SUPORTE PARA ADIÇÃO DINÂMICA DE INSTÂNCIAS EM TEMPO DE EXECUÇÃO NO ANTHILL

DANIEL LACET DE FARIA FIREMAN
Orientador: Renato Celso Ferreira

SUPORTE PARA ADIÇÃO DINÂMICA DE INSTÂNCIAS EM TEMPO DE EXECUÇÃO NO ANTHILL

Dissertação apresentada ao Programa 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.

Belo Horizonte
Maio de 2009

Fireman, Daniel Lacet de Faria
D523s Suporte para Adição Dinâmica de Instâncias em tempo de execução no Anthill / Daniel Lacet de Faria Fireman. — Belo Horizonte, 2009
xiv, 76 f. : il. ; 29cm

Dissertação (mestrado) — Universidade Federal de Minas Gerais
Orientador: Renato Celso Ferreira

1. Sistemas Distribuídos. 2. Filtro-Fluxo. 3. Workflow. 4. Anthill. I. Título.

CDU 519.6*224

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Suporte para a Adição de Instâncias em Tempo de Execução no Anthill

DANIEL LACET DE FARIA FIREMAN

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

PROF. RENATO ANTÔNIO CELSO FERREIRA - Orientador
Departamento de Ciência da Computação - UFMG

DRA. ANOLAN YAMILÉ MILANÉS BARRIENTOS
Bolsista de Pós-Doutorado - DCC - ICEx - UFMG

PROF. DORGIVAL OLAVO GUEDES NETO
Departamento de Ciência da Computação - UFMG

PROF. WAGNER MEIRA JUNIOR
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 24 de abril de 2009.

Resumo

De um lado, a contínua evolução tecnológica nas diversas áreas do conhecimento vem fazendo com que conjuntos cada vez maiores de dados se tornem disponíveis, atingindo a ordem de petabytes. Nesse cenário, faz-se imprescindível a utilização de recursos distribuídos para realizar o processamento de dados em tempo hábil. Do outro lado, com a popularização das plataformas de execução compostas por estações de trabalho, os sistemas de computação vêm se tornando inerentemente dinâmicos: recursos podem ser adicionados ou estão sujeitos à falhas. Entretanto, a utilização desses recursos como plataforma de computação de alto desempenho ainda é restrita, uma vez que o desenvolvimento de sistemas paralelos eficientes e escaláveis se mantém uma tarefa difícil, até mesmo para programadores experientes.

O Anthill é um ambiente de programação paralela baseado no paradigma filtro fluxo. Esse paradigma permite o processamento eficiente de grandes volumes de dados, uma vez que o paralelismo pode ser explorado de maneira simples e intuitiva. Entretanto, o Anthill foi construído como um ambiente de execução estático e, como tal, não permite que a distribuição dos componentes da aplicação seja modificada em tempo de execução. Neste trabalho apresentamos um conjunto de extensões que adicionam suporte eficiente ao aumento dinâmico da plataforma execução de uma aplicação Anthill. Nossa solução explora a localidade de referência e a assincronia que muitas aplicações Antill podem fazer uso. Os resultados de nossa avaliação experimental mostram que esta implementação viabiliza a utilização de mais poder computacional em tempo de execução, mantendo, com baixo custo, a consistência e a continuidade da exploração assíncrona do paralelismo em diversos níveis.

Abstract

On one hand, the continuous evolution of technology in various areas of knowledge is leading to an increase of size of available data sets, reaching the order of petabytes. In this scenario, it is essential to use distributed resources to process data in a timely fashion. On the other hand, with the popularization of platforms composed by workstations, distributed computing systems have become inherently dynamic: resources may be added or are subjected to faults. However, the use of these resources as a platform for high performance computing is still restricted, since the development of efficient and scalable parallel systems remains a difficult task, even for experienced programmers.

Anthill is a parallel programming environment based on the filter-stream paradigm. This paradigm allows to efficiently process large volumes of data, since it can exploit the parallelism in a simple and intuitive manner. However, Anthill has been built as a static environment and as such, it does not allow to modify the application's component distribution at runtime. This work presents a set of extensions that add support for augmenting the runtime platform of an Anthill application. Our solution exploits the locality of reference and the asynchrony which many Anthill application could make use. The results of our experimental evaluation show that this implementation allows the use of more computational power at runtime, keeping with low cost, the execution consistency and the asynchronous exploitation of different levels of parallelism.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

De um lado, a contínua evolução da tecnologia nas diversas áreas do conhecimento vem fazendo com que conjuntos cada vez maiores de dados estejam disponíveis, atingindo a ordem de petabytes, como exposto por Deatrich et al. [2008]. Neste cenário, faz-se imprescindível a utilização de um grande volume de recursos computacionais para realizar o processamento dos dados em tempo hábil.

Por outro lado, ocorre uma significativa evolução tecnológica dos microprocessadores e das redes de computadores. Fato observado com a popularização dos *many/multi-cores*, que atualmente podem realizar um grande volume de processamento paralelo e utilizar uma banda passante de dezenas de gigabits por segundo para comunicação. Neste cenário, os sistemas de computação distribuída compostos de vários microcomputadores conectados em rede - *clusters* - vem se tornando uma alternativa cada vez mais usada [Buyya, 1999; Reed et al., 2003; Kesselman e Foster, 1998]. A combinação dessas tendências está transformando a realização de computação em um processo inerentemente paralelo, podendo ser explorado em diversos níveis: entre os nodos do sistema distribuído ou dentre os núcleos de cada máquina.

Uma vez que grandes corporações e universidades geralmente possuem centenas ou milhares de computadores em seu parque, compor um *cluster* de estações de trabalho é uma maneira simples de prover grande poder computacional com baixo custo. Assim, aproveitar de forma eficaz o grande custo-benefício de sistemas de computação de alto desempenho montados usando hardware de prateleira é fundamental para que universidades, laboratórios e corporações continuem evoluindo na solução de problemas. Entretanto, a programação de sistemas paralelos eficientes e escaláveis para ambientes deste tipo se mantém uma tarefa extremamente difícil.

Para abordar esses problemas foi desenvolvido o Anthill [Ferreira et al., 2005]. Um ambiente de execução paralelo baseado no modelo filtro-fluxo [Beynon et al., 2001], o qual mostrou-se eficiente para o processamento de grandes volumes de dados, uma vez

que permite explorar paralelismo massivo de maneira simples e intuitiva. O modelo de programação é baseado no paradigma de fluxo de dados, onde as aplicações são criadas por meio de um processo de decomposição do processamento em unidades chamadas de filtros, que se comunicam através de fluxos de dados unidirecionais, formando um grafo direcionado. Em tempo de execução, é possível instanciar múltiplas cópias de cada um dos filtros que compõem a aplicação, replicando então cada um dos estágios da execução. Também possui um mecanismo para particionamento consistente do estado entre essas réplicas, chamado de fluxo-rotulado.

Para muitas aplicações, a decomposição natural em filtros leva à um grafo cíclico direcionado, onde a execução consiste em múltiplas iterações sobre os filtros. Em essência, iterações com dependência de dados são automaticamente identificadas e executadas concorrentemente, caso existam recursos para isto. Dessa forma é extraído o máximo de paralelismo das aplicações, uma vez que as unidades de processamento são na verdade cópias de estágios de um *pipeline*, pode-se ter um paralelismo de grão mais fino. Como todo esse processamento pode ocorrer assincronamente, a execução pode não ter gargalos devido ao sistema. Um dos benefícios da combinação destas três dimensões é o alcance de elevados *speedups* em diversas aplicações, como foi mostrado por Ferreira et al. [2005], e a manutenção desses *speedups* para um grande número de nodos de computação.

1.1 Motivação

O Anthill assume a utilização de recursos dedicados. O escalonamento das aplicações atualmente é feito a priori e de forma estática. Porém, no contexto de *clusters* compostos por estações de trabalho, os recursos computacionais freqüentemente são compartilhados e possuem disponibilidade variável. Desta forma, a adaptação às mudanças de disponibilidade e carga da plataforma de execução torna-se um ponto crítico para a utilização eficaz dos recursos.

Em linhas gerais, reconfiguração dinâmica implica na capacidade de modificar o mapeamento entre componentes da aplicação paralela e os recursos onde os mesmos estão sendo executados, mantendo o prosseguimento consistente da execução. Isto implica na capacidade de, durante a execução, utilizar mais recursos que venham a se tornar disponíveis ou suportar de forma consistente a remoção dos mesmos.

Por fim, seria bastante custoso para os desenvolvedores lidar com todos os detalhes envolvidos no processo de reconfiguração dinâmica de aplicações paralelas, dado a complexidade das aplicações e a escala e dinamismo que atingem os *clusters* atualmente. Assim, para atingir alta eficiência e escala nestes ambientes se faz necessário a criação

de ambientes de execução que suportem de forma eficiente reconfigurações dinâmicas.

1.2 Objetivos

Objetivo principal deste trabalho é dar um primeiro passo no sentido de tornar o Anthill um ambiente completamente reconfigurável. Definimos esse passo como o crescimento dinâmico plataforma, ou seja, a adição de instâncias de filtro em tempo de execução.

Visando completude, pretende-se modificar o ambiente de forma a permitir que reconfigurações ocorram em qualquer ponto da execução e independente de qual filtro, fluxo ou combinação destes esteja executando. Desta forma, a inclusão da funcionalidade de suporte à adição dinâmica de instâncias implica na possibilidade de reconfiguração de aplicações com estado. O que, por sua vez, pode levar à uma possível migração de dados entre instâncias de um mesmo filtro, o que não é suportado pelo Anthill em sua versão atual.

Por fim, outro objetivo específico é a manutenção da eficiência, dessa forma, não devem ser adicionadas barreiras ou qualquer outro de tipo de sincronia na implementação da funcionalidade.

1.3 Contribuições

Visando respaldar os esforços de implementação e experimentação, a primeira contribuição deste trabalho é a proposta de um modelo analítico que inclui as entidades componentes do ambiente de execução Anthill: aplicação, filtros, fluxos, plataforma de execução, instâncias de filtro, políticas de roteamento e escalonamento. A partir dos requisitos levantados, a adição dinâmica de instâncias de filtro foi primeiramente proposta como uma extensão desse modelo. Após análise da proposta analítica, implementou-se funcionalidade de adição não-bloqueante de instâncias de filtros.

Com a implementação da adição dinâmica de instâncias de filtro, lidar de maneira eficiente com a migração dinâmica de dados passou a ser de fundamental importância para execução de aplicações com estado. Assim, foi criado o Estado Global Distribuído, módulo do Anthill que tem como principal objetivo abstrair do programador a reconfiguração do estado, mantendo de forma eficiente a simplicidade das abstrações oferecidas pelo paradigma filtro-fluxo. Com a utilização deste, o estado de cada filtro é exportado como uma memória compartilhada, podendo ser acessada por qualquer instância como um vetor local, sendo os detalhes a cerca da migração dos dados abstraídos pelo ambiente de execução.

Trabalhos anteriores verificaram que o mecanismo de particionamento consistente do estado criado no Anthill não é eficiente quando utilizado em ambientes dinâmicos. Assim, uma nova versão do mecanismo de fluxo-rotulado foi implementada de acordo com o conceito de função de assinalamento consistente, proposto por Karger et al. [1997]. Foi mostrada a eficiência da implementação no rearranjo das partições do estado, ou seja, é movida a menor quantidade de dados possível para rebalancear a carga dentre as instâncias do filtro. Além disso, o protocolo tem bom desempenho para o cálculo assinalamentos de chaves, o que é de fundamental importância no caso de aplicações Anthill, uma vez que essa função é chamada para cada mensagem enviada através do fluxo.

1.4 Organização do Texto

O restante deste trabalho foi dividido em seis capítulos, organizados da seguinte forma:

Capítulo 2 [Anthill] Apresenta uma descrição detalhada dos componentes do ambiente de execução Anthill.

Capítulo 3 [Adição Dinâmica de Recursos] Detalha o protocolo de suporte à adição dinâmica de instâncias de filtro no Anthill, bem como sua implementação.

Capítulo 4 [Estado Global Distribuído] Detalha a implementação do Estado Global Distribuído adicionado ao Anthill.

Capítulo 5 [Particionamento Consistente do Estado de Aplicações Anthill] Baseado no modelo analítico exposto no Capítulo 2 e as extensões apresentadas no Capítulo 3, esse capítulo apresenta uma solução para particionamento consistente do estado em aplicações Anthill.

Capítulo 6 [Avaliação Experimental] É apresentado o resultado de um conjunto de experimentos, avaliando a completude e desempenho das soluções implementadas neste trabalho.

Capítulo 7 [Conclusões e Trabalhos Futuros] Apresenta uma série de conclusões sobre o trabalho, bem como possíveis caminhos para continuidade do mesmo.

Capítulo 2

Anthill

O Anthill, proposto por Ferreira et al. [2005], é um ambiente de execução inspirado no modelo filtro-fluxo [Beynon et al., 2001, 2002; Narayanan et al., 2003; Spencer et al., 2002]. Esse modelo é baseado no paradigma de fluxo de dados, onde as aplicações são criadas por meio de um processo de decomposição do processamento em unidades chamadas de filtros, que se comunicam através de fluxos de dados unidirecionais, compondo um *pipeline*.

O processo de decomposição da aplicação propicia um paralelismo de tarefas, uma vez que essa se torna um grafo direcionado e os filtros são executados como um *pipeline*. Além disso, em tempo de execução é possível instanciar múltiplas cópias de cada um dos filtros que compõem a aplicação. Esse mecanismo permite que qualquer vértice do grafo que descreve a aplicação seja replicado em várias máquinas da plataforma. Dessa forma, os dados que vão para filtros que possuem múltiplas instâncias são particionados, atingindo paralelismo de dados.

O desenvolvimento de aplicações paralelas no paradigma filtro-fluxo é naturalmente facilitado por uma divisão de responsabilidades, escondendo do desenvolvedor da aplicação detalhes a cerca do ambiente e da execução. Num primeiro momento, a arquitetura do sistema (*pipeline*) é definida levando-se em conta apenas detalhes envolvidos no comportamento da aplicação. Essa etapa é chamada de *layout* e é ilustrada na Figura 2.1. Num segundo momento o foco se volta para execução da aplicação, onde detalhes como o número de instâncias de cada filtro e o local onde as mesmas são executadas são considerados, visando otimizar o desempenho. Essa etapa é chamada de *placement* e é ilustrada na Figura 2.2.

Para muitas aplicações, a decomposição natural em filtros leva a um grafo cíclico direcionado, onde a execução consiste em múltiplas iterações sobre os filtros. Nossa experiência no desenvolvimento de aplicações segundo o modelo filtro-fluxo mostra que esse comportamento leva frequentemente a execuções assíncronas, onde várias soluções

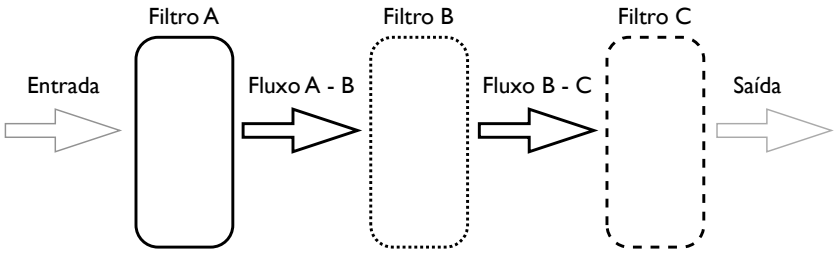


Figura 2.1. *Layout* de uma aplicação filtro-fluxo.

