do hardware, dobrando ou triplicando o custo do sistema (dependendo da redundância utilizada). Em sistemas de computação de alto desempenho as aplicações geralmente não tem restrições de tempo real, assim, quando uma máquina sofre uma falta, podemos nos dar ao luxo de parar o sistema para fazer uma recuperação e de re-executar computação caso seja necessário. Devido a esses fatores, essa solução torna-se muito desvantajosa, principalmente em sistemas cujo objetivo é aumentar o desempenho, assim concentraremos a discussão na abordagem de *checkpoints*.

A abordagem de *checkpoints* e recuperação por *rollback* modela uma aplicação distribuída como conjunto de processos que se comunicam trocando mensagens. Assume-se que os processos têm acesso a algum tipo de armazenamento que sobreviverá mesmo que algum processo falhe, chamado de armazenamento estável. Durante a execução da aplicação, periodicamente (nos *checkpoints*) o sistema grava no armazenamento estável o estado de toda a aplicação, composto pelo estado de todos os processos e todos os canais de comunicação. Caso a aplicação sofra alguma falta, o sistema restaurará o estado da mesma para um estado anterior à falta, trazendo todos os processos de volta ao estado armazenado consistente mais recente. Nessa abordagem, normalmente assume-se um modelo de falhas de parada.

O armazenamento estável, utilizado nessa abordagem, pode ser implementado de várias formas, dependendo do tipo de falhas que o sistema deve suportar. Ele não precisa necessariamente ser um meio de armazenamento persistente como um disco: se o sistema tem que tolerar apenas falhas de uma máquina de cada vez, o armazenamento estável pode ser implementado usando a memória RAM de outras máquinas. Se as falhas forem transientes (é uma premissa), o próprio disco local da máquina pode ser usado. Finalmente, no caso das falhas das máquinas serem permanentes, o armazenamento estável deve estar localizado fora da máquina onde o processo está executando, ficando em um servidor NFS, por exemplo.

A seguir descreveremos em que nível pode ser implementado o *checkpoint* e na Seção 4.2.2 classificaremos os protocolos de *checkpoint* quanto ao tipo de coordenação utilizado para garantir a consistência do estado global sendo gerado.

4.2.1 Nível de funcionamento

O *checkpoint* pode ser implementado tanto em nível de sistema quanto a nível de aplicação. Cada estratégia tem suas vantagens e desvantagens, que serão descritas a seguir.

Checkpoints a nível de sistema são transparentes para aplicação. Isso permite que sejam utilizados em aplicações existentes sem modificação de código, além de facilitar a codificação de novas aplicações, pois o programador não precisa se preocupar em escrever código para lidar com a criação do *checkpoint*. Por outro lado, a criação de *checkpoints* a nível de aplicação torna necessária a inserção de chamadas às funções da API fornecida pelo sistema de *checkpoint* para funcionar. Esse trabalho pode ser feito por um pré-processador, diminuindo em parte esse problema.

Ao contrário de *checkpoints* a nível de sistema, onde se pode salvar o estado da aplicação a qualquer momento, *checkpoints* a nível de aplicação podem salvar o estado da aplicação apenas quando a ela faz chamadas à API de *checkpoint*. Assim não se pode atrasar *checkpoints* ou disparar *checkpoints* forçados quando chegar uma mensagem que gere dependências entre os intervalos de *checkpoints*[1].

Apesar de todas essas vantagens, *checkpoints* a nível de sistema têm uma séria desvantagem: como essa técnica não diferencia quais dados são realmente importantes para a aplicação e quais podem ser descartados, ela obriga a gravação de todo o estado da aplicação. Existem algumas técnicas como exclusão de memória que amenizam esse problema mas mesmo assim o custo ainda é alto. Se considerarmos aplicações intensivas em dados, que são o alvo do Anthill, esse custo se torna proibitivo. Por isso escolhemos implementar *checkpoints* a nível de aplicação, onde o programador pode dizer explicitamente quais dados devem ser salvos e quais não, mantendo assim apenas dados semanticamente relevantes para a aplicação.

4.2.2 Coordenação

Em relação ao modo como se coordenam para garantir a consistência do estado global sendo gerado, os protocolos de *checkpointing* podem ser divididos em três

[1]Essas dependências podem provocar *rollbacks* em cascata se acontecer uma falta. Para mais detalhes, veja *efeito dominó* na Seção 1.3

tipos: não coordenado, coordenado e induzido por comunicação (*communication-induced*).

Nos protocolos não coordenados não há coordenação entre os processos para garantir a consistência do *checkpoint* sendo gerado. Somente há coordenação na recuperação de uma falta. Embora o *checkpoint* nesses protocolos tenha menor custo, só na recuperação o sistema descobriria que o *checkpoint* de algum processo poderia depender de uma mensagem enviada em um momento que não faz parte do *checkpoint* global. Em algumas aplicações, o padrão de comunicação leva a uma probabilidade muito grande de ocorrerem dependências como essa de maneira encadeada. Assim, qualquer falta causa um efeito dominó de *rollbacks* que pode trazer a execução de volta para o seu início. Se o tempo médio entre faltas da plataforma não for muito maior que o tempo de execução da aplicação, o efeito dominó pode chegar a impedir o término da execução pois sempre haverá uma falta que obrigará a aplicação a descartar quase todo seu progresso. Além disso, o protocolo só irá descobrir que um conjunto de estados individuais de processos formam um estado global inconsistente quando tentar recuperar de uma falta, obrigando-o a guardar várias versões do estado de cada processo.

Já nos protocolos coordenados, os processos trocam mensagens para gerar uma visão de *checkpoint* consistente entre todos os processos. Esses protocolos podem ser divididos em duas categorias: bloqueante e não bloqueante. Os protocolos bloqueantes param o sistema integralmente para gerar o *checkpoint* global. Exemplos de *checkpoint* bloqueante são o sistema de *checkpointing* por hardware do IBM SP/2, o sistema de *Checkpoint*/Reinício do LAM/MPI [SSB+03] e bibliotecas de *checkpointing* que aproveitam barreiras da aplicação para gravar o estado do sistema. Como *checkpoints* bloqueantes paralisam toda a aplicação, eles são muito caros. Já os protocolos não-bloqueantes não necessitam parar toda a aplicação, tornando-se uma alternativa muito mais interessante. O protocolo mais famoso dessa categoria é o de Chandy-Lamport [CL85].

Protocolos de *checkpoint* induzido por comunicação evitam o efeito dominó e ainda assim permitem que os processos salvem seus estados de forma independente [Tre05, EAWJ02]. Porém, para garantir o progresso da linha de recuperação (O conceito de linha de recuperação é apresentado na seção 1.3), essa independência é restringida obrigando processos a fazerem *checkpoints* adicionais. Os *checkpoints* feitos pelos processos de forma independente são chamados de *checkpoints* locais, enquanto os *checkpoints* que os processos foram obrigados a fazer são chamados de *checkpoints* forçados. Esses protocolos adicionam informação própria à cada mensagem da aplicação e os processos receptores utilizam essa informação para determinar

se eles têm que fazer um *checkpoint* forçado. O *checkpoint* forçado deve ser efetuado antes de a aplicação processar o conteúdo de mensagens, possivelmente prejudicando o desempenho da aplicação. Nesses sistemas é desejável reduzir o número de *checkpoints* forçados sempre que possível. Ao contrário dos protocolos coordenados, no *checkpoint* induzido por comunicação não são trocadas mensagens especiais de coordenação.

4.3 Modelos utilizados

Nesta seção discutiremos os modelos de faltas e os mecanismos de tolerância a faltas utilizados neste trabalho, assim como as justificativas dessas escolhas e descreveremos o que assumimos quanto ao funcionamento de sistema.

Assumimos um sistema constituído de:

- Um conjunto de nós de processamento que podem falhar segundo o modelo de falha de queda (explicado na Seção 4.1), conectados por uma rede de alta velocidade.

- Um dos nós executa o gerente do Anthill. Assumimos que esse nó não cai. Por enquanto isso não é um problema pois como só há um gerente, sua chance de falhar é muito menor que a chance de uma das dezenas ou até centenas de instâncias de filtros. Caso seja necessário, é possível aumentar a confiabilidade do gerente utilizando redundância de hardware [Tre05], porém estes detalhes estão fora do escopo deste trabalho.

- Se o armazenamento estável for implementado como um nó especial no sistema (um servidor NFS, por exemplo), também assumimos que este nó não sofrerá faltas.

- Os canais de comunicação físicos seguem o modelo de falta silenciosa. Porém o PVM possui retransmissão de mensagens e garante a ordem de entrega entre duas máquinas, além possuir de mecanismo de envio periódico de mensagens para detectar faltas silenciosas. Assim podemos assumir que o PVM é capaz de nos fornecer canais FIFO confiáveis que se comportam segundo o modelo de faltas de parada.

Assumimos que o sistema não sofrerá uma segunda falta enquanto o mecanismo de tolerância a faltas não conseguir recuperar da primeira. Caso isso aconteça atualmente o Anthill será obrigado a reiniciar toda a aplicação. Como o período que o sistema fica em recuperação é muito menor que o tempo total de execução da

aplicação, esperamos que reinícios de toda a aplicação aconteçam raramente e não cheguem a atrapalhar seu desempenho.

Quanto à aplicação, assumimos que ela é constituída por processos resilientes. Processos resilientes são processos cuja execução (ou parte dela) pode ser repetida várias vezes, sempre gerando o mesmo resultado. Para que isso seja possível, eles não podem fazer E/S ou trocar mensagens com um processo fora do sistema. Assumimos também que não há restrições de tempo real sobre a aplicação, ou seja, podem ser feitas replicações no tempo e reexecuções sempre que necessário. O único efeito externo da aplicação/tarefa deve ser alterações no espaço de dados e uma saída ao final da execução.

Como modelo de falha para os componentes desse sistema, escolhemos o modelo de queda ou falha silenciosa. Essa escolha se deve a duas razões: se um processo parar de responder, podemos assumir que ele falhou (mesmo que isso gere falsos positivos não será problema pois todos os dados da aplicação estão replicados) e esse modelo já cobre a maior parte das faltas sofridas pelos sistemas computacionais atualmente.

Nosso sistema trata da queda de processos. Quedas de nodo são mapeadas como quedas de todos os processos naquele nodo (por isso o servidor de replicação fica em outra máquina). Caso a falta seja no canal de comunicação com algum nodo, o sistema considerará que o nodo falhou e todos os processos daquele nodo serão excluídos do sistema. Assim, mesmo que o canal se recupere e esses processos ainda estejam ativos e voltem a enviar mensagens, eles não vão comprometer o resto do sistema.

Quanto à abordagem de tolerância a faltas, utilizaremos pontos de controle (*checkpoints*) e recuperação por volta ao passado (*rollback*). Como foi discutido na Seção 4.2, a abordagem de máquina de estados tem um alto custo, além de exigir replicação do *hardware*, apresentando um custo muito alto.

Quanto ao nível onde será implementado o *checkpoint*, o faremos a nível de aplicação, pois a nível de sistema seríamos obrigados a salvar todos os dados da aplicação, o que geraria uma quantidade muito grande de informação a ser armazenada no armazenamento estável. Como utilizamos *checkpoints* a nível de aplicação, implementamos uma API de tarefas, na qual cada tarefa tem um espaço de dados, onde a aplicação deve armazenar os dados que não podem ser perdidos na ocorrência de uma falta. Embora o programador tenha que mapear sua aplicação em tarefas e especificar as dependências de dados entre tarefas, isso é muito menos trabalhoso que lidar com a consistência da aplicação e a tolerância a faltas diretamente.

Além disso, esse trabalho pode ser automatizado mediante o uso de pré-compiladores

que fariam a análise do código de uma aplicação Anthill convencional e gerariam uma aplicação tolerante a faltas utilizando tarefas e espaços de armazenamento.

Uma vez que no modelo de programação da API de tarefas não permitimos a troca de mensagens entre tarefas, podemos utilizar *checkpoints* não coordenados sem correr o risco de sofrer o efeito dominó. Não há mensagens que façam uma tarefa replicada em todo o sistema depender de uma tarefa não replicada, obrigando o mecanismo de recuperação a descartá-la. Assim, em caso de falha, só perderemos progresso das tarefas que não foram terminadas.

Capítulo 5

Implementação

Neste capítulo, explicaremos em detalhes como funciona o Anthill e o mecanismo de tolerância a faltas implementado. A seguir será dada uma breve descrição da arquitetura interna de funcionamento do Anthill. Na Seção 5.1 será descrito o funcionamento do mecanismo de tolerância a faltas e nas Seções 5.1.1 e 5.1.2 serão descritos seus módulos principais: o módulo cache e o protocolo de recuperação.

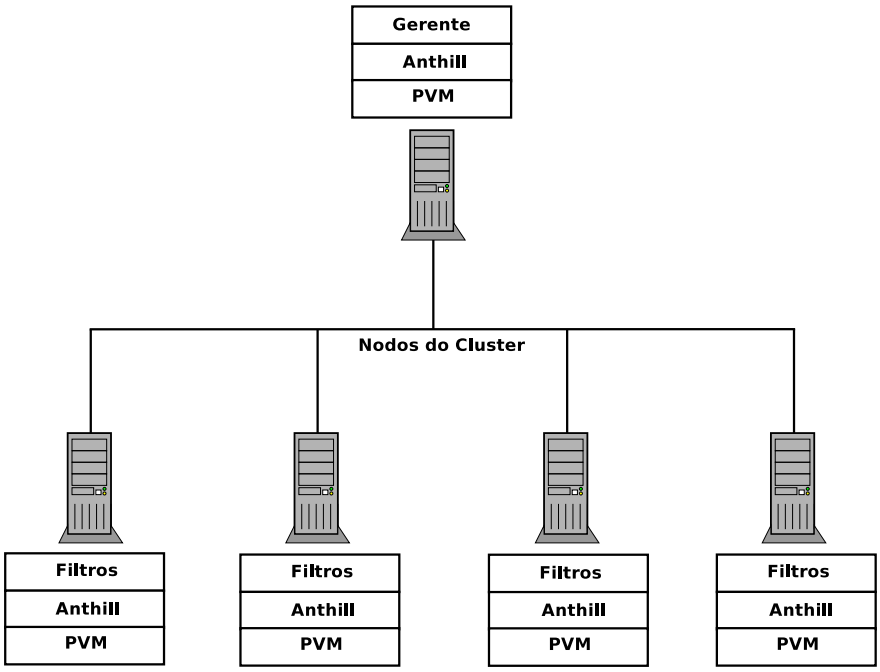


Figura 5.1: Arquitetura do Anthill

A arquitetura do Anthill é, em sua forma mais genérica, composta por um gerente (*Manager*) e filtros da aplicação, como pode ser visto na Figura 5.1. O papel do gerente é ler o arquivo de configuração do sistema, garantir que as aplicações sejam

executadas de acordo com o especificado e cuidar da monitoração das aplicações em execução. Antes de começarem a execução os filtros recebem, por parte do gerente, informações necessárias para a execução de suas tarefas. Estas informações são, dentre outras:

- O nome do filtro;
- Quantas instâncias o filtro possui;
- Qual posição da instância entre as instâncias do filtro;
- Quantos e quais são seus fluxos de entrada; e
- Quantos e quais são os fluxos de saída.