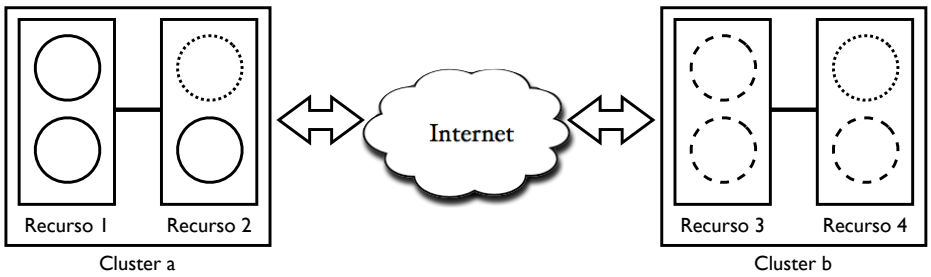


Figura 2.2. *Placement* de uma aplicação filtro-fluxo.

candidatas, que podem ter sido criadas em diferentes estágios do processamento, podem ser avaliadas simultaneamente. Uma ilustração desse tipo de aplicações Anthill é mostrada na Figura 2.3.

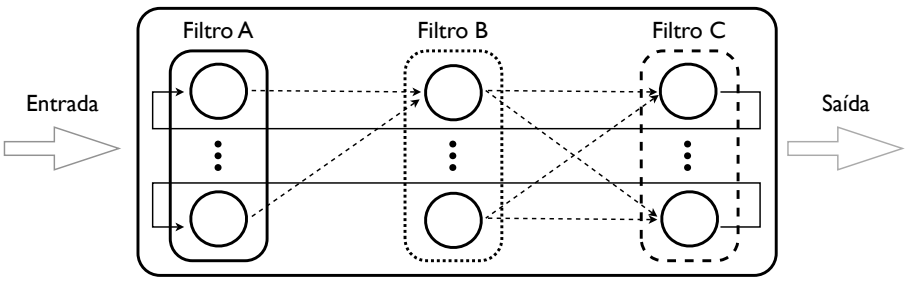


Figura 2.3. Aplicação Anthill cíclica.

O ambiente de execução permite a utilização de paralelismo de dados e tarefas. Além disso, viabiliza a utilização de uma terceira dimensão, a qual permite explorar a assincronia existente entre tarefas independentes ao longo do tempo. Uma vez que as unidades de processamento podem ser várias cópias de estágios de um *pipeline*, os dados a serem processados podem ser particionados, atingindo-se um paralelismo de grão mais fino. Por fim, como esse processamento pode ocorrer de forma assíncrona, gargalos podem ser transparentemente eliminados. Um dos benefícios da combinação dessas três dimensões é o alcance de elevados *speedups* em diversas aplicações, como foi mostrado por Ferreira et al. [2005].

2.1 O Ambiente de Execução

O ambiente de execução Anthill foi implementado como uma biblioteca, utilizada pelos desenvolvedores para construir aplicações, e um *daemon* que é executado em cada um dos nodos de computação disponíveis. Existem atualmente duas implementações do Anthill, as quais exportam a mesma interface de programação: uma versão baseada em PVM [Sunderam, 1990] e outra construída sobre MPI [Snir e Otto, 1998]. Uma aplicação Anthill precisa de três componentes para ser executada: o aplicativo console, um arquivo de configuração da aplicação e a biblioteca de filtros.

O console da aplicação é um programa executável que invoca uma função da biblioteca Anthill para iniciar o ambiente de execução. Essa função recebe o arquivo de configuração como argumento e esse processo passa a atuar como coordenador da execução da aplicação paralela, disparando a execução das cópias de filtros nos recursos, tal como definido pela configuração. Em seguida, são criadas e enviadas estruturas de dados que contêm os parâmetros para execução dos filtros (também chamados de *works*). Após o encerramento da aplicação, o console dispara uma função de finalização de todo o ambiente. A Figura 2.4 ilustra o código fonte de uma aplicação console simples, que utiliza o número de argumentos da função principal como parâmetro de execução dos filtros.

```
#include "anthill.h"
int main(int argc, char* argv[]){
   int myParam = argc;
   Layout* systemLayout = initAH("./conf.xml", argc, argv);
   ...
   appendWork(systemLayout, (void*) &myParam, sizeof(int));
   ...
   finalizeAH(systemLayout);
   return 0;
}
```

Figura 2.4. Uma aplicação console exemplo.

O arquivo de configuração é um descritor da aplicação escrito em formato XML. Um exemplo desse arquivo é mostrado na Figura 2.5, onde são ilustradas três seções:

1. *hostdec*: Especifica os nodos disponíveis para computação. Associado a cada nodo pode haver uma série de recursos, os quais são declarados pelo programador;

2. *placement*: Contém informações a respeito dos filtros, indicando a biblioteca de ligação dinâmica a ser carregada, quantas instâncias serão iniciadas e onde iniciá-las. A definição do local de execução de cada instância pode ser feito de forma implícita ou explícita: no primeiro caso nenhuma informação é fornecida e

o ambiente realiza a associação de forma aleatória; no último caso, a associação é realizada através do casamento (*matching*) entre a demanda da instância e os recursos declarados na seção anterior;

3. *layout*: Descreve as ligações entre filtros, as quais são definidas como fluxos de dados unidirecionais. Cada fluxo possui um nome que é utilizado como identificador único pelos filtros que se comunicam através de mais de um fluxo. Por fim, os campos *policy* e *policylib* são utilizados para definir a política de roteamento de mensagens, utilizada como base para a escolha de quais instâncias serão destinatárias das mensagens que trafegarão através de um determinado fluxo. As políticas de roteamento são explicadas em mais detalhes na Seção 2.3

```
<?xml version="1.0"?>
<config>
   <hostdec>
        <host name="speed01"><resource name="mysql"/></host>
             <host name="speed02"/>
   </hostdec>
   <placement>
        <filter name="filterA" libname="filterA.so" instances="1">
                <instance demands="mysql"/>
        </filter>
        <filter name="filterB" libname="filterB.so" instances="2">
   </placement>
   <layout>
        <stream>
                <from filter="filterA" port="toB"/>
                <to filter="filterB" port="fromA" policy="broadcast"/>
        </stream>
        <stream>
                <from filter="filterB" port="toA"/>
                <to filter="filterA" port="fromB" policy="ls" policylib="chash_label_func.so"/>
        </stream>
   </layout>
</config>
```

Figura 2.5. Exemplo de arquivo de configuração Anthill.

A Figura 2.6 ilustra uma possível disposição resultante do processamento do arquivo de configuração descrito na Figura 2.5. Uma vez que o processo console está presente em todas as aplicações, ele será omitido das demais ilustrações deste trabalho.

O último componente da aplicação Anthill é a biblioteca de ligação dinâmica (DLL) que contém o código a ser executado pelas instâncias. É esperado que, para cada filtro, o desenvolvedor forneça uma DLL contendo a implementação das funções initFilter, processFilter e finalizeFilter. No exemplo das Figuras 2.4 e 2.5, as bibliotecas são filterA.so e filterB.so.

Durante a inicialização da aplicação, o console envia mensagens aos *daemons* requisitando o carregamento das bibliotecas de acordo com o especificado na seção *placement*.

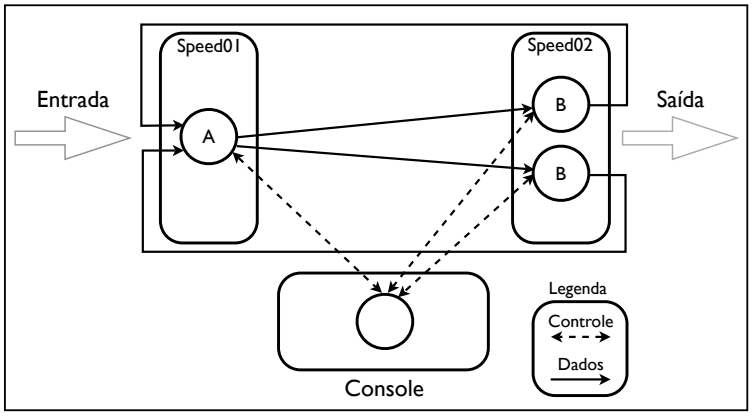


Figura 2.6. Exemplo de disposição de uma aplicação Anthill em tempo de execução.

Em seguida, cada uma das instâncias executa a função initFilter. Após o término da inicialização, a função processFilter é invocada para cada mensagem recebida pelas instâncias. Ao final da execução, a função finalizeFilter é executada. Todas as chamadas recebem como parâmetro todos os *works* adicionados no programa console.

2.2 Filtros

Visando auxiliar os desenvolvedores na tarefa de implementação dos filtros, a biblioteca Anthill oferece um conjunto de funções que realizam operações sobre os fluxos. Como mencionado, os fluxos possuem identificadores únicos definidos no arquivo de configuração e esses identificadores são utilizados, por exemplo, para leitura e escrita de dados nos fluxos.

Apesar do processo de inicialização da aplicação ser coordenado pelo ambiente de execução, cabe ao desenvolvedor determinar seu encerramento. Para aplicações acíclicas, esse processo ocorre através do encerramento dos fluxos de saída e as instâncias são consideradas encerradas quando todos os fluxos de saída forem finalizados. No caso de aplicações cíclicas foi desenvolvido um mecanismo de detecção de terminação baseado no protocolo de consenso distribuído [Ferreira et al., 2005]. Os filtros (estágio do *pipeline*) tem sua execução encerrada quando todas as suas instâncias são consideradas terminadas. Por fim, uma vez que que todos os filtros tenham sua execução terminada, o console executa o procedimento de término da aplicação.

Além dessas, estão disponíveis funções para obter o número de cópias de um determinado filtro e o identificador único de uma determinada cópia. Essas funções são comumente utilizadas para permitir que instâncias localizem suas posições dentro do estado e determinem as posições que devem processar.

Por fim, o Anthill provê um conjunto de funções de acesso a informações de monitoração. É possível obter, em tempo de execução, informações sobre utilização da memória e carga da máquina que uma determinada instância está executando. Munido dessas informações os desenvolvedores podem identificar gargalos e outros problemas.

2.3 Fluxos

Os fluxos representam a abstração de comunicação entre filtros. Na definição do *layout* de uma aplicação, os fluxos são um canal entre dois filtros. Já em tempo de execução, uma vez que várias instâncias de cada filtro podem estar executando em diversos componentes da plataforma, os fluxos escondem os detalhes a cerca do roteamento de mensagens.

Uma vez que as instâncias (ou cópias de filtro) são iniciadas pelo Anthill em tempo de execução e de forma transparente, foi proposta a utilização de políticas de roteamento de mensagens através dos fluxos. Essas definem um protocolo de escolha dos destinatários das mensagens que trafegam através dos fluxos, deixando a cargo do ambiente realizar a entrega correta das mensagens, escondendo assim os detalhes de implementação inerentes.

A versão atual do Anthill possui três políticas de roteamento: i) um caso básico, onde novas mensagens enviadas através de um fluxo são entregues à próxima instância destino seguindo um algoritmo tipo *round-robin*. Essa política utiliza o identificador numérico único de cada instância para estabelecer a ordenação necessária e definir qual será a próxima instância a receber mensagens; ii) a política de roteamento *broadcast*, onde todas as instâncias do filtro destino recebem uma cópia da mensagem e iii) a política fluxo rotulado (*labeled stream*), que expõe uma nova gama de possibilidades e pode ser vista como um *multicast* seletivo. A próxima seção é dedicada a prover maiores detalhes sobre o funcionamento do fluxo rotulado.

2.4 Fluxo Rotulado

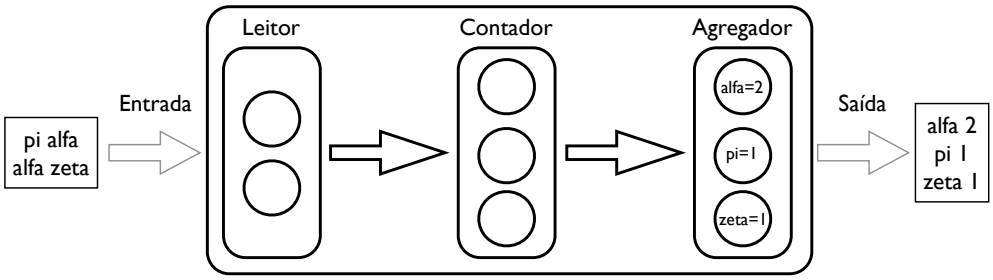
A possibilidade de dividir o programa em tarefas bem definidas e fazer uso quase transparente de diversos níveis de paralelismo torna o modelo filtro-fluxo atrativo para o programador. Um problema ocorre quando existe a necessidade de manter dados derivados do processamento de mensagens, o qual chamamos de estado do filtro. Como o modelo não prevê a comunicação entre instâncias de um mesmo filtro e não há um controle preciso, determinado pelo usuário, sobre o destino dos dados, nas primeiras implementações do modelo não existia uma maneira simples de se implementar filtros

com estado. Assim, foi introduzido pelo Anthill um mecanismo de manutenção consistente do espaço de estados que permite a implementação e execução eficiente de aplicações que possuem estados.

Para tal, foi criada uma nova política de roteamento, chamada de fluxo rotulado (*labeled stream*) [Veloso et al., 2004; Ferreira et al., 2005]. Em um fluxo rotulado, toda mensagem enviada é analisada por uma função que identifica uma informação especial: o rótulo (*label*). O formato do rótulo é dependente de cada aplicação, assim é definido pelo programador. Após a identificação do rótulo da mensagem, outra função é aplicada sobre o rótulo. O resultado dessa função irá determinar a instância destino da mensagem. Essa última função é consistente entre instâncias de um mesmo filtro, significando que, dado um determinado rótulo, independente de qual instância esteja executando, a saída será sempre a mesma. A utilização dessas duas funções oferece ao usuário um controle mais fino sobre o destino dos dados, solucionando o problema de particionamento consistente do estado. Por fim, essas funções não podem ser alteradas em tempo de execução.

Consideremos, como exemplo, a decomposição em filtros da aplicação Conta Palavras ilustrada na Figura 2.7. Ela é divida em três estágios: Leitor, Contador e Agregador. O filtro Leitor particiona o texto em linhas e envia uma linha por vez para o filtro Contador. Por sua vez, o filtro Contador, conta o aparecimento de cada palavra em cada linha e envia esses resultados de contagem para o filtro Agregador. Esse último recebe a quantidade de vezes que uma palavra qualquer apareceu numa determinada linha e soma com a quantidade de vezes que a mesma palavra apareceu na próxima linha e assim sucessivamente para todas as linhas e todas as palavras. Quando aplicação termina, o resultado da contagem é escrito num arquivo. Essa informação, que precisa permanecer no filtro Agregador entre o processamento das mensagens é o estado do filtro.