No desenvolvimento do Anthill utilizou-se a biblioteca de computação paralela PVM (*Parallel Virtual Machine*). A escolha dessa ferramenta na implementação do sistema está, principalmente, ligada à simplicidade com que as abstrações do paradigma *filtro-fluxo* podem ser expressas através das funções existentes no PVM, além da estabilidade desse ambiente: há mais de 10 anos o PVM vem sendo utilizado pela comunidade científica, inclusive em ambientes de produção. Essa biblioteca também gera abstrações que permitem a conexão de máquinas com *hardware* heterogêneo em uma única "máquina virtual", que aparece para o programador como uma máquina paralela. Outras características importantes da biblioteca são:

- Interoperabilidade: uma aplicação desenvolvida sobre o PVM pode ser migrada para várias arquiteturas diferentes.
- Reconfiguração dinâmica: pode-se iniciar/terminar processos durante a execução da aplicação.
- Tolerância a faltas: O PVM possui mecanismos simples de detecção de faltas que indicam quando um processo terminou sua execução de forma anormal.
- Comunicação simplificada: O PVM possui mecanismos de comunicação simples que são utilizados para a construção das funções implementadas.

5.1 Mecanismo de tolerância a faltas

O mecanismo de tolerância a faltas não pode reduzir oportunidades de assincronia da aplicação. Se o sistema impuser alguma sincronia à aplicação, processadores

que poderiam estar trabalhando ficarão ociosos, esperando o processador mais lento terminar alguma tarefa. Essa ociosidade causada por excesso de sincronia é um dos grandes fatores que limitam a escalabilidade de muitas aplicações. Além disso, o progresso da aplicação deve ser independente do progresso do sistema de tolerância a faltas. No pior caso isso ocasionará uma perda maior de progresso da execução da aplicação quando ocorrer uma falta.

A implementação do sistema de tolerância a faltas funciona totalmente em modo usuário e a nível de aplicação. Essa abordagem tem sua vantagens e desvantagens. As vantagens são a portabilidade, o sistema funciona em qualquer SO Unix, pois não possui partes que executam dentro do *kernel* e a facilidade de depuração (não é que o sistema seja fácil de ser depurado, mas é muito menos difícil que a depuração de módulos do *kernel* e sistemas de restauração de processos a partir de arquivos de *checkpoint*). A desvantagem é que o programador da aplicação fica responsável por inserir chamadas ao sistema de *checkpoint*, mas ainda assim isso é mais fácil que implementar a tolerância a faltas em si.

O sistema de tolerância a faltas não altera o número de instâncias de cada filtro da aplicação. O máximo que ele consegue fazer é mover instâncias de uma máquina para outra. Para que isso fosse possível, seria necessário redividir os dados dos processo, o que não é nada trivial. Por exemplo, na aplicação Eclat, redividir as listas invertidas do filtro *Merger* (Seção 3.2.2) e garantir que as mensagens cheguem às listas corretas mesmo que elas mudem de máquina é uma tarefa bastante complexa.

Sob o ponto de vista da tolerância a faltas, a execução da tarefa é atômica. Isso permite que o grão de tolerância a faltas seja uma tarefa inteira e, como não são permitidas mensagens entre tarefas, o sistema de tolerância a faltas não precisará lidar a comunicação entre as instâncias dos filtros na replicação do estado de uma tarefa (e do estado da instância do filtro como um todo, que não é mais que o estado de todas as tarefas). Assim as instâncias dos filtros podem fazer a replicação do estado de uma tarefa terminada sem se importarem com o que está acontecendo com as outras instâncias. Na implementação atual, essa replicação consiste em gravar no armazenamento estável (um servidor de arquivos) os dados colocados no espaço de dados e as tarefas filhas da tarefa replicada (para construção do grafo de tarefas na recuperação). Isso requer muito menos espaço de armazenamento e banda de rede que replicar todo o conteúdo da memória do processo. Para garantir que na recuperação não sejam lidos arquivos corrompidos, a replicação é feita para um arquivo em um diretório temporário (no próprio armazenamento estável) e após ela terminar, o arquivo é movido para o diretório final. Uma vez que o arquivo temporário e o final vão ficar no mesmo sistema de arquivos, essa movimentação é

apenas uma mudança no nome do arquivo que é atômica[1].

Além da API de tarefas, o sistemas de tolerância a faltas possui outros dois módulos: o módulo cache, que funciona como uma cache do armazenamento estável e o protocolo de recuperação. O módulo cache será descrito na seção 5.1.1 e o protocolo de recuperação será descrito na seção 5.1.2.

5.1.1 Módulo Cache

O módulo cache tem esse nome, pois ele funciona como uma cache local do armazenamento estável. Cada instância de filtro tem uma instância do módulo cache, que armazena a parte do espaço de dados da tarefa utilizado pela instância. Após o término da tarefa, quando seus dados não podem mais ser modificados, ela é replicada para o armazenamento estável.

Nesse primeiro protótipo o armazenamento estável é o sistema de arquivos, assim pode-se utilizar o disco local, ou um servidor de arquivos remoto (como mencionado, nosso modelo de sistema assume que se o armazenamento estável for implementado como um nó especial no sistema, ele não sofrerá faltas).

A abstração fornecida pela cache é muito parecida com a do modelo de tarefas, exceto que a manipulação do espaço de dados trata apenas dos dados locais de cada instância de filtro.

Sempre que uma tarefa termina, ela é enviada pela cache para a *thread* de replicação, que escreve seus dados no armazenamento estável: o espaço de dados da tarefa, os argumentos passados para a chamada *ahCreateTask()* durante a criação da tarefa e os argumentos passados durante a criação de todas as tarefas filhas, para que sistema possa reexecutar as tarefas filhas se necessário. Mais exatamente, são gravados o identificador, as dependências e o campo com metadados passado para a tarefa. Não é necessário fazer sincronização entre a *thread* da aplicação e a *thread* replicadora pois ambas acessam os dados da tarefa terminada apenas para leitura.

5.1.2 Protocolo de Recuperação

Como já foi dito na seção 4.1, assumimos que o sistema não sofrerá faltas enquanto estiver em recuperação, logo o protocolo de recuperação não lida com faltas durante sua execução.

[1] Na verdade existe uma pequena janela de tempo na qual os dois nomes estão apontando para o arquivo sendo movido, mas isso é irrelevante. Se houvesse uma falta dentro dessa janela de tempo, o nome temporário do arquivo seria apagado durante a recuperação sem afetar o nome permanente nem o conteúdo do arquivo.

Protocolo de Recuperação:
1. Gerente termina todos os processos restantes
2. Gerente reinicia todos os processos
3. Gerente envia para todos os processos a "Unidade de trabalho" ou *Unit of Work (UOW)* com um aviso, informando os processos que ocorrerá uma recuperação
4. Cada processo monta uma lista das tarefas que conseguirá recuperar
5. Processos enviam suas listas de tarefas que podem ser recuperadas para o gerente (lista de tarefas recuperada localmente)
6. Gerente faz interseção de todas as listas de tarefas recuperadas de todos os processos, gerando a lista de tarefas recuperadas globalmente
7. Gerente envia resultado das interseções a todos os processos
8. Processos recuperam as tarefas que estão na lista enviada pelo gerente
9. Para cada tarefa recuperada:
10. Processos recuperam os dados da tarefa do armazenamento estável
11. Se a tarefa estiver na lista de tarefas a serem reexecutadas, ela será retirada de lá
12. (A tarefa será inserida nessa lista pela sua tarefa mãe, na linha 16).
13. Tarefa é reinserida no hash de tarefas e seu estado é modificado para "terminada"
14. Para cada tarefa filha:
15. Se a tarefa filha não estiver no hash de tarefas (não existir):
16. Insere tarefa na lista de tarefas a serem reexecutadas
17. Para cada tarefa na lista de tarefas a serem reexecutadas:
18. Faz a tarefa mãe da tarefa a ser reexecutada se tornar a tarefa corrente
19. Reexecuta a tarefa chamando a função *callback* registrada pelo filtro em init() (Mais detalhes desse processo estão na seção 3.1).