Figura 2.7. Decomposição em filtros da aplicação conta palavras.



Uma função de extração de rótulo simples e eficiente para esse problema pode extrair o código correspondente à primeira letra de cada palavra e retorná-lo. A função *hash* de módulo (*mod*) é aplicada nesse rótulo e é utilizado o número de instâncias como

operador do módulo. Com isso, temos que, por exemplo, a primeira instância, com identificação 0, será sempre o destino da contagem de todas palavras que começarem por "a" e assim sucessivamente para todas as palavras componentes do texto.

A política de fluxo rotulado tem sido amplamente utilizada em aplicações Anthill permitindo que o estado original do filtro seja particionado entre suas instâncias. Em essência, a política garante a localidade de referência nos acessos ao estado efetuado por cada uma das instâncias de filtro. Essa localidade é assegurada pela função *hash* pois, uma vez realizado o primeiro acesso, cada instância irá receber mensagens de atualização de posições do estado disponíveis localmente.

2.5 Modelo

2.5.1 Aplicação

Seja $\mathcal{F} = \{F_k \mid k \in \mathbb{N}\}$ um conjunto finito não-vazio de filtros. Podemos definir mais formalmente um filtro F, como uma função $F : \mathcal{I}^* \rightarrow \mathcal{O}^*$, onde $\mathcal{I}^*$ e $\mathcal{O}^*$ são os conjunto de todas as entradas e saídas possíveis de filtros. Consideremos $\mathcal{S} = \{(F_i, F_j) \mid i, j \in \mathbb{N}\ e\ i, j < |\mathcal{F}|\}$ um conjunto finito não-vazio de fluxos. Cada membro $S_{ikj},\ k \in \mathbb{N}\ e\ (F_i, F_j) \in \mathcal{S}$ é definido como a representação de um fluxo partindo de F_i e chegando em F_j. Uma vez que k apenas se faz necessário para diferenciar o caso em que temos vários fluxos, com a mesma direção, ligando dois filtros, a partir de agora só o explicitaremos para evitar ambigüidades.

Assim, de forma simplificada, o *pipeline* de execução de uma aplicação Anthill pode ser descrito da seguinte forma: cada vértice $F \in \mathcal{F}$ executa uma transformação sobre um conjunto de dados que resulta em um outro conjunto de dados. Esse último serve de entrada para outro vértice do grafo e assim sucessivamente. Mais formalmente, definimos uma aplicação Anthill como o grafo direcionado rotulado $\mathcal{A} = (\mathcal{F}, \mathcal{S}, \mathbb{N})$.

Uma transformação F é dita regular quando seu custo estrito pode ser previsto para todas as entradas durante uma execução. Para ilustrar esse conceito, consideremos uma função F que retorna a soma dos elementos de um arranjo A. Supondo que F consista em iterar sobre todo o arranjo, somando seus elementos, temos um custo de execução $\Theta(|V|)$, para um vetor V de entrada. Assim, podemos definir essa função de transformação como regular.

De forma complementar, uma transformação é dita irregular quando tal estimativa não pode ser feita. Por exemplo, quando o tempo de execução de F é dependente dos elementos da entrada. Consideremos uma função que calcula a solução de um sistema linear através de métodos numéricos iterativos. Para cada entrada, F pode encontrar a solução em uma, duas ou em um número máximo de iterações (parâmetro). Note que

o tempo de execução de F é dependente das entradas e das configurações do sistema, não sendo possível tal previsão antes da execução da função.

Qualquer menção a $\mathcal{F}$ ou $\mathcal{S}$ estará sempre associada a uma aplicação $\mathcal{A}$ qualquer e será feita explicitamente quando se fizer necessário. Usaremos a notação de ponto para nos referir a membros de conjuntos e/ou tuplas. Por exemplo, $\mathcal{A}.\mathcal{F}$ e $\mathcal{A}.\mathcal{S}$ representam os conjutos de filtros e fluxos da aplicação $\mathcal{A}$, respectivamente.

2.5.2 Plataforma de execução

Definimos uma plataforma de execução, como o grafo não direcionado completo formado pelo conjunto de vértices $\mathcal{R} = \{R_i \mid i \in \mathbb{N}\}$. Cada componente deste conjunto representa um nodo de computação, sendo definido como $R_i = (i, c), i \in \mathbb{N}\ e\ c \in \mathbb{R}$, onde i representa a identificação única do recurso na plataforma e c a capacidade do mesmo.

Uma plataforma de execução é dita homogênea quando todos os recursos possuem a mesma capacidade ($\forall\,(R_i, R_j \in \mathcal{P})\,(R_i.c = R_j.c)$ e heterogênea caso contrário.

2.5.3 Cópias transparentes

O mecanismo de cópia transparente foi criado com o objetivo de possibilitar a exploração transparente do paralelismo de dados. Esse mecanismo permite a replicação de qualquer vértice do grafo da aplicação em vários recursos da plataforma. Os dados que iriam inicialmente para um determinado filtro podem ser particionados entre as várias cópias (ou instâncias), atingindo-se o paralelismo de dados.

Uma instância de filtro pode ser definida como a tupla $I_{kij} = (k, F_i, R_j),\ k \in \mathbb{N},\ F_i \in \mathcal{F}\ e\ R_j \in \mathcal{R}$. Definimos também operador $I(F_i)$ ou I_i, que retorna a configuração do filtro, ou seja, o conjunto formado por todas as instâncias do filtro F_i. A união das configurações de todos os filtros que compõem a aplicação é chamada de configuração da aplicação. Uma vez que k se faz necessário somente nos casos onde queremos diferenciar entre diferentes instâncias de um mesmo filtro executando em um mesmo nodo de computação, esse será explicitado somente para evitar ambigüidades.

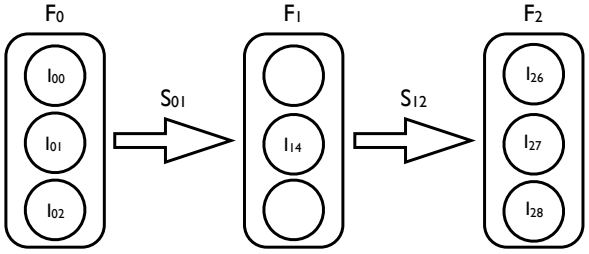
A Figura 2.8 ilustra os conceitos apresentados até aqui. Nela é mostrada a configuração de uma aplicação composta por três filtros. A execução ocorre em uma plataforma de execução composta por nove recursos. Primeiramente podemos destacar o conjunto de filtros, $\mathcal{F} = \{F_0, F_1, F_2\}$, de fluxos $\mathcal{S} = \{S_{01}, S_{12}\}$ e o conjunto de recursos $\mathcal{R} = \{R_0, R_1, ..., R_8\}$, os quais compõem a plataforma.

O filtro F_0 possui três instâncias, $I(F_0) = I_0 = \{I_{00}, I_{01}, I_{02}\}$, executando nos recursos R_0, R_1 e R_2, respectivamente. O filtro F_1 possui apenas uma instância, $I(F_1) =$

$I_1 = \{I_{14}\}$, executando no recurso R_4. Por fim, o filtro F_2 possui três cópias, $I(F_2) = I_2 = \{I_{26}, I_{27}, I_{28}\}$, executando nos recursos R_6, R_7 e R_8, respectivamente.

A configuração desta aplicação é $\{I_{00}, I_{01}, I_{02}, I_{14}, I_{26}, I_{27}, I_{28}\}$.

Figura 2.8. Exemplo de aplicação de cópia transparente.



2.5.4 Roteamento de mensagens

Políticas de roteamento definem um protocolo para escolha dos destinatários das mensagens que trafegam através dos fluxos, deixando a cargo do ambiente realizar a entrega correta das mesmas e escondendo assim os detalhes de implementação inerentes.

Sejam S_{ij} um fluxo qualquer que liga dois filtros quaisquer i e j, $rand()$ uma função que retorna um número aleatório no intervalo $[0, 1)$, $inc()$ uma função que retorna incrementalmente os naturais no intervalo $[0, \infty)$ e *extraiRotulo* uma função definida pelo programador para extração de rótulos de mensagens, as três políticas de roteamento existentes na versão corrente do Anthill podem ser definidas:

1. Um caso onde mensagens enviadas através de um fluxo são entregues a próxima instância destino seguindo um algoritmo do tipo *round-robin*.

$$rr(msg, I_j) = \{I_{ik} \mid k = inc() \ mod \ |I_j|\} \tag{2.1}$$

2. Uma política *broadcast*, onde todas as instâncias do filtro destino recebem uma cópia da mensagem:

$$broadcast(msg, I_j) = \{I_j\} \tag{2.2}$$

3. O fluxo-rotulado, onde toda mensagem enviada é analisada por uma função que identifica uma informação especial: o rótulo (*label*). O formato do rótulo é dependente de cada aplicação, desta forma, é definido pelo programador. Após a identificação do rótulo da mensagem, outra função é aplicada, desta vez sobre o rótulo. O padrão para essa função é baseado no resto da divisão inteira (*mod*). O resultado dessa composição de funções irá determinar a instância destino da

mensagem.

$$fluxo\_rotulado(msg, I_j) = \{I_{ik} \mid k = extraiRotulo(msg) \bmod |I_j|\} \quad (2.3)$$

2.6 Resumo

Neste capítulo apresentamos em detalhes o Anthill. Um ambiente de programação paralela baseado no paradigma filtro fluxo. Ele permite o processamento eficiente de grandes volumes de dados, uma vez que o paralelismo pode ser explorado de maneira simples e intuitiva. Foram apresentados os conceitos básicos do paradigma filtro fluxo, bem como detalhes a cerca da execução de aplicações. Por fim, foi apresentado uma modelagem do ambiente, a qual será utilizada em demonstrações no decorrer do trabalho.

Entretanto, o Anthill foi construído como um ambiente de execução estático e, como tal, não permite que a distribuição dos componentes da aplicação seja modificada em tempo de execução. O que vai de encontro a natureza dinâmica das plataformas distribuídas atuais. Nos demais capítulos apresentaremos um conjunto de extensões que adicionam suporte eficiente ao aumento dinâmico da plataforma execução de aplicações Anthill.

Capítulo 3

Adição Dinâmica de Instâncias

Diferenças nos padrões de processamento de uma aplicação Anthill podem provocar mudanças na carga dos diferentes filtros ao longo do tempo. Além disso, a plataforma onde a aplicação está sendo executada pode mudar e, por exemplo, novos nodos de computação podem se tornar disponíveis durante o processamento da mesma. Nessas situações, pode fazer sentido variar o número de instâncias de filtros durante a execução.

Outra situação em que o ambiente precisa se ajustar a uma nova configuração é na ocorrência de falhas. Se uma instância se torna indisponível durante o processamento de uma aplicação, a carga a que esta instância estava sendo submetida pode ser automaticamente redistribuída entre as demais instâncias, ou até mesmo migrada para uma nova cópia, instanciada em alguma máquina menos carregada.

Em linhas gerais, reconfiguração dinâmica implica na capacidade de modificar o mapeamento entre componentes da aplicação e os recursos onde os mesmos estão sendo executados, mantendo o prosseguimento consistente da execução. Isto implica na capacidade de, durante sua execução, utilizar mais recursos que venham a se tornar disponíveis ou suportar de forma consistente a remoção dos mesmos. O objetivo principal deste trabalho é dar um primeiro passo no sentido de tornar o Anthill um ambiente reconfigurável. Definimos esse passo como o crescimento dinâmico da plataforma, ou seja, a adição de instâncias de filtro em tempo de execução. A partir deste ponto qualquer menção à reconfiguração dinâmica (ou simplesmente reconfiguração) estará se referindo apenas ao aumento do número de instâncias em tempo de execução.

O restante do capítulo é organizado da seguinte forma: Na Seção 3.1 apresentaremos uma motivação para a preservação da assincronia e em seguida o funcionamento do mecanismo de suporte a adição dinâmica de instâncias no Anthill é explicado sem fazer referência específica a nenhuma das políticas de roteamento. Na Seção 3.2 mostramos que as quatro políticas de roteamento existentes no Anthill funcionam de forma correta, mesmo em face a reconfigurações dinâmicas. Em seguida, a Seção 3.3 apresenta uma

extensão no modelo do Anthill apresentado na Seção 2.5. Estas extensões tornam o modelo apto a lidar com reconfigurações dinâmicas e são necessárias para algumas demonstrações apresentadas mais adiante. Por fim, a Seção 3.4 mostra detalhes de implementação e a *interface* de programação exportada pela solução.

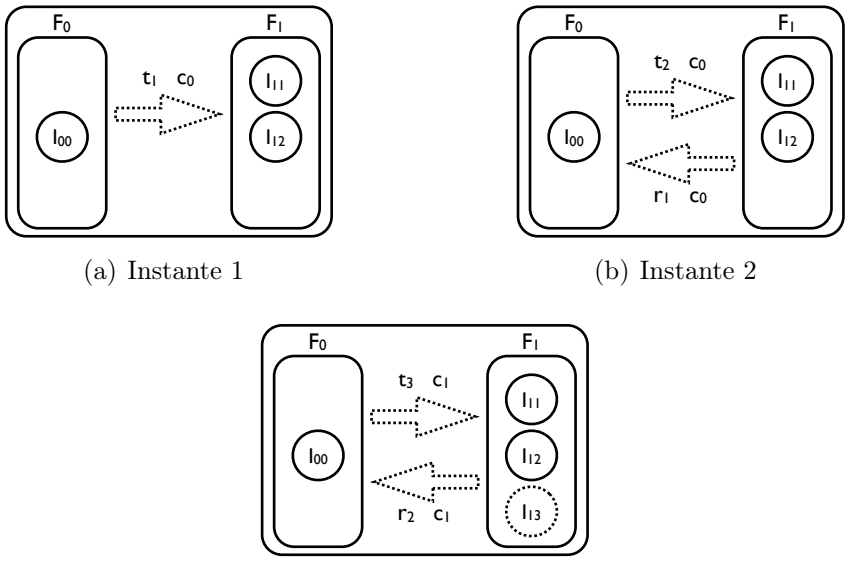
3.1 Protocolo de Funcionamento

Para filtros que não possuem estado, o mecanismo de reconfiguração precisa apenas propagar corretamente as alterações ocorridas na configuração da aplicação, uma vez que não existem requisitos de dependência ou particionamento do estado. Podemos ilustrar essa situação através das aplicações embaraçosamente paralelas, como por exemplo as simulações Monte Carlo [Metropolis e Ulam, 1949], aplicações de buscas massivas e varredura de parâmetros [Abramson et al., 2000] e algumas aplicações de processamento de imagens [Kong et al., 2007].

Como exposto no Capítulo 2, a assincronia é uma poderosa característica que aplicações Anthill podem utilizar. A Figura 3.1 ilustra o efeito da adição dinâmica de recursos em uma aplicação assíncrona. A aplicação é composta pelos filtros F_0 e F_1 e por dois fluxos os ligando. Esses fluxos estão em sentidos opostos, compondo um ciclo. Digamos que o primeiro filtro envia candidatos à solução e o segundo filtro verifica se esses candidatos são realmente soluções. Os candidatos avaliados como solução são enviados de volta para F_0, que por sua vez utiliza-os para a geração de mais candidatos. Por fim, a iteração continua até que não restem novos candidatos. Para os fins deste estudo, podemos supor uma política de roteamento *broadcast* em ambos os fluxos.

No primeiro instante, temos o ambiente na configuração inicial, identificada por 0, o que pode ser visto na Figura 3.1(a). O filtro F_0 envia a tarefa 1 para o filtro F_1, o que é representado pelo par t_1 c_0 acima da seta à direita. Na Figura 3.1(b) ilustramos o segundo momento, onde temos a resposta r_1 enviada pelo filtro F_1 e o envio de mais uma tarefa (t_2) pelo filtro F_0. Por fim, vamos supor que logo após a chegada da tarefa t_2 e antes do envio da tarefa t_3 ocorra uma reconfiguração e a instância I_{13} é adicionada, cenário representado pela Figura 3.1(c). A resposta a essa tarefa, r_2, e nova tarefa, t_3, foram criadas e enviadas em conformidade com os parâmetros da nova configuração. Uma vez disparada, t_3 pode ser executada concorrentemente com t_2, por exemplo, I_{13} executa t_3 enquanto I_{11} e I_{12} continuam executando t_2. Como pode ser notado, a possibilidade de assincronia no contexto da adição de recursos em tempo de execução é uma poderosa característica.

Assim, propor um modelo assíncrono para adição de instâncias de filtro no Anthill implica em apresentar uma forma de lidar com o roteamento consistente de mensagens



(a) Instante 1

(b) Instante 2

(c) Instante 3

Figura 3.1. Reconfiguração Dinâmica: Assincronia

em várias configurações, numa mesma execução. Em linhas gerais, o modelo proposto para adicionar o suporte à reconfiguração dinâmica no Anthill é baseado no conceito de relógios lógicos, proposto por Lamport [1978]. Assim como em Lamport [1978], usamos um contador monotonicamente crescente para conseguir uma ordenação parcial do tipo "aconteceu-antes" num sistema assíncrono e com isso processar e rotear mensagens de forma consistente.

No modelo de reconfiguração dinâmica proposto, o relógio lógico representa identificação da configuração da aplicação e esse é incrementado toda vez que ocorre uma mudança de configuração (evento). Este evento carrega a descrição da nova configuração da aplicação. Assim, o ambiente será responsável por armazenar as descrições de configurações e adicionará ao cabeçalho de cada mensagem um campo que indica a configuração na qual ela foi criada. O comportamento das instâncias no recebimento e envio de mensagens é descrito a seguir:

- **Recebimento de mensagens**

  Um requisito para execução consistente no Anthill é que os dados precisam ser processados na mesma configuração em que foram criados. Isso porque é possível que o correto comportamento da aplicação esteja condicionado à configuração no momento da criação da mensagem. Por exemplo, pode-se utilizar o número de instâncias de um determinado filtro como chave de criptografia das mensagens enviadas. Assim, o filtro receptor precisa desse número para processar a mensagem. Tal fato só é possível pois a API Anthill exporta métodos de consulta a

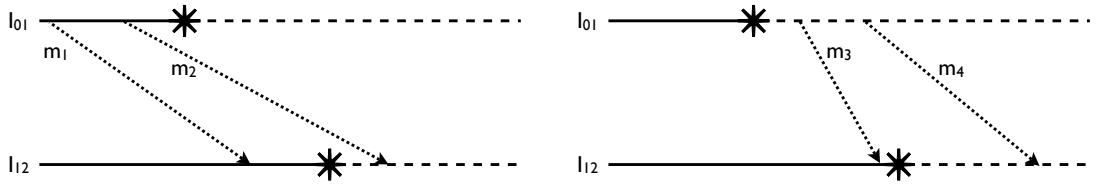
configurações do ambiente, os quais podem ser utilizados para otimizações.

Assim, os cabeçalhos de todas as mensagens recebidas são avaliados e as instâncias destino alteram seu estado interno para a configuração indicada no cabeçalho antes de executar a rotina de processamento devida. Dessa forma, API do Anthill irá responder segundo a configuração na qual a mensagem foi criada, mantendo assim a consistência entre a configuração de criação e processamento das tarefas. As mensagens recebidas de configurações ainda não conhecidas vão para uma fila de espera aguardar processamento. Uma vez que a notificação de reconfiguração chega, esta fila é percorrida e as mensagens referentes a esta nova configuração são processadas.

- **Envio de mensagens**

  As mensagens de saída recebem em seu cabeçalho a identificação da configuração marcada pelo valor do relógio lógico corrente na instância. Esse valor será, no mínimo, a identificação da configuração em que a mensagem foi criada, uma vez que, para que o processamento de uma mensagem ocorra, a instância precisa conhecer a configuração de criação da mensagem.

A Figura 3.2 mostra duas linhas do tempo de duas instâncias de filtro. Nelas temos as notificações de reconfiguração representadas por estrelas e as mensagens trocadas representadas pelas setas.



(a) Exemplo 1 (b) Exemplo 2

Figura 3.2. Assincronia no envio de mensagens

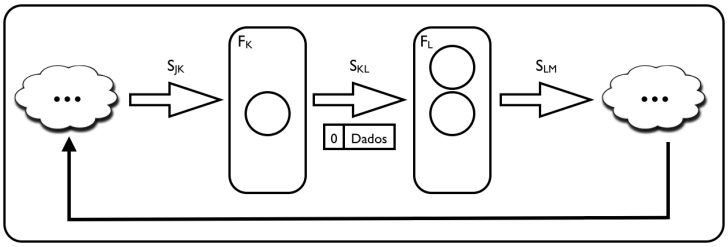
No instante inicial as instâncias possuem a mesma configuração corrente, o que é indicado pelo traço contínuo de ambas as linhas de tempo. No primeiro exemplo, ainda na configuração inicial, a qual vamos chamar de 0, a instância I_{01} envia a mensagem m_1. Como a mensagem é recebida ainda na configuração 0 (conhecida por ambas as instâncias), ela é processada segundo essa configuração e a mensagem resultante desse processamento será marcada de acordo.

Dando prosseguimento ao exemplo, a mensagem m_2 é enviada e tem em seu cabeçalho a identificação de configuração 0, valor do relógio lógico de I_{01} no momento do envio.

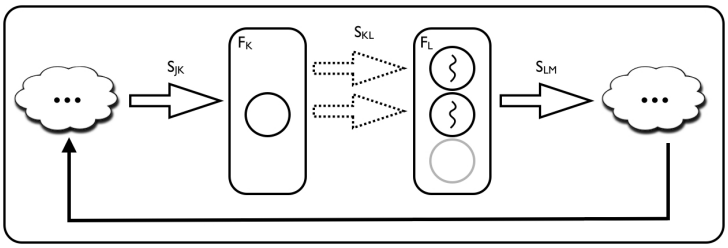
Porém, m_2 é recebida por I_{12} após a chegada da notificação de reconfiguração. Esta nova configuração é indicada pela linha tracejada e será identificada por 1. Mesmo tendo seu relógio lógico corrente configurado para 1, a instância I_{12} irá processar a mensagem segundo a configuração 0, identificada no cabeçalho da mensagem.

No segundo exemplo, a situação contrária ocorre com a mensagem m_3. Essa mensagem é enviada na configuração 1 e recebida quando a configuração corrente ainda é 0. Neste caso ela irá para a fila de espera e será processada no momento em que a notificação contendo a descrição da configuração 1 for recebida. Por fim, uma situação análoga a m_1 ocorre com m_4, só que agora ambas as instâncias possuem 1 como configuração corrente.

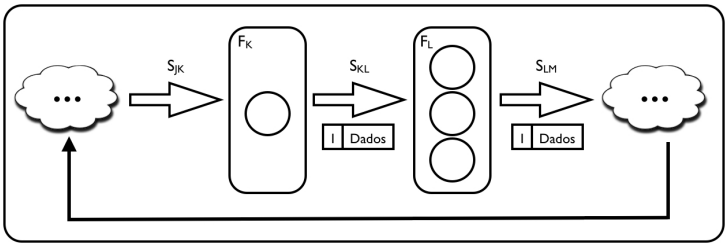
Para fixar os conceitos, a Figura 3.3 mostra uma visão mais prática do comportamento do ambiente quando exposto a uma situação similar à descrita. Novamente, para fins de ilustração, vamos definir que a política de roteamento que liga os dois filtros F_K e F_L é *broadcast*.



(a) Instante 0



(b) Instante 1



(c) Instante 2

Figura 3.3. Exemplo do Comportamento do Modelo de Adição Dinâmica de Instâncias de Filtros

Primeiramente a instância do filtro F_K processa um conjunto de dados e os encaminha para envio (Figura 3.3(a)). O cabeçalho da mensagem possui a identificação da configuração na qual ela foi criada, no caso, a configuração de número 0. Antes da mensagem ser roteada ocorre uma reconfiguração e uma instância do filtro F_L é adicionada. Neste caso optamos por uniformizar o comportamento do sistema e assumimos uma postura restritiva. Assim, a configuração utilizada para roteamento é a configuração que está indicada no cabeçalho da mensagem, ou seja, somente as instâncias de F_L que fazem parte da configuração 0 irão receber e processar a mensagem. Isso que é ilustrado na Figura 3.3(b). Ao receber as mensagens, o processamento dos dados pelas instâncias de F_L ocorre segundo os parâmetros da configuração 0 e todas as chamadas à API do Anthill respondem segundo essa configuração.

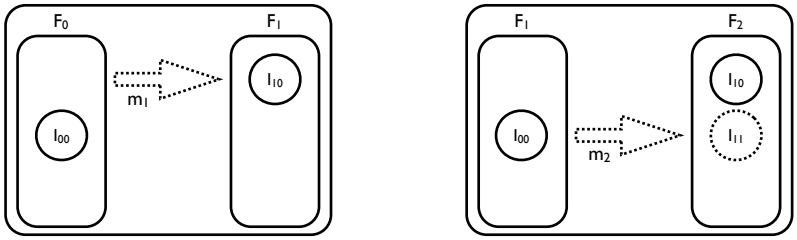
A Figura 3.3(c) ilustra o envio do resultado de um processamento. Note que as mensagens de saída recebem em seu cabeçalho a identificação da configuração marcada pelo valor do relógio lógico corrente na instância. Supondo que a aplicação é assíncrona, podemos propor o caso em que o ciclo é completado e chegam ao filtro F_B mensagens criadas já na configuração 1, porém ainda existem mensagens da configuração 0 executando. Nesse caso, não existe impedimento à existência de execuções simultâneas de diversas configurações por instâncias diferentes do filtro F_B. Assim, a instância que recebeu essa última mensagem muda sua configuração para 1 no momento do processamento e, como nesse caso 1, é o valor da configuração mais atual, a mensagem resultante do processamento, terá o cabeçalho identificado com o valor de configuração 1.

3.2 Políticas de Roteamento

Quando um fluxo é configurado para rotear mensagens segundo a política *round robin*, o algoritmo de mesmo nome é usado para escolha do nodo destino de cada mensagem. Pela definição 2.1, exposta na Seção 2.5.4, o correto funcionamento da política *round robin* depende apenas do tamanho do conjunto de instâncias do filtro destino e da função $inc()$. Sendo que essas últimas não são dependentes de mudanças ocorridas na configuração da aplicação.

Vamos supor o cenário ilustrado na Figura 3.4. Temos dois filtros F_1 e F_2 e cada filtro possui uma única instância, na configuração inicial. As mensagens partem de F_1 para F_2 e são entregues usando o protocolo de roteamento *round robin*. Supondo que F_1 envia uma mensagem para F_2, aplicando a definição exposta na Seção 2.5.4, temos que $inc() = 0$, $|I_2| = 1$ e $0 \; mod \; 1 = 0$, o que resultará na entrega da mensagem para a instância de identificação 0, I_{10}. O que pode ser visto na Figura 3.4(a)

Vamos supor que após o envio da mensagem ocorre a adição de uma instância de F_2, que por sua vez é seguida pela criação e envio de outra mensagem, m_2. Supondo que a instância de F_1 seja avisada antes do criação desta nova de m_2, essa deve ser entregue segundo os novos parâmetros de configuração do ambiente: $inc() = 1$, $|\beta_{10}| = 2$ e $1\ mod\ 2 = 1$, o que resultará na entrega da mensagem para a instância de identificação 1, I_{11}. A Figura 3.4(b) ilustra o correto comportamento do ambiente neste novo cenário.



(a) Configuração 0 (b) Configuração 1

Figura 3.4. Reconfiguração Dinâmica: Round robin

Caso m_2 fosse criada antes de I_{00} receber o aviso de reconfiguração, essa seria roteada segundo a configuração inicial. Ou seja, teríamos que $inc() = 0$, $|I_2| = 1$ e $0\ mod\ 1 = 0$, o que resultaria na entrega da mensagem para a instância I_{10} novamente.

Podemos estender esse caso para as demais políticas de roteamento. Para as políticas *broadcast* e fluxo rotulado apenas o conjunto de instâncias destino muda de acordo com a configuração definida pela visão do relógio lógico da instância.

3.3 Extensão do Modelo

Um vez que o modelo exposto na Seção 2.5 não prevê reconfigurações, propomos uma extensão do mesmo onde definimos mais formalmente as mudança configuração. Uma vez que $I(F_0)$ representa o conjunto de instâncias do filtro F_0, definimos $I_k(F_0)$ ou β_{0k}, o conjunto de instâncias do mesmo filtro, num determinado instante da execução da aplicação, após k configurações. Podemos omitir a identificação do filtro, quando esta não se fizer necessária.

O histórico das configurações (visões) assumidas por β até a k-ésima reconfiguração é definido como $\mathcal{B} = \{\beta_0, \beta_1, ... \beta_k\}$ e o conjunto formado por todas as instâncias que fazem parte desse histórico é $\mathcal{H} = \bigcup_{i=0}^{i<k} \beta_i$.

3.4 Implementação e Interface de Programação

A adição desta funcionalidade no Anthill foi facilitada pela existência de uma entidade central que gerencia a execução de cada aplicação (console). Essa entidade possui três responsabilidades principais: (i) coordenar o início da execução, o que consiste em iniciar os processos remotos e enviar mensagens contendo a configuração da aplicação e os parâmetros passados pelo usuário; (ii) coordenar o término da aplicação, o que é feito através do recebimento de mensagens individuais de término e propagação de uma mensagem de término global; e (iii) coordenar a transição entre os estágios de início, processamento e finalização.