Figura 5.2: Protocolo de recuperação

Quando ocorrer uma falta, o PVM poderá detectá-la através de *timeouts* nas mensagens e *pings* periódicos entre seus *daemons*. Assim que o PVM detectá-la, ele avisará o gerente, informando o identificador do processo afetado. Caso a falta tenha causado a queda de uma máquina, serão enviadas outras mensagens informando esse fato.

O sistema não deixa processos órfãos pois o gerente finaliza os processos restantes e redispara tudo novamente. Caso a falta tenha sido a aparente queda de uma máquina, causada por uma falha de rede, o *daemon* PVM que ficou isolado irá matar os processos locais e terminar. Mesmo que sobre algum processo e, após a recuperação, ele tente se comunicar novamente com a aplicação, o próprio PVM rejeitará suas mensagens pois ele e sua máquina serão marcados como mortos e excluídos da máquina virtual do PVM.

Assim que for notificado de uma falta, o gerente inicia o protocolo de recuperação (Figura 5.2). Primeiramente ele reinicia todas instâncias dos filtros, terminando-as

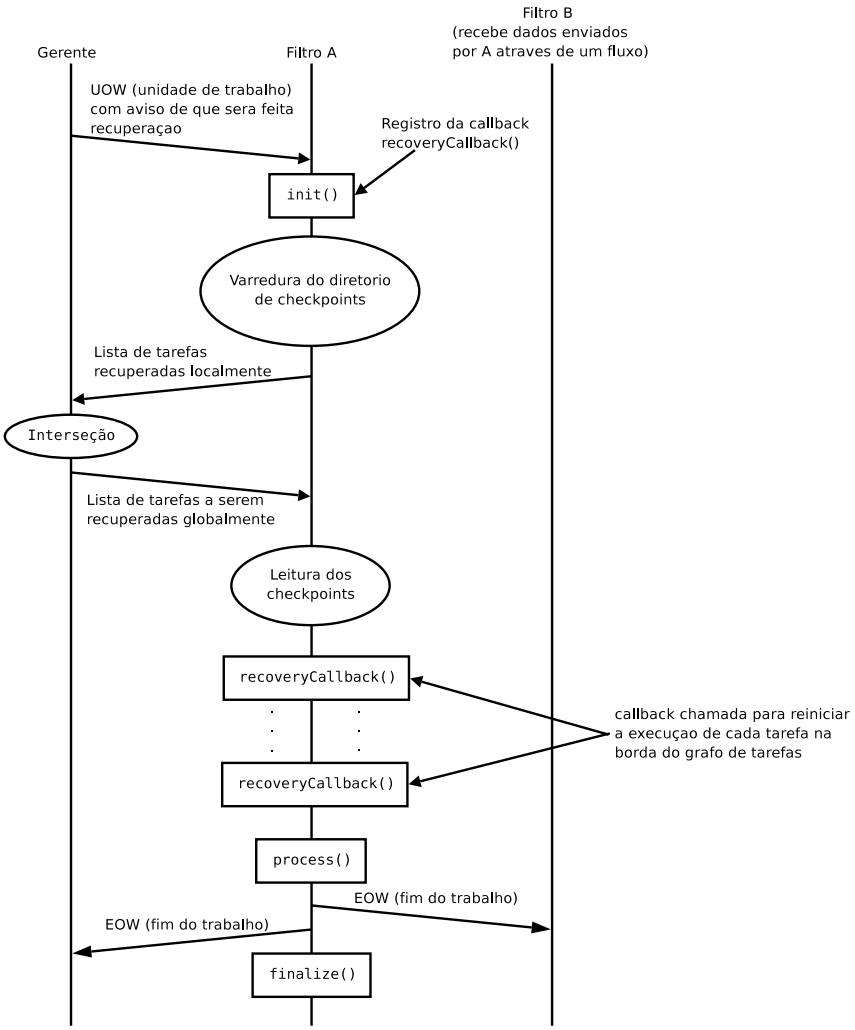


Figura 5.3: Mensagens trocadas no protocolo de recuperação

e criando novas. Neste trabalho não exploramos o escalonamento dos processos na recuperação. Se alguma máquina caiu, o gerente seleciona outras aleatoriamente para executar os processos que estavam nela enquanto os processos que estavam em máquinas que não caíram continuam a ser executados aonde estavam.

Após subir as novas instâncias, o gerente envia a unidade de trabalho que estava sendo executada quando o falta ocorreu, com um aviso de que ocorrerá uma recuperação. Assim, após a execução da função init(), onde o filtro registra a *callback* de recuperação, as instâncias examinam seus diretórios de *checkpoints* e enviam para o gerente as listas com os identificadores (*taskIds*) das tarefas que conseguirão recuperar (a lista de tarefas recuperadas localmente). O gerente faz a interseção de todas essas listas e envia de volta para as instâncias a lista das tarefas que elas devem recuperar, também chamada de lista de tarefas a serem recuperadas globalmente

(Figura 5.3). Assim que recebem essa lista, as instâncias recuperam as tarefas que aparecem nela.

Como os filtros trocam mensagens dentro de uma tarefa, todos os *checkpoints* de uma tarefa em todas as instâncias de filtro são interdependentes. Se houver uma falta e alguma instância não conseguir recuperar o estado de uma tarefa, a tarefa terá que ser reexecutada em todas instâncias novamente. Ainda assim, o Anthill não sofre de efeito dominó, pois sempre que uma tarefa terminar sua execução e for replicada em todas as instâncias, ela não terá mais que ser reexecutada, ao contrário de sistemas de tolerância a faltas que sofrem de efeito dominó nos quais, em qualquer momento da execução, corre-se o risco de voltar ao início.

Quando o sistema recupera os *checkpoints* das tarefas, ele coloca todas as filhas não recuperadas dessas tarefas (a borda do grafo de tarefas) em uma lista de tarefas a serem reexecutadas (Figura 5.2, linhas 11 a 15). A informação de quais tarefas são filhas de uma tarefa recuperada é obtida do arquivo de *checkpoint*. Para cada tarefa na lista de tarefas a serem reexecutadas é chamada a *callback* registrada pelo filtro para reexecutá-la e finalmente é chamada a função process() do filtro que volta à sua execução normal.

Capítulo 6

Experimentos

6.1 Configuração de testes

Nossos experimentos foram realizados em um *cluster* de Athlon64 3200+ (2010 MHz) com 2 GB de RAM, disco SATA (ATA 150) de 160 GB, rede Gigabit Ethernet e rodando Linux 2.6.8 .

A aplicação utilizada em nossos testes é uma adaptação do Eclat paralelo [VJF+04] (descrito na Seção 3.2.2), para o uso de tarefas. O Eclat varre uma base de transações e gera regras de associação mostrando qual a chance de alguma das transações que possui um determinado item, também possuir outro item. O algoritmo Eclat está descrito mais detalhadamente na seção 3.2.1.

Como já foi discutido na seção 3.2.3, o grafo de tarefas gerado por essa aplicação tende a ficar cada vez mais largo no início da execução e a partir de certo ponto começa a se estreitar até o término da execução.

Quando ocorre uma falta, o Anthill tem que chamar as *callbacks* para redisparar a execução das tarefas na borda do grafo de tarefas. Assim, o tempo de processamento das callbacks tende a ser proporcional ao tamanho da borda do grafo de tarefas.