Não é objetivo deste trabalho o estudo de estratégias de otimização para utilização de recursos, o que deve ser estudado em trabalhos futuros. Assim optamos por exportar uma interface geral que pode ser utilizada por qualquer módulo Anthill, por exemplo, um módulo especializado em escalonamento. A adição de instâncias de filtro é iniciada com a invocação da chamada:

```
void createFilterInstance(filterName, hostNames, numberOfInstances)
```

Onde filterName indica o nome do filtro a ser copiado, hostNames é um vetor que contém os nomes dos nodos de computação onde as cópias serão iniciadas e numberOfInstances é o número de instâncias que serão adicionadas. Ocorrerá erro caso numberOfInstances seja maior que a quantidade de elementos de hostNames e em caso contrário serão utilizados os numberOfInstances primeiros elementos de hostNames.

Uma vez invocada essa função, é construída uma estrutura que descreve as novas instâncias, que é enviada ao console, empacotada numa mensagem de reconfiguração. Esse primeiro passo é ilustrado na Figura 3.4, por uma seta partindo da entidade Reconfigurador. A entidade Reconfigurador representa qualquer módulo Anthill que venha a ser responsável por decidir sobre reconfigurações. A seta possui em seu rótulo a chamada da função createFilterInstance com os parâmetros "B", HN e NI, indicando que será criada 1 instância do filtro nomeado "B"no nodo "N".

A invocação desta função implica no envio de uma mensagem de controle para o console. Esta mensagem possui o rótulo MSGT_CREATE_INSTANCE e o seu conteúdo possui a descrição das novas instâncias a serem criadas. Ao receber mensagens com o rótulo MSGT_CREATE_INSTANCE, o console irá tentar efetuar a criação das instâncias requeridas (FilterSpec.c:fsSpawnOneInstance()) e, caso esta ocorra com sucesso, o console criará uma estrutura que contém os descritores da nova configuração

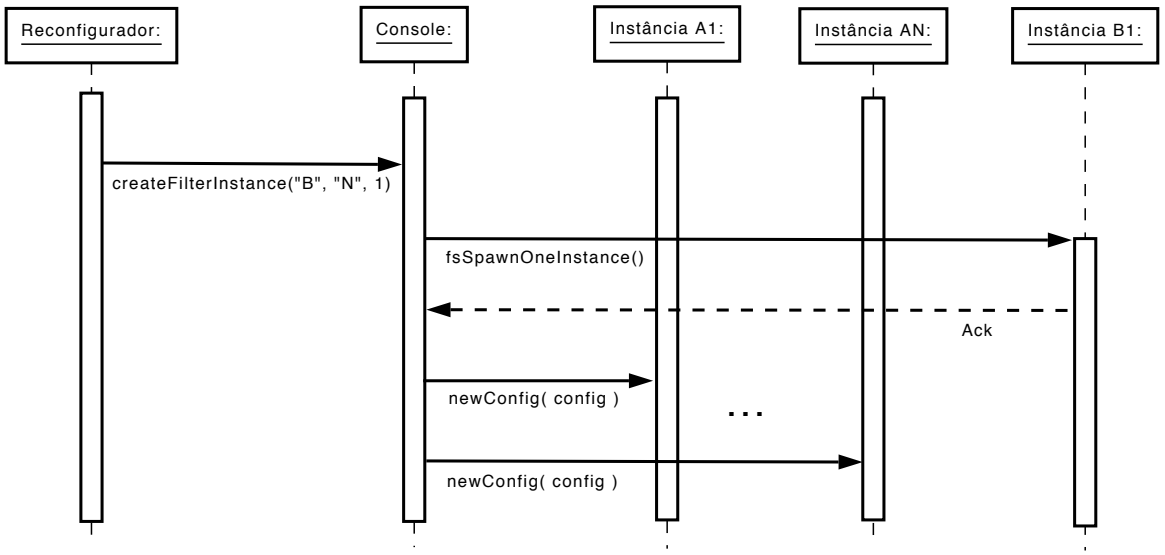


Figura 3.5. Diagrama de seqüência do protocolo de reconfiguração adicionado no Anthill

da aplicação. Caso não seja possível iniciar alguma das instâncias requeridas a aplicação termina com erro (Manager.c:appendWork()).

A tarefa de construir uma estrutura contendo os descritores da configuração de toda a aplicação só é possível pois o console centraliza informações sobre todo o sistema em execução. Uma vez criada esta estrutura, o descritor da nova configuração é empacotado em uma mensagem de notificação de reconfiguração (rótulo MSGT_RECONF) que é enviada para as demais instâncias de todos os filtros. O tratamento de mensagens desse tipo implica na atualização dos relógios lógicos das instâncias e e na adição dessa nova configuração na lista dos descritores de configuração (FilterDev.c:dsTreatMessageFromManager()). A partir desse momento as instâncias estão prontas para processar tarefas criadas nessa nova configuração.

3.5 Resumo

Diferenças nos padrões de processamento de uma aplicação Anthill ou mudanças na plataforma de execução são situações onde pode fazer sentido variar o número de instâncias de filtros durante a execução. Neste capítulo apresentamos o funcionamento do mecanismo de suporte a adição dinâmica de instâncias no Anthill. Foi proposta uma extensão no modelo do Anthill (Seção 2.5) que torna esse modelo apto a lidar com reconfigurações dinâmicas. Por fim, foram mostrados os detalhes de implementação e a *interface* de programação exportada pela solução.

O problema de particionamento do estado em aplicações Anthill pode ser resolvido de forma eficiente e compatível com o modelo filtro fluxo pelo mecanismo de fluxo

rotulado. Com a adição do suporte à adição dinâmica de recursos, a localização e/ou número de instâncias de filtro pode mudar durante a execução da aplicação. Nesse cenário pode ser necessário migrar dados armazenados nesse estado. Para solucionar esse problema foi adicionado ao Anthill um novo módulo chamado de Estado Global Distribuído (EGD), o qual é descrito no próximo capítulo.

Capítulo 4

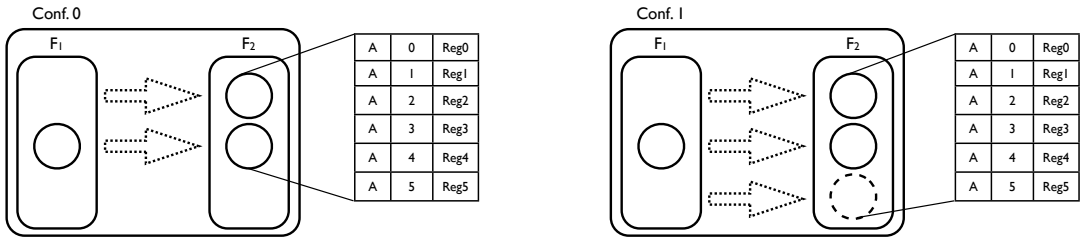
Estado Global Distribuído

Como mencionado anteriormente, cada estágio do *pipeline* pode ser replicado nos vários nodos de computação da plataforma distribuída. Freqüentemente existe a necessidade de acessar e modificar dados derivados do processamento de mensagens, o qual chamamos de estado do filtro. Embora o mecanismo de cópia transparente já tenha sido implementado no ambiente DataCutter [Beynon et al., 2001], esse possui apenas duas formas de implementar filtros com estado distribuído neste ambiente: replicar todo o estado e adicionar um filtro para agregação dos resultados, ou explicitamente particionar o estado entre os filtros, o que elimina a transparência do mecanismo de cópia transparente.

Nenhuma dessas soluções é satisfatória, uma vez que elas aumentam o trabalho dos desenvolvedores e/ou incorrem em custos adicionais. A política de fluxo rotulado [Veloso et al., 2004; Ferreira et al., 2005] resolve esse problema de forma genérica. Os programadores apenas tem que definir um rótulo que associa cada mensagem a uma variável do estado e o ambiente de execução garante que todas as mensagens com o mesmo rótulo são entregues à mesma instância de filtro.

Mesmo tendo resolvido o problema de particionamento do estado em tempo de execução, um problema ainda permanece sem solução: viabilizar a migração dos dados armazenados nesse estado. Tal característica pode ser necessária por várias razões, entre elas, quando a localização e/ou número de instâncias de filtro muda durante a execução da aplicação. Usaremos a Figura 4.1 para ilustrar esse último caso. No primeiro quadro (Figura 4.1(a)), temos que a aplicação está na configuração inicial, onde os filtros F_1 e F_2 possuem uma e duas instâncias respectivamente. Existe um fluxo partindo de F_1 e mensagens são entregues segundo a política de fluxo rotulado às instâncias de F_2. Consideremos que F_2 possui um estado compartilhado por suas instâncias.

Suponhamos a adição de uma instância de F_2 durante a execução da aplicação



(a) Configuração 0 (b) Configuração 1

Figura 4.1. Reconfiguração do Estado do Filtro

em questão. Esse cenário é ilustrado na Figura 4.1(b), onde vemos que, nessa nova configuração, as mensagens enviadas por F_1 podem ser entregues a três instâncias de F_2. Neste caso, o estado deve ser reparticionado e a nova instância de F_2 receberia uma partição deste.

Este capítulo descreve um novo módulo Anthill chamado de Estado Global Distribuído (EGD). Esse módulo foi criado para solucionar o problema de movimentação de dados do estado entre instâncias de um mesmo filtro. Em linhas gerais, o estado é abstraído como um espaço de endereçamento único, tal como em uma memória compartilhada distribuída. A adoção dessa abstração implica em localizar, acessar, migrar e/ou replicar os dados compartilhados, mantendo a consistência na propagação destes acessos entre os nodos da rede. Uma vez que o ambiente se encarrega dessa tarefa, a programação se torna mais simples. No Anthill, o módulo de gerenciamento dinâmico do estado (EGD) garante a premissa de que a informação armazenada nesta área compartilhada estará disponível para todas as cópias de filtro em qualquer momento da execução da aplicação, mesmo que seja necessário migrá-la de uma cópia pra outra.

4.1 Funcionamento

Uma comparação entre os paradigmas de programação de sistemas de memória compartilhada e distribuída é apresentada por Tanenbaum [1994]. O autor argumenta que a programação de sistemas de memória compartilhada é considerada mais simples, uma vez que a comunicação entre processos resume-se a escrita e leitura de dados da memória e semáforos ou monitores podem ser usados para evitar condições de corrida. Para programação de sistemas compostos de diversos computadores, geralmente é utilizado paradigma de troca de mensagens. Nesse paradigma, a comunicação é realizada através de primitivas que explicitamente controlam a movimentação dos dados, o que traz uma série de dificuldades. Em resumo, *clusters* são simples de construir mas difíceis de programar, sendo os multiprocessadores o oposto, difíceis de construir mas simples de

programar [Tanenbaum, 1994]. Nesse contexto nasceram as memórias compartilhadas distribuídas (DSM) [Keleher et al., 1994; Bershad et al., 1993; Bal et al., 1992; Carter, 1995; Carriero et al., 1994].

Um sistema DSM oferece a abstração de um espaço de endereçamento lógico a que todos os componentes de um sistema distribuído têm acesso. Podendo ser implementados tanto em software quanto em hardware, esses sistemas controlam a distribuição física dos dados, oferecendo ao programador acesso transparente à memória virtual compartilhada. Nesse sentido, o objetivo principal é ocultar do programador a comunicação inerente ao acesso distribuído e fornecer um modelo de programação baseado em dados compartilhados ao invés de troca de mensagens. Além disso, esses sistemas aproveitam a escalabilidade e a relação custo/eficiência aos *clusters* de memória distribuída [Tanenbaum, 1994].

Utilizando DSMs o programador pode focar no desenvolvimento do algoritmo paralelo, ao invés de investir tempo em questões relacionadas a distribuição dos dados utilizados pelo mesmo. Somado a esse benefício, ainda provêem uma interface de programação semelhante às oferecidas pelas linguagens de programação seqüenciais, viabilizando um aumento da portabilidade. A figura 4.1 ilustra uma DSM composta por N estações de trabalho, cada um com sua própria memória física, conectadas por uma infra estrutura de redes de computadores.

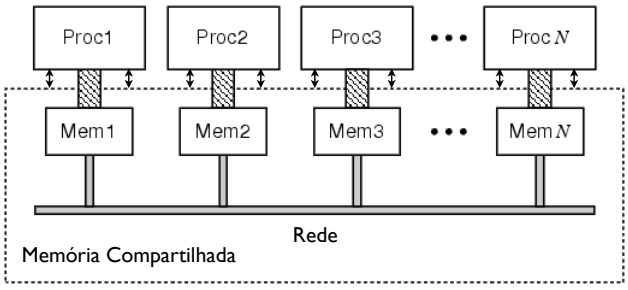


Figura 4.2. Sistema de memória distribuída.

A maioria das implementações de memória compartilhada distribuída analisadas são de uso geral e, visando aumento do desempenho, os dados compartilhados são replicados na memória volátil de múltiplas máquinas [Bal et al., 1992; Bennett et al., 1990; Bershad et al., 1993; Keleher et al., 1994]. Esse cenário nos leva ao problema da manutenção da consistência entre múltiplas cópias.

O modelo de consistência de memória é a política que determina como e quando mudanças na memória feitas por um processador são vistas pelos outros processadores do sistema. Esse modelo define um contrato entre a aplicação e o programador que

utiliza o sistema, impondo regras específicas para a programação e garantindo uma previsibilidade do comportamento do mesmo, caso essas regras sejam respeitadas. Vários modelos consistência têm sido propostos para otimizar implementações de DSMs. Eles utilizam variáveis de sincronização ou travas para garantir a consistência, o que geralmente leva a perda de desempenho. Um vez que estamos lidando com um modelo assíncrono, essas travas são indesejáveis.

Como foi mostrado, a utilização do fluxo rotulado leva à uma alta localidade de referência. Assim, na grande maioria dos casos, apenas uma instância irá acessar um item do estado durante a execução de uma aplicação. Esta situação só é mudada com a presença de reconfigurações, que leva a uma redistribuição do estado. Nesse cenário específico, onde cada item de dado vai ser acessado por poucas instâncias, não vale a pena utilizar cópias e arcar com os custos das travas e outros mecanismos para garantia de consistência. Além disso, cogitamos adaptar uma implementação de DSM existente, a TreadMarks Keleher et al. [1994], mas após uma análise do código chegamos a conclusão que a alteração levaria mais tempo que implementar o EGD desde o início.

Dessa forma, nos baseamos nos conceitos de memória compartilhada distribuída e optamos por implementar todo o Espaço Global Distribuído. Esse módulo abstrai o estado, que é distribuído entre as máquinas, como um espaço de endereçamento único e é otimizado para aplicações com alta localidade de referência. O restante dessa seção apresenta da estrutura geral dessa solução.