6.2 Resultados

No primeiro teste medimos o tempo de execução do algoritmo sem injeção de faltas. Nesse teste, executamos o algoritmo com uma instância do filtro Adder (restrição do algoritmo) e suporte mínimo de $5,5\%$. Variamos o número de máquinas executando instâncias do filtro Merger (8, 12 e 16) e o tamanho da base (1,6; 2,4 e 3,2 milhões de transações respectivamente), assim cada instância de Merger sempre processava 800.000 transações. Esses testes foram realizados sem injeção de faltas. Os resultados foram os seguintes:

Número de processadores	8	12	16
Tempo de execução (s)	24,2	34,0	49,8

Como a base é proporcional ao número de processadores espera-se que todas as execuções durem o mesmo tempo, mas o algoritmo suporta apenas uma instância do filtro Adder, a qual se torna um ponto de contenção à medida que o número de processadores aumenta. Esse comportamento já foi observado em um trabalho anterior [VJF+04] que implementava uma paralelização do Eclat sobre o DataCutter[BKC+01, BCC+02, AKSS01, NCK+03, NKCS03, SFB+02].

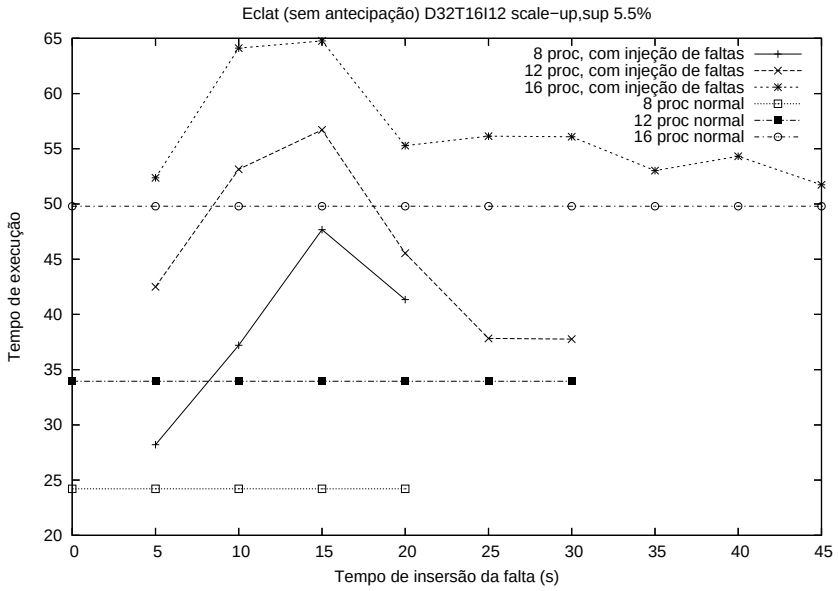


Figura 6.1: Tempo de execução de acordo com tempo de injeção da falta (os tempos de execução sem falta são indicados como retas paralelas ao eixo x)

Para medir o custo da recuperação, realizamos esses mesmos testes, porém com injeção de uma falta durante o teste e variando o tempo de injeção da mesma. Em nossos experimentos, a injeção de falta consiste em escolher aleatoriamente uma máquina utilizada no teste (exceto a máquina que executa o gerente) e interromper a execução de todas as instâncias de filtros executando nela. Os resultados podem ser vistos na Figura 6.1. Para comparação, os tempos de execução sem faltas aparecem como retas paralelas ao eixo x. As retas de 8 e 12 processadores terminam nos tempos 20 s e 30 s respectivamente, pois os tempos de execução sem faltas são 24,2 s e 34,0 s e se realizarmos experimentos com tempos superiores a esses a aplicação terminará normalmente antes da falta ser injetada.

A seguir mostramos o tempo total de recuperação medido durante o teste anterior e o decompomos em tempo de recuperação no gerente (tempo gasto pelo gerente para

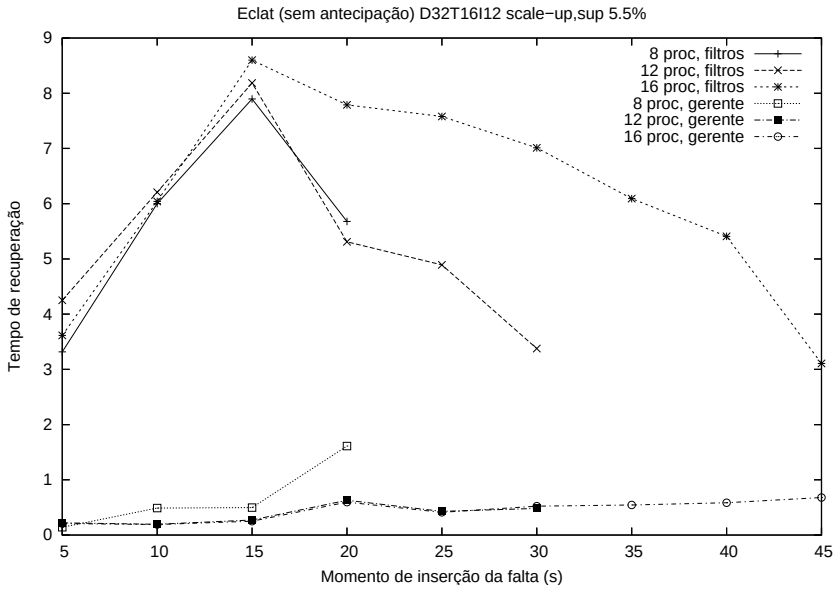


Figura 6.2: Tempo de recuperação decomposto em tempo no gerente e tempo nos filtros

montar a linha de recuperação) e tempo de recuperação nas instâncias dos filtros (Figura 6.2). Pode-se ver que o tempo de recuperação no gerente cresce à medida que aumentamos o tempo de execução antes da falta, pois isso aumenta o tamanho das listas que o gerente terá de obter a interseção (as listas de tarefas executadas). Já o tempo de recuperação nos filtros cresce até o tempo de faltas de 15 s e depois começa a diminuir, ao invés de continuar aumentando como seria esperado, uma vez que o estado sempre aumenta à medida que avança a execução. Para explicar isso, a Figura 6.3 apresenta a decomposição do tempo de recuperação nos filtros.

O tempo de recuperação nos filtros é composto pelas seguintes partes:

Leitura de *checkpoints*: o tempo gasto nessa atividade varia com o tamanho do estado armazenado nos *checkpoints*, que sempre aumenta à medida que a execução avança.

Execução de callbacks de recuperação: Quando o sistema de tolerância a faltas recupera uma tarefa porém não consegue recuperar alguma de suas filhas, ele chama uma *callback* definida pela aplicação para reexecução das tarefas filhas. No caso do Eclat, o custo de execução das *callbacks* das tarefas filhas é muito maior que a recuperação do estado da tarefa mãe, uma vez que a recuperação requer a varredura da lista invertida de um *itemset* (da tarefa mãe), e cada *callback* percorre 3 listas invertidas de *itemsets*: as listas dos dois *itemsets* que estão sendo combinados e a do novo *itemset*. Durante a recuperação

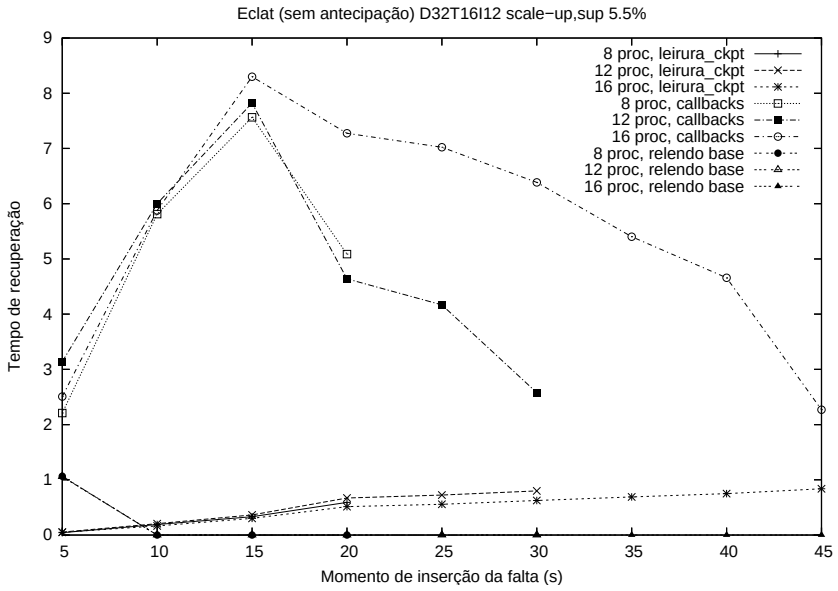


Figura 6.3: Decomposição do tempo de recuperação nos filtros

será executada uma *callback* para cada tarefa que esteja na borda do grafo de tarefas (Seção 3.1). Em todas execuções, a borda cresce até os 15 segundos de execução, diminuindo em seguida (Figura 6.5).

Releitura da base: quando a falta é inserida antes do algoritmo processar toda a base, ele é obrigado a relê-la completamente, mesmo que o sistema de tolerância a faltas consiga recuperar parte dela. O custo da re-leitura da base é a linha que está em 1s quando $y = 5$ e fica 0 nos outros casos. Ele ocorre somente quando inserimos a falta aos 5 segundos de execução, causando 1 s de tempo adicional de execução. Esse custo não varia quando aumentamos o número de processadores, pois aumentamos a base na mesma proporção, de forma que cada processador tenha que ler 800.000 transações.

A seguir mostramos o tempo de execução útil (Figura 6.4), que é o tempo total de execução menos o tempo de recuperação. O tempo de execução útil para 12 e 16 processadores apresenta elevações de cerca de 10 s quando as faltas são inseridas nos tempos de 10 e 15 s e seguem um patamar estável de cerca de 48 s com 16 processadores e 35 s com 12 processadores.

Para explicar o pico do tempo de execução quando injetamos a falta nos tempos de 10 e 15 s, mostraremos o tempo de execução e o número de *itemsets* processados (Figura 6.5). Pode-se ver que quando há picos no tempo de execução, são processados mais *itemsets*, ou seja, existem *itemsets* que não foram recuperadas e acabaram

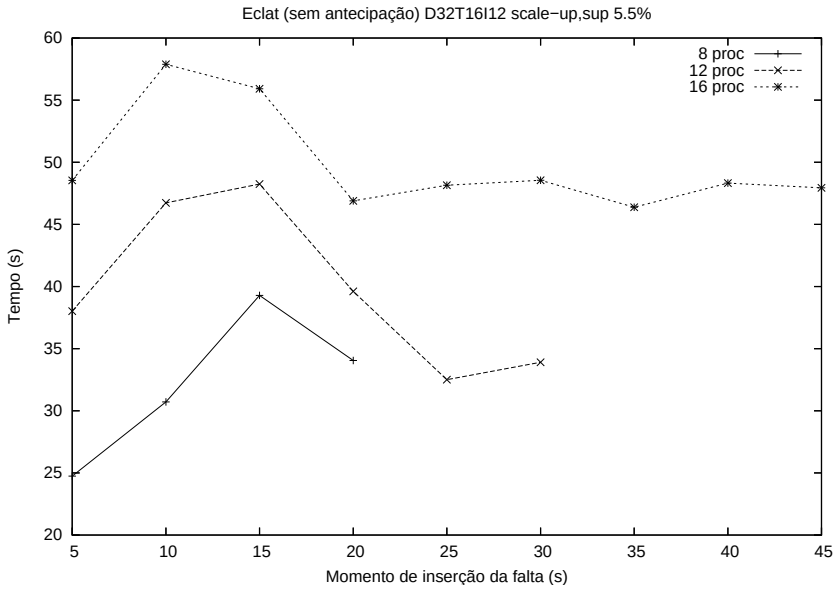


Figura 6.4: Tempo de execução útil (tempo total de execução menos tempo de recuperação)

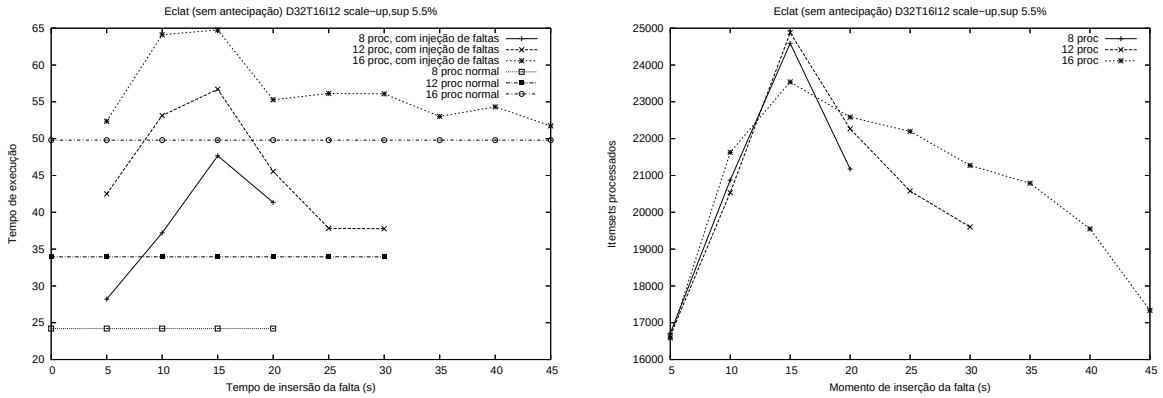


Figura 6.5: Tempo total de execução e *itemsets* processados

sendo reexecutados. As curvas de tempo de execução e *itemsets* processados não têm a mesma aparência pois o número de *itemsets* processados influencia no tempo de execução, mas não de forma direta, uma vez o tempo de processamento de um *itemset* pode variar muito.

6.3 Sumário

Os resultados mostraram que o custo de uma recuperação é particularmente alto para poucos processadores, mas diminui à medida que adicionamos mais processa-

dores. Mostramos também que grande parte desse tempo é devida a re-execução de trabalho perdido. Apesar de em alguns casos o custo ser relativamente alto, lembramos que estamos lidando com o pior caso possível: uma aplicação com tempo de execução muito curto, muito estado e pouca computação por MB de estado. Provavelmente no caso médio, o custo da recuperação será muito menor.

Capítulo 7

Conclusão e trabalhos futuros

Neste trabalho foi implementado um sistema de tolerância a faltas baseado no modelo de *checkpoint-rollback*. Esse sistema foi implementado totalmente em nível de usuário, sendo, assim, um sistema facilmente portável para qualquer sistema operacional. Como o sistema funciona a nível de aplicação, o programador da aplicação necessita utilizar a API de tarefas para que sua aplicação fique tolerante a faltas. Com relação ao desempenho, esse sistema mostrou ser escalável, na medida que o custo da recuperação não é muito alto, nem aumenta muito com o número de processadores.

7.1 Trabalhos Futuros

Esse sistema de tolerância da faltas, apesar de ter funcionado razoavelmente bem está longe de ser um sistema completo. Ainda há muita coisa a ser feita, principalmente:

- Implementar mecanismo de replicação que envie os dados para um processo em outra máquina no lugar de escrever no sistema de arquivos. Assim, se o modelo de faltas não assumir que as máquinas podem voltar após uma falta não será necessário gravar os *checkpoints* em um servidor NFS centralizado, que se torna um gargalo.

- Existe uma versão do Anthill que suporta o disparo de um appendWork() sem que o anterior tenha terminado, permitindo que o filtro inicie o processamento da próxima unidade de trabalho sem ter que esperar que todos os filtros terminem o anterior. O sistema de tolerância a faltas ainda não funciona nessa versão.