4.1.1 Estrutura

A estrutura dos dados representa a disposição global do espaço de endereçamento compartilhado, bem como a organização desses dados. Na variação mais simples, descrita por Li [1986]; Li e Hudak [1989], o espaço de endereçamento é dividido em um conjunto de blocos de tamanho fixo chamados de páginas. O acesso a uma página que não está presente na memória local gera uma exceção (falta de página) e o sistema de memória virtual trata essas faltas realizando a localização e acesso ao dado remoto. Outra solução é compartilhar somente variáveis ou estruturas de dados. Esses sistemas compartilham conjunto de variáveis com tipo, ao invés de um conjunto de blocos de memória, oferecendo ao programador um nível mais alto de abstração[Bennett et al., 1990; Bershad et al., 1993]

Na tentativa de unir os benefícios dessas duas formas de representação, os itens armazenados no Estado Global Distribuído são organizados numa lista indexada de itens. O menor bloco de dados endereçável é um elemento que compõem essa lista e esses elementos não tem tamanho nem tipo pré-definidos. Essa decisão foi impulsi-

onada pelo objetivo de alcançar as vantagens de localidade de referência inerente as aplicações alvo, uma vez que a indexação é utilizada e os desenvolvedores podem endereçar registros inteiros de um banco de dados, por exemplo. Outra motivação é não arcar com os problemas de falso compartilhamento existentes em DSMs orientadas a páginas.

Os desenvolvedores ainda tem a flexibilidade de poder variar tipo e tamanho dos objetos armazenados nos listas, bem como criar outras listas. A Figura 4.1.1 mostra um exemplo de variável que pode ser armazenada no EGD. A posição 0 desta variável possui um inteiro, a posição 1 possui a cadeia de caracteres "Daniel"e assim sucessivamente.

Figura 4.3. Exemplo de arranjo armazenado no EGD.

4.1.2 Distribuição

As duas estratégias mais comuns utilizadas para distribuição de dados compartilhados são replicação e migração. Replicação permite que várias cópias do mesmo item de dado residam em diferentes memórias locais. Esse método é escolhido principalmente para aumentar a eficiência de leituras simultâneas a um mesmo dado, porém em aplicações Anthill temos alta localidade de referência. Assim não se justifica a utilização de travas e outros mecanismos de sincronia são utilizados para garantir a consistência entre as réplicas.

Devido ao mecanismo de fluxo-rotulado, para execuções estáticas da grande maioria das aplicações Anthill, os dados armazenados no estado são acessados com alta localidade de referência. Uma vez que esse assinalamento acontece através de uma função *hash*, se uma certa fatia de dados é acessada por uma determinada instância, ela vai continuar sendo acessada apenas por aquela instância até o final da execução da aplicação.

Com o suporte a adição dinâmica de instâncias, a redistribuição do estado ocorre apenas devido a reconfiguração, para preservar o balanceamento. A redistribuição do estado (que leva à migração de dados) ocorre após uma reconfiguração quando, devido a assincronia do sistema, duas instâncias origem estiverem operando com relógios lógicos diferentes. Essa situação pode acontecer enquanto existirem mensagens criadas em diferentes configurações a serem processadas. Uma vez que durante o envio as mensagens são identificadas com o identificador de configuração mais recente da instância, a propagação da configuração antiga tende a ficar restrita.

Por esses motivos, optamos pelo desenvolvimento do EGD como uma DSM onde existe apenas uma cópia de cada um dos itens do estado. Também fazemos uso de uma estratégia agressiva de migração de dados, onde o dado é migrado ao primeiro acesso remoto.

4.1.3 Localização

Para simplificar a codificação e aproveitar a estrutura de uma aplicação Anthill, que tem um componente central (console), optamos pela implementação mais simples dos mecanismos de localização baseados em diretório [Li, 1986]. Nessa proposta temos apenas um diretório (repositório de metadados), que fica encarregado de manter informações de localização sobre todos os itens do estado. Assim, todas as operações de leitura e escrita que exigem localização de dados remotos, são enviadas para esse repositório central.

Em qualquer acesso ao estado distribuído, primeiro é checado se a variável é local ou remota. No primeiro caso, o acesso continua sem nenhuma troca de mensagens via rede. A Figura 4.4 ilustra o caso de leituras remotas. Ela apresenta um diagrama baseado nos diagramas de seqüência tradicionais, cuja particularidade é que os retângulos representam apenas os estágios bloqueantes das linhas de execução.

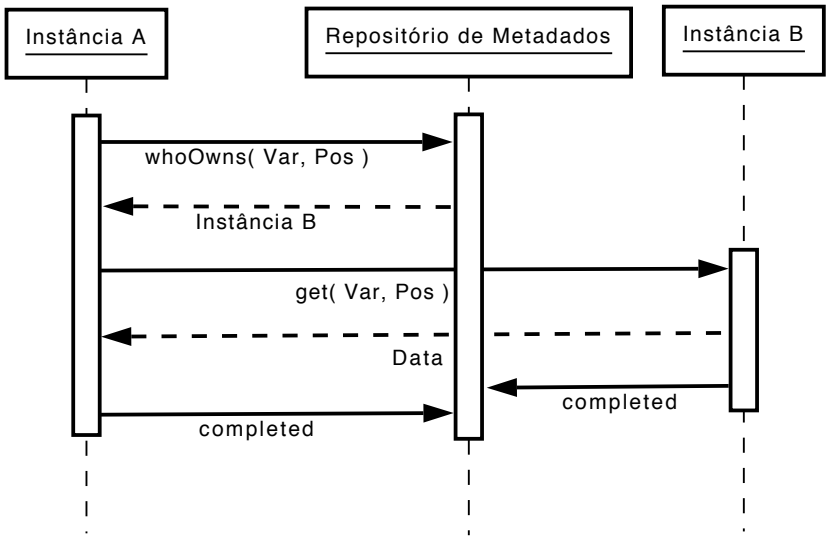


Figura 4.4. Diagrama de seqüência do acesso a dados remotos.

Primeiro o repositório de metadados (RM) é contactado para descoberta de qual recurso está fisicamente armazenando o objeto (dono). Caso a posição exista, um identificador do recurso dono do dado requisitado é enviado e a partir daí é possível enviar uma mensagem com o pedido dos dados. Note-se que, uma vez terminada a operação, o requerente passa a ser o novo repositório do item requerido.

4.1.4 Modelo de Consistência e Operações

Na implementação de memórias compartilhadas distribuídas um ponto muito importante é a consistência. Essa determina como e quando mudanças feitas por um componente são vistas pelos outros componentes do sistema. A implementação do modelo de consistência mais estrito [Li e Hudak, 1989] exige a utilização de barreiras ou o assinalamento estático dos itens de dados. Uma vez que a primeira condição levaria a uma perda de desempenho na maioria dos casos e a segunda opção a vai de encontro ao cenário em questão, descartamos esse modelo de consistência.

Uma opção mais relaxada que o modelo de consistência estrito é chamada de consistência seqüencial [Lamport, 1979]. A única restrição desse modelo é que todos os os componentes do sistema observem a mesma seqüência de leituras e escritas. Nossa experiência no desenvolvimento de aplicações Anthill mostra que as operações que envolvem as variáveis do estado são em sua grande maioria associativas e comutativas [da Mata et al., 2009]. Nesse classe de operações, não importa a ordem estrita em que as sucessivas execuções acontecem, o que faz o modelo consistência seqüencial relaxado o suficiente para atender aos requisitos das aplicacões Anthil.

Uma vez que o repositório de metadados responde a todas as requisições remotas, a simples a serialização destas operações nos leva a implementação do modelo de consistência seqüencial no Estado Global Distribuído. Ou seja, uma próxima requisição só é atendida quando o RM recebe a confirmação dos demais envolvidos na requisição em execução: o requisitante e o "dono" do dado. Para esclarecer melhor a forma como consistência seqüencial é atingida, nas próximas seções descreveremos com mais detalhes as principais operações do EGD.

4.1.4.1 nget

Em implementações Anthill como a do algoritmo *K-Nearest Neighbors*[Fireman et al., 2008] o estágio mais intensivo em computação não realiza manipulações no estado. Ele é composto de uma série de leituras a base de dados e o processamento segue baseado nessas leituras. A possibilidade de utilizar mais recursos em tempo de execução, pode levar os desenvolvedores à utilizar o módulo EGD para armazenar essa base de dados.

Para aplicações que não alteram o valor do estado, foi desenvolvido a operação *nonblocking get* (nget). Um diagrama de seqüência desta operação foi mostrado na Figura 4.4 e um exemplo mais complexo é apresentado na Figura 4.5. Nela são mostradas as linhas de tempo das instâncias I_{01} e I_{02} e visa ilustrar não somente o funcionamento dessa operação mas também como a consistência é atingida nesse caso. No início das linhas de tempo a instância I_{01} já é o repositório do dado.

No instante i_1, a cópia I_{02} envia um pedido do tipo whoOwns para o RM, iniciando

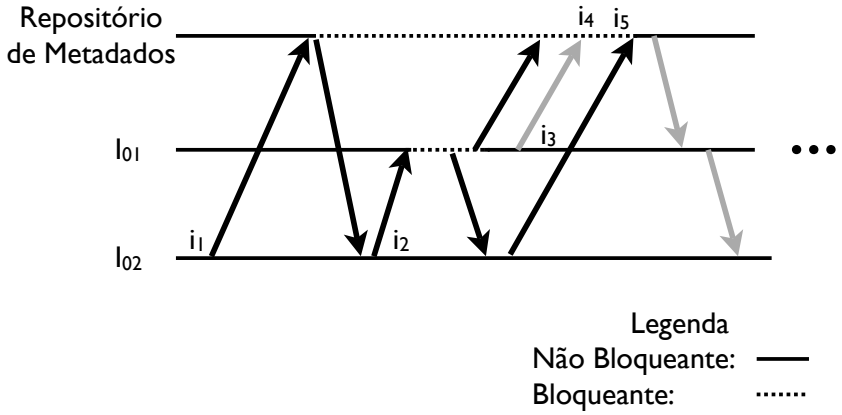


Figura 4.5. Exemplo de arranjo armazenado no EGD.

a operação de migração do dado (para detalhes dessa operação ver Figura 4.4). Este pedido marca o início de uma área bloqueante no gerente e as requisições que chegarem durante a realização desta operação vão para uma fila de espera. A resposta indica que I_{01} é o dono do dado e no instante i_2 a instância I_{02} envia para a instância I_{01} o pedido do dado. I_{01} envia o dado e logo depois a mensagem complete para o repositório de metadados. O envio dessa mensagem marca o fim de um operação bloqueante em I_{01}, que teve início no recebimento da mensagem de pedido de dado. Para evitar inconsistências, durante esse período, as instâncias enfileiram todas as requisições locais e remotas ao EGD.

Supondo que a cópia I_{01} processa uma requisição ao dado e no instante i_3 e envia o pedido ao RM, que chega no instante i_4. Uma vez que a mensagem do tipo complete, vinda de I_{02} ainda não chegou a operação ainda não foi terminada, logo o pedido originado de I_{01} será enfileirado. No instante i_5 a mensagem *complete* de I_{02} chega no repositório de metadados, liberando o mesmo para o processamento das demais requisições.

4.1.4.2 get e put

Uma vez que a adição dinâmica de instâncias ocorre de forma assíncrona, no caso em que o acesso aos dados (leituras) compartilhados é seguido de alterações nos mesmos, a consistência precisa ser mantida entre essas duas operações. A Figura 4.6 ilustra uma situação em que esse problema ocorre. A semântica dos retângulos da linha de execução permanece a mesma da Figura 4.4 e os retângulos de linha tracejada representam estágios não bloqueantes da execução.

As entidades componentes do diagrama são: i) Inst. A e B, que representam as linhas de execução de duas instâncias de um mesmo filtro; ii) EGD Inst. A e EGD

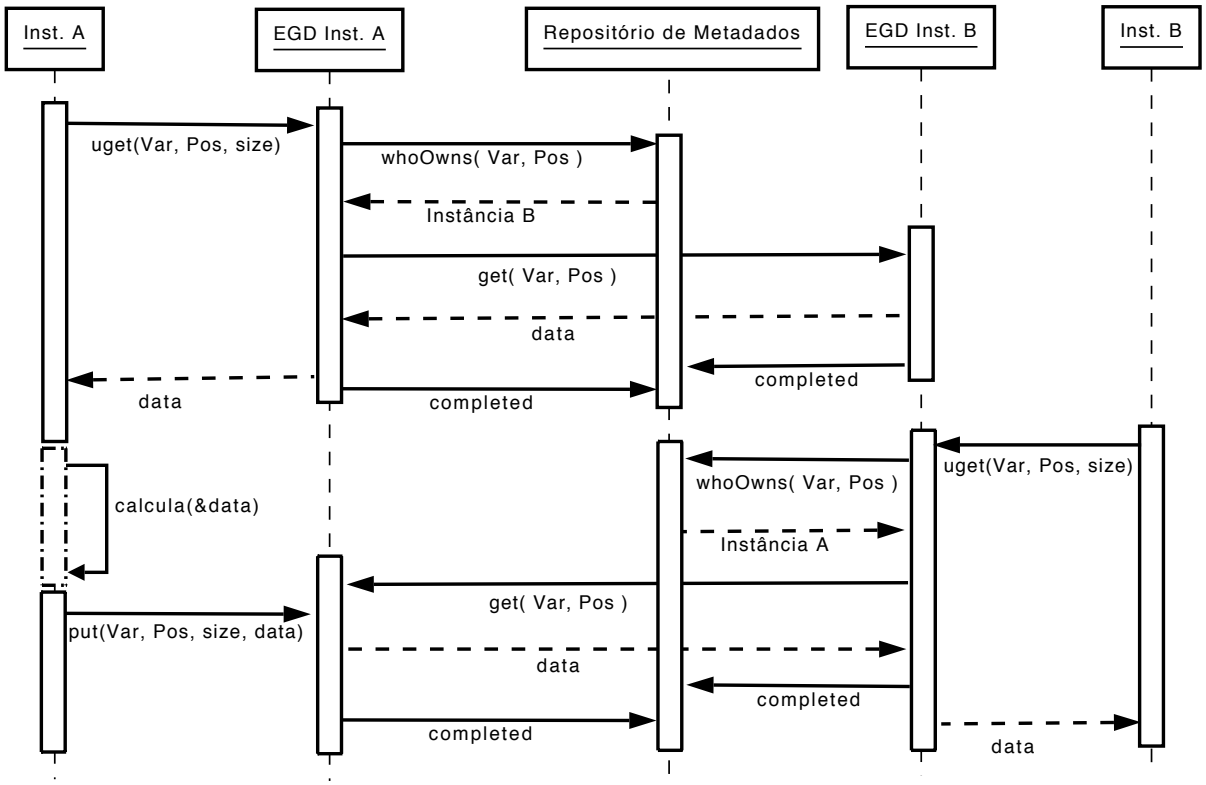


Figura 4.6. Problema de consistência: assincronia x alterações no estado.

Inst. B, que representam as linhas de execução (*threads*) dos componentes do EGD anexados a essas instâncias; por fim, temos iii) o Repositório de Metadados.

O exemplo tem início com uma chamada nget, efetuada pela instância A. Notemos que as linhas de execução Inst. A e EGD Inst. A ficam bloqueadas até o recebimento do dado e posterior carregamento na memória local do nodo de computação onde a instância A está executando. Seguindo o fluxo de execução, instância A realiza um processamento, altera e atualiza o estado através da chamada put. Enquanto esse processamento era realizado, a instância B inicia uma operação nget, acessando o mesmo item do estado. Uma vez que a instância estava numa etapa não bloqueante, a execução possui uma condição de corrida.

Ao receber a mensagem get(Var,Pos) a linha de execução EGD Inst. A bloqueia para evitar inconsistências nas estruturas de dados locais. Dessa forma, o processamento da chamada put só vai ocorrer após o desbloqueio da linha de execução EGD Inst. A. A partir desse ponto, o estado pode vir ficar inconsistente, uma vez que será atualizado com o resultado de um processamento de uma instância sem levar em consideração um processamento que possa vir a ser realizado pela outra.

Para solucionar esse problema, foi adicionado a API do EGD a operação get. Nessa operação, a linha de execução da instância fica bloqueada apenas até a chegada do

dado, sendo liberada para realização do processamento do mesmo. Enquanto isso, a linha de execução do EGD anexada a instância (e.g. Figura 4.6, entidade EGD Inst. A) permanece bloqueada. Dessa forma, qualquer mensagem de pedido de dado compartilhado originado por outras instâncias será enfileirada. O desbloqueio da linha de execução do EGD anexada a instância acontece com a atualização consistente do item do estado, realizada com a operação put.

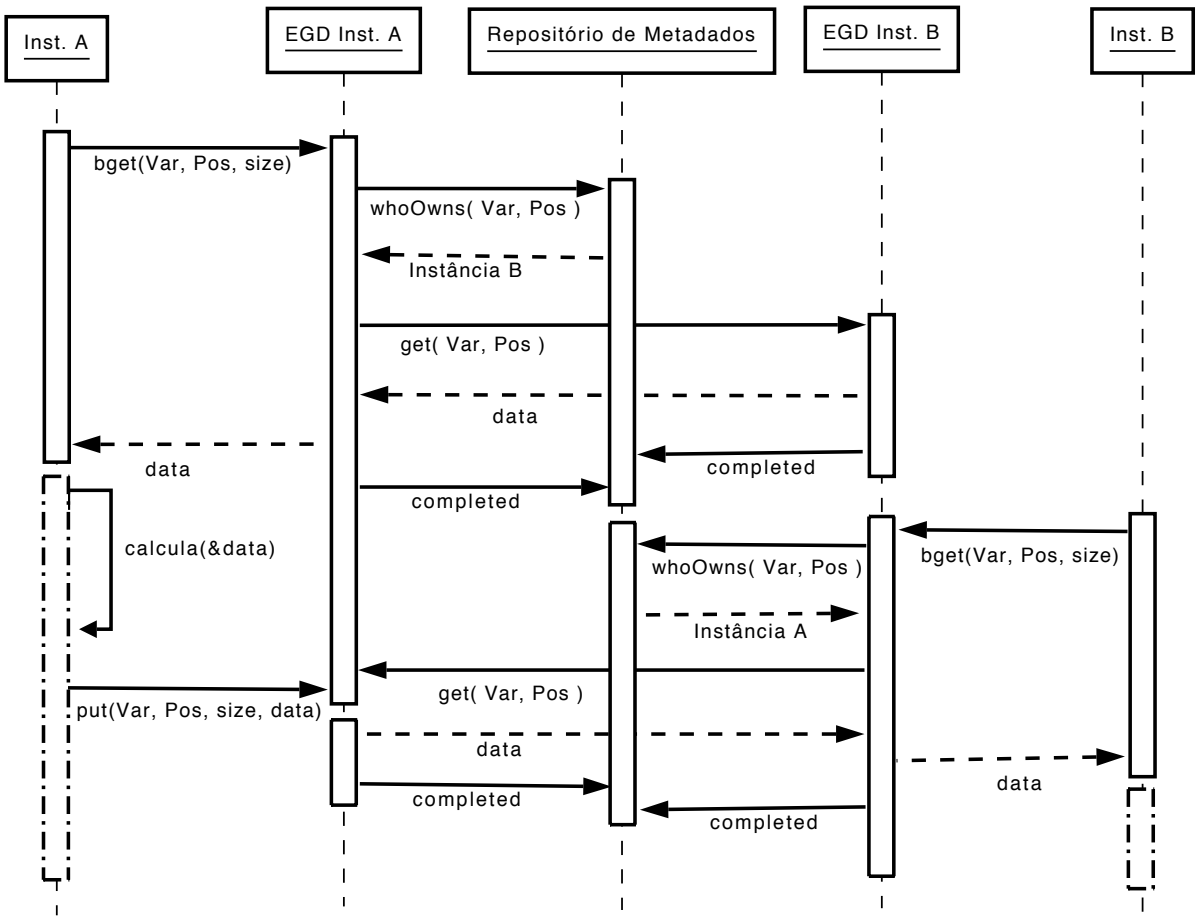


Figura 4.7. Diagrama de seqüência da operação *get*.

A Figura 4.7 ilustra o comportamento dessa solução. O diagrama possui os mesmos componentes do ilustrado na Figura 4.6 e a semântica dos retângulos da linha de execução também permanece a mesma. Nota-se que a resposta a mensagem get(Var, Pos) enviada por EGD Inst. B só ocorre depois do processamento da operação put, a qual desbloqueia EGD Inst. A para processamento das próximas mensagens. Após receber data, a instância B entra numa área assíncrona, permitindo o processamento do dado recebido. Já a linha de execução do EGD anexada a essa instância permanece bloqueada aguardando o put seguinte.

4.2 Implementação e Interface de Programação

Implementado de forma desacoplada dos demais módulos do Anthill, o EGD é baseado no conceito de *middleware*. Assim, não se faz necessário nenhum suporte adicional de hardware e o compartilhamento de dados é implementado baseado na comunicação entre componentes de uma camada que executa no nível de usuário (*daemons*). O acesso às variáveis do estado é realizado através de chamadas a essa camada e seus componentes são anexados transparentemente às instâncias de filtros, o que implica que toda cópia transparente está apta a armazenar variáveis do estado distribuído. Essa decisão de projeto foi tomada visando facilitar a utilização deste módulo em próximas versões do Anthill. A Figura 4.2 ilustra essa estrutura onde é possível notar a separação entre a implementação da interface de programação filtro-fluxo (FF API) e a do estado global distribuído.

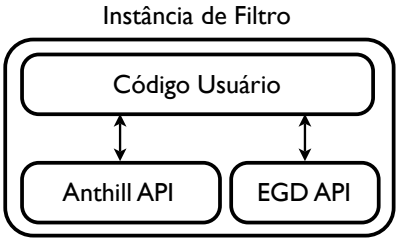


Figura 4.8. Estrutura de uma instância de filtro.

A tarefa de manutenção do estado compartilhado pelas instâncias de filtro é facilitada pelo EGD, que esconde do programador os detalhes de localização e transferência dos dados através de uma interface no estilo malloc/free. A Figura 4.9 contém um trecho da função de tratamento de mensagens do filtro, que ilustra a manipulação de estado no Anthill. Na linha 1, o índice do dado é extraído e então é utilizado na linha abaixo, onde a camada de memória compartilhada distribuída cuida dos detalhes de migração do dado, caso se faça necessário. Por fim, na linha 4, o resultado das manipulações sobre o dado é armazenado no espaço compartilhado e ficará disponível para acesso a partir de qualquer instância.

```
1 extractIndex(&msg, &index) ;
2 egd_get(VARIABLE, index dataSize, &data) ;
3 doCalculus(&data) ;
4 egd_put(VARIABLE, index, dataSize, &data) ;
```

Figura 4.9. Trecho de código da utilização do EGD em aplicações Anthill

Durante a execução da linha 2 pode ocorrer troca de mensagens pela rede ou apenas acesso à memória local, dependendo do local onde o dado requerido esteja armazenado. Como exposto na Seção 4.1.3, caso o dado identificado por *index* já esteja na memória local do nodo que está invocando a função egd_get, somente um acesso à memória local é realizado. Caso contrário, o fluxo mostrado na Figura 4.4 é executado. Caso contrário, execução da linha 4 não envolve nenhuma operação remota. Uma vez que após a execução da função egd_get o repositório de metadados estará devidamente atualizado e que qualquer acesso remoto a esse par (variável, índice) será redirecionado ao atual dono da posição, apenas uma atualização do estado local do dado se faz necessária, assim, a linha 4 nunca resultará num acesso remoto.

Para facilitar ainda mais implementação de aplicações que precisam em realizar iterações sobre dados locais (i.e. varredura de bases de dados), foi adicionado um iterador à interface de programação do EGD. O mesmo itera sobre a partição local do estado global distribuído. A Figura 4.10 mostra um trecho de código que utiliza essa funcionalidade. Ela é composta de três métodos: i) edg_first, que reinicia o iterador, fazendo ele apontar para a primeira posição, ou seja, o registro de menor índice; ii) edg_hasNext, que verifica se existem mais registros locais a serem visitados e iii) edg_next, que retorna o próximo objeto de dados armazenado localmente.

```
1 edg_first(VARIABLE) ;
2 while egd_hasNext(VARIABLE) do
3     edg_next(VARIABLE, dataSize, &data) ;
4     doCalculus(&data) ;
5 end
```

Figura 4.10. Exemplo de utilização das funções de iteração local

4.3 Resumo

O problema de particionamento do estado em aplicações Anthill pode ser resolvido de forma eficiente e compatível com o modelo filtro fluxo pelo mecanismo de fluxo rotulado. Com a adição do suporte à adição dinâmica de recursos, a localização e/ou número de instâncias de filtro pode mudar, causando uma mudança nos assinalamentos do fluxo rotulado e a necessidade de migração de dados entre as instâncias. Para solucionar este problema foi adicionado um novo módulo ao Anthill chamado de Estado Global Distribuído (EGD), que foi descrito nesse capítulo. Esse módulo é baseado no funcionamento das memórias compartilhadas distribuídas e abstrai o estado distribuído entre as instâncias de filtro como um espaço de endereçamento único, provendo migra-

ção transparente de dados. Por fim, ele é otimizado para lidar com a alta localidade de referência provida pelos assinalamento do fluxo rotulado.

Com o suporte à adição dinâmica de instâncias e de um mecanismo que permite a migração transparente de partições do estado dos filtros, uma pergunta precisa ser feita: o que aconteceria com o assinalamento de chaves no Anthill caso o conjunto de instâncias destino mudasse? No próximo capítulo mostramos que a função de dispersão padrão utilizada pelo mecanismo de fluxo rotulado é ineficiente nesse cenário e apresentamos uma solução para esse problema. Essa solução é baseada no protocolo de *hash* consistente e tem como principal objetivo minimizar o número de reassinalamentos causados pela reconfiguração. Será apresentada uma discussão informal sobre as características de distribuição consistente atingidas com a utilização dessa função e mostradas as modificações apropriadas no mecanismo de fluxo rotulado para utilização dessa função.

Capítulo 5

Particionamento Consistente do Estado de Aplicações Anthill

A forma utilizada para dividir o programa em tarefas bem definidas torna o modelo filtro-fluxo simples e intuitivo, permitindo que faça uso quase transparente de diversos níveis de paralelismo. Um problema ocorre quando as aplicações executam tarefas que necessitem manter algum tipo de estado. Com a adição do suporte à adição dinâmica de instâncias de filtros, uma pergunta tem que ser feita: o que aconteceria com o assinalamento de chaves no Anthill caso o conjunto de instâncias destino mudasse? Como a distribuição acontece usando a função dispersão baseada no resto da divisão inteira (*mod*), o resultado da adição de instâncias seria a movimentação de praticamente todos os itens que compõem o espaço. Nesse capítulo apresentamos mais detalhes sobre o problema, bem como propomos uma solução.

O restante do capítulo é organizado da seguinte forma: a Seção 5.1 analisa duas estratégias tradicionais para particionamento do estado (incluindo a estratégia padrão utilizada no Anthill) e mostra os problemas que surgem quando recursos são adicionados e o estado precisa ser redistribuído. Em seguida, a Seção 5.2 introduz o protocolo *hash* consistente, base da solução proposta nesse trabalho. A Seção 5.3 apresenta a nova função de dispersão a ser utilizada pelo Anthill e a Seção 5.4 mostra como essa função de dispersão é utilizada no mecanismo de fluxo rotulado, tornando esse mecanismo consistente. A Seção 5.5 mostra maiores detalhes sobre a implementação dessa solução e uma discussão informal sobre as características de distribuição consistente atingida é apresentada na Seção 5.6.

5.1 Funções de Dispersão Tradicionais: *Mod* e *Div*

Como discutido anteriormente, uma solução simples para a distribuição consistente e balanceada de objetos, tarefas ou requisições é a utilização de uma função de dispersão tradicional (função *hash*). Essa função assinala de forma determinística a chave de identificação de um objeto ao intervalo $[0..n-1]$, utilizado para identificar os n recipientes disponíveis num determinado instante do tempo. Essa estratégia é utilizada no Anthill, o qual utiliza a função resto da divisão inteira (*mod*), $h(x) = x \ mod \ n$, onde x é um número inteiro fornecido pelo usuário, utilizado pelo ambiente como chave de identificação de um objeto.

Dentre as qualidades desta função podemos citar o baixo custo, $\mathcal{O}(1)$, a simplicidade de implementação e o alto grau de balanceamento na distribuição. Entretanto, Silva et al. [2005] apontam graves problemas no assinalamento de chaves em aplicações onde o número de recipientes varia durante execução. Neste trabalho focamos num subconjunto deste problema: o suporte à adição dinâmica de instâncias à plataforma de execução. Essa situação é ilustrada pela Figura 5.1[1] onde é mostrado o resultado do assinalamento de um número fixo de chaves em dois momentos: i) para um conjunto de n recipientes e ii) para um conjunto de $n+1$ recipientes. Uma vez que caixas representam as chaves e os tons de cinza identificam os recipientes as quais as mesmas foram assinaladas, é possível notar que, após a adição de um recipiente, apenas uma pequena fração do espaço de chaves tiveram seus assinalamentos mantidos.

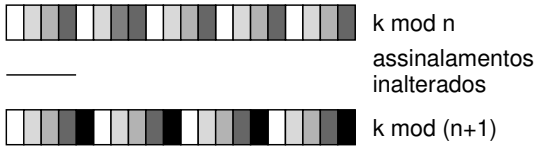


Figura 5.1. Redistribuição do assinalamento baseado na função *mod*.

Voltando ao Anthill, o cenário ilustrado na Figura 5.1 pode ser interpretado como resultado da adição, em tempo de execução, de uma instância de um filtro que possui estado. Imaginando a utilização do fluxo-rotulado padrão (*mod*), uma grande quantidade de itens do estado migraria, podendo gerar um grande tráfego de mensagens na rede, o que levaria à uma degradação no desempenho da aplicação. Pode-se mostrar que, utilizando essa função de dispersão, das k/n chaves armazenadas em cada nó antes da reconfiguração, apenas $k/n(n+1)$ têm seus assinalamentos mantidos [Silva et al., 2005]. Uma análise complementar mostra que, depois da reconfiguração, $k/(n+1)$ chaves são movidas a partir de cada nó, levando a um total de $nk/(n+1)$ movimentações.