- Implementar um coletor de lixo que verifique que apague os dados das tarefas assim que não existam mais tarefas que necessitarão desses dados e que não haja risco de uma falta forçar a reexecução de alguma dessas tarefas.
- Implementar um mecanismo para apagar os arquivos deixados pelo sistema no diretório temporário de *checkpoints* quando ocorre uma falta.
- Investigar melhor como reescalonar os processos que estavam executando em uma máquina que caiu.

Além disso, o próprio Anthill precisa de um *buffer* de mensagens para agregar várias mensagens do filtro em uma mensagem do PVM. Isso permitiria um melhor aproveitamento da largura de banda da rede, principalmente em redes mais rápidas, que só conseguem utilizar toda sua banda quando são transmitidas mensagens grandes, como redes Gigabit Ethernet.

Referências Bibliográficas

[AAB+05] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Çetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag S. Maskey, Alexander Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, , and Stan Zdonik. The design of the borealis stream processing engine. In *2nd Biennial Conference on Innovative Data Systems Research (CIDR'05)*, pages 277–289, January 2005.

[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, EUA, 2001. IEEE Computer Society Press.

[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*, page 143, Washington, DC, EUA, 2002. IEEE Computer Society Press.

[AIS93] Rakesh Agrawal, Tomasz Imielinski, and Arun Swami. Mining association rules between sets of items in large databases. In *SIGMOD '93: Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, pages 207–216, New York, NY, USA, 1993. ACM Press.

[AKSS01] Henrique Andrade, Tahsin Kurc, Alan Sussman, and Joel Saltz. Efficient execution of multiple query workloads in data analysis applications. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pages 53–53, New York, NY, EUA, 2001. ACM Press.

[AS94] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *VLDB '94: Proceedings of the*

20th International Conference on Very Large Data Bases, pages 487–499, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[BBMS05] Magdalena Balazinska, Hari Balakrishnan, Samuel Madden, and Michael Stonebraker. Fault-tolerance in the Borealis distributed stream processing system. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 13–24, New York, NY, USA, 2005. ACM Press.

[BCC+02] Michael Beynon, Chialin Chang, Umit Catalyurek, Tahsin Kurc, Alan Sussman, Henrique Andrade, Renato Ferreira, and Joel Saltz. Processing large-scale multi-dimensional data in parallel and distributed environments. *Parallel Computing*, 28(5):827–859, 2002.

[BDV94] Greg Burns, Raja Daoud, and James Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.

[BG05] Romain Boichat and Rachid Guerraoui. Reliable and total order broadcast in the crash-recovery model. *Journal of Parallel and Distributed Computing*, 65(4):397–413, 2005.

[BGS02] R. Badrinath, Rakesh Gupta, and Nisheeth Shrivastava. Ftop: A library for fault tolerance in a cluster. In *Proceedings of the Fourteenth IASTED International Conference on Parallel and Distributed Computing Systems (PDCS 2002)*, pages 515–520, Cambridge, EUA, Novembro 2002. IASTED/ACTA Press.

[BKC+01] Michael D. Beynon, Tahsin Kurc, Umit Catalyurek, Chialin Chang, Alan Sussman, and Joel Saltz. Distributed processing of very large datasets with datacutter. *Parallel Computing*, 27(11):1457–1478, Novembro 2001.

[BMPS03] Greg Bronevetsky, Daniel Marques, Keshav Pingali, and Paul Stodghill. Automated application-level checkpointing of mpi programs. In *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 84–94, San Diego, California, EUA, 2003. ACM Press.

[BPHV00] Guillaume Brat, SeungJoon Park, Klaus Havelund, and Willem Visser. Java PathFinder - Second Generation of a Java Model Checker. In *Pro-*

ceedings of Post-CAV Workshop on Advances in Verification, Chicago, Julho 2000.

[Bre01] Eric A. Brewer. Lessons from giant-scale services. *IEEE Internet Computing*, Julho/Agosto 2001.

[BV] Guillaume Brat and Arnaud Venet. C Global Surveyor. http://ti.arc.nasa.gov/ase/cgs/.

[CASD85] F. Cristian, H. Aghili, R. Strong, and D. Dolev. Atomic broadcast: From simple message diffusion to byzantine agreement. In *Proceedings of the 15th International Conference on Fault-Tolerant Computing*, pages 200–206, Ann Arbor, Michigan, Junho 1985. IEEE Computer Society Press.

[CCF04] George Candea, James Cutler, and Armando Fox. Improving availability with recursive microreboots: A soft-state system case study. *Performance Evaluation Journal*, 56(1-3):213–248, Março 2004.

[CCG+04] Sagar Chaki, Edmund Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular verification of software components in C. *IEEE Transactions on Software Engineering*, 30(6):388–402, 2004.

[CF01] George Candea and Armando Fox. Recursive restartability: Turning the reboot sledgehammer into a scalpel. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, page 125. IEEE Computer Society Press, Maio 2001.

[CF03] George Candea and Armando Fox. Crash-only software. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS IX)*, Lihue, HI, Maio 2003.

[CKL04] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer-Verlag GmbH, 2004.

[CL85] K. Mani Chandy and Leslie Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 3(1):63–75, 1985.

[Cri91] Flavin Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, 1991.

[CW02] Hao Chen and David A. Wagner. Mops: an infrastructure for examining security properties of software. Technical report, University of California at Berkeley, Berkeley, CA, EUA, 2002.

[DG04] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the Sixth Symposium on Operating System Design and Implementation (OSDI'04)*, pages 137–150, Dezembro 2004.

[DHR02] J. Duell, P. Hargrove, and E. Roman. The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart. Berkeley Lab Technical Report LBNL-54941, Berkeley Lab, 2002.

[dRC94] Juan Miguel del Rosario and Alok N. Choudhary. High-performance I/O for massively parallel computers: problems and prospects. *Computer*, 27(3):59–68, 1994.

[EAWJ02] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys (CSUR)*, 34(3):375–408, 2002.

[EL02] David Evans and David Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, 2002.

[FJG+05] Renato Ferreira, Wagner Meira Jr., Dorgival Guedes, Lúcia Drummond, Bruno Coutinho, George Teodoro, Túlio Tavares, Renata Araújo, and Guilherme Ferreira. Anthill: A scalable run-time environment for data mining applications. In *Proceedings of the 17th Brazilian Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, Rio de Janeiro, Brazil, Outubro 2005. IEEE Computer Society Press.

[Fos02] Jeffrey Foster. *Type Qualifiers: Lightweight Specifications to Improve Software Quality*. PhD thesis, Dezembro 2002.

[GA03] Sudhakar Govindavajhala and Andrew W. Appel. Using Memory Errors to Attack a Virtual Machine. In *SP '03: Proceedings of the 2003 IEEE Symposium on Security and Privacy*, page 154, Washington, DC, EUA, 2003. IEEE Computer Society.

[GGHL+96] Al Geist, William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing L. Lusk, William Saphir, Tony Skjellum, and Marc Snir. MPI-2: extending the message-passing interface. In *Euro-Par '96: Proceedings of the Second International Euro-Par Conference on Parallel Processing*, pages 128–135, London, UK, 1996. Springer-Verlag.

[GHLL+98] William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir. *MPI – The Complete Reference: Volume 2, the MPI-2 Extensions.* MIT Press, 1998.

[GL96] William Gropp and Ewing Lusk. User's guide for mpich, a portable implementation of MPI. Technical Report ANL/MCS-TM-ANL-96/6, Argonne National Laboratory, Mathematics and Computer Science Division, Fevereiro 1996.

[GLDS96] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. High-performance, portable implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6):789–828, 1996.

[GLS94] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface.* MIT Press, 1994.

[Gri01] Steven D. Gribble. Robustness in complex systems. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, pages 21–26, Maio 2001.

[Gär99] Felix C. Gärtner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Computing Surveys*, 31(1):1–26, 1999.

[HDR04] John Hatcliff, Matthew B. Dwyer, and Robby. Bogor: An extensible framework for domain-specific model checking. *Newsletter of European Association of Software Science and Technology (EASST)*, 9:24–34, Dezembro 2004.

[Hew] Hewlett-Packard Development Company, L.P. HP Integrated Lights-Out Standard. http://h18013.www1.hp.com/products/servers/management/ilo/index.html.

[HJMS03] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Software Verification with Blast. In *Proceedings of the 10th*

SPIN Workshop on Model Checking Software (SPIN), Portland, Oregon, EUA, Maio 2003.

[HL02] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 291–301, New York, NY, EUA, 2002. ACM Press.

[Hol00] Gerard J. Holzmann. Logic Verification of ANSI-C Code with SPIN. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, pages 131–147, London, UK, 2000. Springer-Verlag GmbH.

[Hol02] Gerard J. Holzmann. Static source code checking for user-defined properties. In *Proceedings of the 6th World Conference on Integrated Design and Process Technology (IDPT2002)*, Pasadena, CA, Junho 2002. Society for Design and Process Science. Extended paper version available in http://spinroot.com/uno/uno_long.pdf.

[Inc04] Advanced Micro Devices Inc. AMD Sempron Product Data Sheet. Publication # 31805, United States, Agosto 2004.

[KA05] John Kodumal and Alex Aiken. Banshee: A Scalable Constraint-Based Analysis Toolkit. In *Proceedings of The 12th International Static Analysis Symposium (SAS '05)*, London, UK, Setembro 2005.

[LF03] Ben C. Ling and Armando Fox. A self-tuning, self-protecting, self-healing session state management layer. In *Proceedings of the 5th Annual Workshop On Active Middleware Services (AMS 2003)*, Seatle, WA, Junho 2003.

[Lin] Linux Networkx, Inc. Icebox Cluster Management Appliance. http://linuxnetworx.com/icebox/.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[MM88] A. Mahmood and E. J. McCluskey. Concurrent Error Detection Using Watchdog Processors-A Survey. *IEEE Transactions on Computers*, 37(2):160–174, Fevereiro 1988.

[MPI94] Forum MPI. Mpi: A message-passing interface. Technical report, 1994.

[MR91] A. Mankin and K. Ramakrishnan. Gateway congestion control survey. RFC 1254, IETF Performance and Congestion Control Working Group, United States, 1991.

[MS92] Shivakant Mishra and Richard D. Schlichting. Abstractions for constructing dependable distributed systems. Technical Report TR 92-19, CS Dept. University of Arizona, 1992.

[MSDS05] Hans Meuer, Erich Strohmaier, Jack Dongarra, and Horst D. Simon. Top 500 Supercomputer Sites. http://www.top500.org, 2005.

[NCK+03] Sivaramakrishnan Narayanan, Umit Catalyurek, Tahsin Kurc, Xi Zhang, and Joel Saltz. Applying database support for large scale data driven science in distributed environments. In *GRID '03: Proceedings of the Fourth International Workshop on Grid Computing*, pages 141–148, Washington, DC, EUA, 2003. IEEE Computer Society Press.

[New05] CNET News.com. Google's secret of success? Dealing with failure. http://news.com.com/Googles+secret+of+success+Dealing+with+failure/2100-1032_3-5596811.html, 2005.

[NKCS03] Sivaramakrishnan Narayanan, Tahsin Kurc, Umit Catalyurek, and Joel Saltz. Database support for data-driven scientific applications in the grid. *Parallel Processing Letters*, 13(2):245–271, 2003.

[NMW02] George C. Necula, Scott McPeak, and Westley Weimer. Ccured: type-safe retrofitting of legacy code. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 128–139, New York, NY, EUA, 2002. ACM Press.

[PBB+02] David A. Patterson, Aaron Brown, Pete Broadwell, George Candea, Mike Chen, James Cutler, Patricia Enriquez, Armando Fox, Emre Kiciman, Matthew Merzbacher, David Oppenheimer, Naveen Sastry, William Tetzlaff, Jonathan Traupman, and Noah Treuhaft. Recovery-oriented computing (ROC): Motivation, definition, techniques, and case studies. Computer Science Technical Report UCB//CSD-02-1175, UC Berkeley, Março 2002.

[PBS+88] D. Powell, G. Bonn, D. Seaton, P. Verissimo, and F. Waeselynck. The Delta-4 approach to dependability in open distributed computing systems. In *Proceedings of the 18th International Symposium on Fault Tolerant Computing Systems (FTCS 18)*, pages 246–251, Tokyo, Junho 1988.

[Pri95] Dick Price. Pentium FDIV flaw-lessons learned. *IEEE Micro*, 15(2):88–87, 1995.

[Ran75] Bryan Randell. System structure for software fault tolerance. *ACM SIGPLAN Notices*, 10(6):437–449, 1975.

[Sch90] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.

[SFB+02] Matthew Spencer, Renato Ferreira, Michael Beynon, Tahsin Kurc, Umit Catalyurek, Alan Sussman, and Joel Saltz. Executing multiple pipelined data analysis operations in the grid. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–18, Los Alamitos, CA, EUA, 2002. IEEE Computer Society Press.

[SOHL+96] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI: The Complete Reference.* MIT Press, 1996.

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

[SSB+03] Sriram Sankaran, Jeffrey M. Squyres, Brian Barrett, Andrew Lumsdaine, Jason Duell, Paul Hargrove, and Eric Roman. The LAM/MPI checkpoint/restart framework: System-initiated checkpointing. In *Proceedings, LACSI Symposium*, Sante Fe, New Mexico, EUA, Outubro 2003.

[ST90] D. Shasha and J. Turek. Beyond Fail-Stop: Wait-Free Serializability and Resiliency in the Presence of Slow-Down Failures. Technical Report TR1990-514, New York University, New York, NY, EUA, Setembro 1990.

[Tea98] The LAM Team. Getting started with LAM/MPI. www.lam-mpi.org/tutorials/lam/, 1998.

[Tre05] Michael Treaster. A survey of fault-tolerance and fault-recovery techniques in parallel systems. Technical report, National Center for Supercomputing Applications (NCSA), Janeiro 2005.

[Tur00] Dave Turner. Scientific applications on workstation clusters vs supercomputers. In *APS March 2000 Meeting*, Minneapolis, MN, Março 2000.

[VdL89] Paulo Veríssimo and Rogério de Lemos. Confiança no Funcionamento: Proposta para uma Terminologia em Português. Publicação conjunta, INESC e LCMI/UFSC, 1989.

[VJF+04] Adriano Veloso, Wagner Meira Jr., Renato Ferreira, Dorgival Guedes Neto, and Srinivasan Parthasarathy. Asynchronous and anticipatory filter-stream based parallel algorithm for frequent itemset mining. In *Knowledge Discovery in Databases: PKDD 2004, 8th European Conference on Principles and Practice of Knowledge Discovery in Databases, Proceedings*, volume 3202 of *Lecture Notes in Computer Science*, pages 422–433, Pisa, Italy, Setembro 2004. Springer-Verlag GmbH.

[WGT99] Ewing Lusk William Gropp and Rajeev Thakur. *Using MPI-2: Advanced Features of the Message Passing Interface.* MIT Press, 1999.

[Wik] Wikipedia, the free encyclopedia. ENIAC. http://www.wikipedia.org/wiki/Eniac.

[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)*, pages 273–288, San Francisco, CA, Dezembro 2004.

[ZPOL97] Mohammed J Zaki, Srinivasan Parthasarathy, Mitsunori Ogihara, and Wei Li. New algorithms for fast discovery of association rules. Technical report, Rochester, NY, USA, 1997.