[1] Extraída de [Silva et al., 2005], com permissão do autor

Outra função de dispersão tradicional que poderia ser utilizada para resolução de problemas como esse é a função de divisão inteira (*div*). A Figura 5.2[1] ilustra o comportamento da função $h(x) = x \; div \; n$ num cenário análogo ao caso anterior.

Figura 5.2. Redistribuição do assinalamento baseado na função *div*.

Como pode ser observado na figura, a utilização desta função apresenta um padrão mais regular de reassinalamento, onde as chaves são migradas apenas para os nós vizinhos, o que difere do assinalamento baseado em *mod*. Continuando as observações nota-se que um número menor de chaves são reassinaladas após a adição de um recipiente. Entretanto, uma análise mais cuidadosa mostra que o tráfego continua intenso a cada reconfiguração. Silva et al. [2005] mostram que $(b+1)k/(n(n+1))$ chaves são migradas de cada recipiente, onde b é a identificação de um recipiente (variando de 0 a $n-1$). Ou seja, metade das chaves são transferidas entre os nós cada vez que um recipiente é adicionado.

5.2 Hash Consistente

Na seção anterior mostramos que a utilização de estratégias de assinalamento baseadas em funções de dispersão tradicionais *div* e *mod*, apesar de muito eficientes para realização do mapeamento, geram um alto tráfego quando expostas à adição dinâmica de recursos. Visando amenizar esse problema, Karger et al. [1997] propuseram o protocolo de *hash* consistente. Em linhas gerais, os autores definem que uma função de dispersão é consistente quando, com alta probabilidade, (i) o sistema permanece balanceado e (ii) somente o número mínimo de chaves necessário para a manutenção do balanceamento é migrado na ocorrência de reconfigurações. As premissas têm que se manter válidas não só considerando uma configuração (ou visão), mas também entre várias configurações.

A idéia básica para atingir esse comportamento é utilizar um mesmo espaço de endereçamento para os identificadores de chaves e os recipientes aos quais esses identificadores serão associados. Assim, cada recipiente recebe um identificador único gerado aleatoriamente dentro do espaço de endereçamento e o assinalamento se torna uma função da posição relativa das chaves aos identificadores dos recipientes. Por exemplo, uma determinada chave pode ser assinalada ao recipiente mais próximo no intervalo. Desta forma, no momento em que um novo recipiente é adicionado, um identificador aleatório

é associado a ele e somente as chaves mais próximas serão reassinaladas, não havendo migração entre recipientes que já faziam parte do conjunto. Por fim, os autores mostram que para se atingir a consistência com alta probabilidade, a cada recipiente deve ser associado um número V de identificadores únicos gerados aleatoriamente dentro do espaço, onde V é da ordem do logaritmo do tamanho do espaço de endereçamento.

Outra característica a ser analisada é a complexidade da tarefa de assinalamento. Supondo que os recipientes dentro de uma mesma visão conheçam uns aos outros, pode-se definir um tabela de intervalos representando cada subdivisão do espaço de endereçamento. Essa tabela pode ser implementada, por exemplo, como uma árvore balanceada. Ao utilizar essa estrutura, o tempo necessário para determinar o assinalamento de uma chave será $\mathcal{O}(\log n)$, onde n é o número de recipientes. Esse desempenho é pior que nas alternativas baseadas em *hash* tradicional, onde essa mesma operação é realizada em tempo $\mathcal{O}(1)$.

O comportamento da redistribuição das chaves considerando-se o protocolo de *hash* consistente é ilustrado na Figura 5.3[1]. Para simplificar a ilustração os autores utilizaram um espaço linear, porém é mais comum a utilização de espaços de endereçamento circulares. Inicialmente temos quatro recipientes e cada um possui identificadores associados. Desta forma, o espaço de endereçamento é dividido em quatro regiões, baseado numa função qualquer de proximidade. Num segundo momento, um quinto recipiente é adicionado e um novo identificador é adicionado ao espaço.

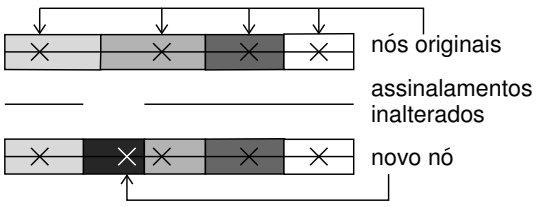


Figura 5.3. Redistribuição do assinalamento baseado em hash consistente.

Pode-se notar que a quantidade de chaves reassinaladas, ou seja, o volume de tráfego causado por essa adição, limita-se exclusivamente às chaves que precisam ser transferidas para o novo nó. Com alta probabilidade, chaves não são trocadas entre os nós previamente existentes e o custo mínimo para adição é atingido.

5.3 Construção da Função de Dispersão Consistente

Esta seção mostra a construção da função de dispersão (*hash*) consistente utilizada no Anthill. A notação e modelagem apresentados serão baseados no modelo proposto

[1]Extraída de [Silva et al., 2005], com permissão do autor

na Seção 2.5 e na extensão desse, proposta na Seção 3.3. Em seguida faremos uma caracterização baseada nas propriedades descritas por Karger et al. [1997].

No contexto das aplicações Anthill, definimos função *hash* variável como uma função da forma:

$$f \ : \ \mathcal{B} \ \times \ E \mapsto \mathcal{H}, \tag{5.1}$$

$$f\left(\beta, e\right) \in \beta, \ \forall(\beta \in \mathcal{B}) \wedge \forall(e \in E) \tag{5.2}$$

Onde f é uma função de dispersão variável, β uma configuração de aplicação, e um item e I um instância de filtro. A partir desse ponto usaremos $f_\beta(e)$ ao invés de $f\left(\beta, i\right)$.

Sejam duas funções aleatórias r_E e r_β, a idéia da construção é que ambos, itens e instâncias-destino, sejam mapeados no mesmo intervalo "global". Assim, é definido que a função r_E mapeia um item e aleatoriamente num intervalo discreto R, e a função r_β realiza o mesmo assinalamento aleatório com instâncias:

$$r_E : e \mapsto R$$
$$r_\beta : I \mapsto R$$

A função $f_\beta(e)$ assinalará e para o recipiente em β que minimizar

$$|r_\beta(I) - r_E(e)|$$

Uma vez que as definições são baseada em funções aleatórias, pode existir um desbalanceamento na distribuição causado pela variância nas associações. Para solucionar esse problema, Karger et al. [1997] argumentam que o problema pode ser solucionado com criação de $k \log C$ identificadores, para uma constante k, e assumindo que o número máximo de instâncias de filtro no intervalo seja sempre menor que C.

5.4 Fluxo Rotulado Consistente

Para facilitar o entendimento de como o mecanismo de fluxo rotulado se comportará na presença da adição dinâmica de instâncias de filtros, vamos considerar a aplicação-exemplo Conta Palavras, apresentada na Seção 2.4. Uma função de extração de rótulo simples e eficiente para esse problema pode extrair o código do caractere correspondente à primeira letra de cada palavra e retorná-lo, cujo exemplo de código é mostrado na Figura 5.4.

```
1 void extraiRotulo(void *msg, int size, char rotulo[ ] ){
2     memcpy(rotulo,msg,1);
3 }
```

Figura 5.4. Exemplo de função para extração de rótulo.

Este rótulo é então parâmetro de uma função *hash*. A função padrão de assinalamento de mensagens via fluxo rotulado no Anthill é descrita pela equação 2.3:

$$fluxo\_rotulado(msg, I_j) = \{I_{ik} \mid k = extraiRotulo(msg) \bmod |I_j|\}$$

Porém, realizar o assinalamento em ambientes dinâmicos utilizando a função de módulo pode levar a uma perda de desempenho, o que foi discutido na Seção 5.1. Assim, uma nova política de roteamento foi adicionada ao ambiente. Essa é baseada na definição da função de fluxo rotulado e realiza o assinalamento utilizando a função f definida na seção anterior:

$$fluxo\_rotulado\_chash(msg, I_j) = \{I_{ik} \mid k = f(extraiRotulo(msg))\} \quad (5.3)$$

5.5 Implementação e Interface de Programação

Com o objetivo de manter desacoplados o protocolo de assinalamento consistente e alguma funcionalidade específica do Anthill, foi implementada uma biblioteca que exporta as funções necessárias para utilização do protocolo de assinalamento segundo a descrição acima. Tal decisão foi tomada pois essa é a forma padrão de lidar com o problema de consistência, podendo ser utilizada tanto pela nova versão proposta de fluxo rotulado quanto pelo sistema de gerenciamento dinâmico do estado.

As funções r_E e r_β foram implementadas utilizando gerador de números pseudo-aleatórios chamado *Mersenne Twister*, definido por Matsumoto e Nishimura [1998]. Esse algoritmo foi escolhido pois possui características muito desejadas: (i) Foi provado que o mesmo possui um período de 2^{19937}, existe uma correlação serial desprezível entre sucessivos valores gerados e foi exposto com sucesso a inúmeros testes estatísticos de aleatoriedade. A implementação utilizada produz de forma pseudo-aleatória números naturais no intervalo $[0, 2^{32})$. Assim, esse foi intervalo utilizado para endereçamento de chaves, ou seja, as contradomínio das funções r_E e r_β.

Assim, definiu-se $C = 2^{32} - 1$ como um limite superior para o número total de instâncias em todas as visões e $\log(2^{32} - 1)$ identificadores associados a cada instância de filtro. Essa implementação usa uma estrutura de dados de árvore balanceada para

armazenar o espaço onde as instâncias são mapeadas. O que implica que consultas são realizadas usando com custo assintótico de $\mathcal{O}(\log 2^{32})$ e alterações sobre no tamanho do conjunto são realizadas com custo $\mathcal{O}(\log^2 2^{32})$.

Visando simplicidade, reduzimos a interface de programação a uma classe, que por sua vez possui dois métodos principais:

- `void addBin( unsigned int binId )`

  Esse método adiciona um recipiente que, a partir desse momento, poderá ter itens associados a ele.

- `unsigned int hash( unsigned int itemId )`

  Retorna o recipiente associado a um item identificado por *itemId*.

5.6 Propriedades da Função de Dispersão Consistente

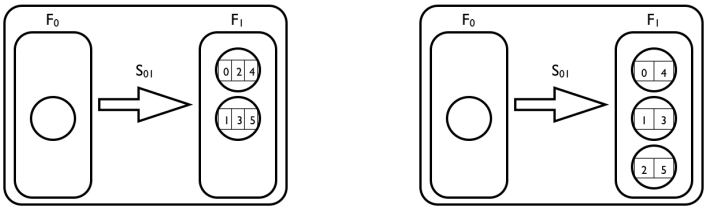
A utilização da função fluxo_rotulado_chash no roteamento de itens do estado tem como principal objetivo tornar consistente o roteamento realizado pela política de fluxo rotulado. Karger et al. [1997] definiram consistência utilizando 4 propriedades: i) balanceamento, ii) monotonicidade, iii) espalhamento e iv) carga. Apresentaremos a seguir uma discussão informal de como elas são atingidas, considerando o Anthill com suporte a adição dinâmica de instâncias:

5.6.1 Balanceamento

Uma função *hash* variável é dita balanceada, se

$$\forall(e \in E), \forall(\beta \in \mathcal{B}), \forall(I \in \beta), \exists(constante\, c) \quad | \quad P[f_\beta(e) = I] \leq \frac{c}{|\beta|}$$

Se uma função *hash* variável possui essa propriedade, a probabilidade de um determinado item e ser associado a uma instância em particular é baixa. Assumindo que as funções $r_\beta(I)$ e $r_E(e)$ se comportam de forma puramente aleatória para todas instâncias de filtro, a probabilidade de um item e em particular ser associado a uma instância I em particular é igual a probabilidade desse item ser associado a qualquer outra instância. Assim, pode ser dito que a função f é balanceada, o que significa que, para uma determinada configuração de aplicação k, $P[f_{\beta_k}(e) = I] \leq \frac{\mathcal{O}(1)}{|\beta_k|}$. A Figura 5.5 ilustra a ocorrência de reconfiguração onde a distribuição do espaço é feita por uma função balanceada.



(a) Configuração 0 (b) Configuração 1

Figura 5.5. Balanceamento.

5.6.2 Monotonicidade

Assumindo a seguinte situação: seja β_0 um conjunto de instâncias-destino e e um item assinalado por f a uma destas instâncias. Em algum momento da execução da aplicação ocorre um aumento no número destas instâncias e o conjunto resultante, β_1, é tal que $\beta_0 \subseteq \beta_1 \subseteq \mathcal{B}$. A Figura 5.6 ilustra a situação descrita.

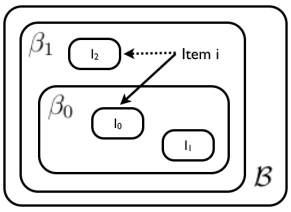


Figura 5.6. Monotonicidade.

Se f é monotônica, o item e deve ser reassinalado apenas à instância que foi adicionada, I_2. A monotonicidade garante que itens não serão redistribuídos desnecessariamente. Além disso, itens só devem ser movidos se for necessário para preservar a uniformidade da distribuição, não só em uma configuração, mas em todas as configurações.

Mais formalmente, uma função *hash* variável é dita monotônica, se:

$$\forall(\beta_m \subseteq \beta_n \subseteq \mathcal{B}) \quad | \quad f_{\beta_n}(e) \in \beta_m \;\Rightarrow\; f_{\beta_m}(e) = f_{\beta_n}(i)$$

Na função f, quando um novo recipiente é adicionado, ele é assinalado aleatoriamente a vários pontos no intervalo, fazendo com que apenas os itens que estão mais próximos a esses pontos serão assinalados e nenhum item será movido entre antigos recipientes. Concluímos assim que f é monotônica.

5.6.3 Espalhamento

O espalhamento $\mathcal{E}(e)$ de um item e, é a quantidade de instâncias diferentes em que e é assinalado em todas as configurações:

$$\mathcal{E}(e) = \left| \bigcup_{0 \leq k \leq |\mathcal{B}|} f_{\beta_k}(e) \right|$$

A Tabela 5.1 ilustra uma seqüência de configurações e o comportamento da função em cada caso. Nesse exemplo, como o item e foi assinalado a duas instâncias, temos que $\mathcal{E}(e) = 2$. Uma boa função de *hash* consistente deve ter um baixo espalhamento não apenas em um único item, mas em todos os itens.

$f_{\beta_0}(e) = I_0$	$\beta_0 = \{I_0, I_1\}$
$f_{\beta_1}(e) = I_2$	$\beta_1 = \{I_0, I_1, I_2\}$
$f_{\beta_2}(e) = I_2$	$\beta_2 = \{I_0, I_1, I_2, I_3\}$
$\mathcal{E}(e) = 2$	

Tabela 5.1. Seqüência de três reconfigurações mostrando o espalhamento de um item.

Assumindo que o número de configurações $|\mathcal{B}| = \rho C$, para alguma constante ρ e o número de itens $|E| = C$. Como estamos lidando com adição dinâmica de instâncias e o assinalamento é dado por uma função monotônica, $|\beta_0|$ instâncias serão compartilhadas entre todas as configurações. Dessa forma, com alta probabilidade, um ponto que está em todas as configurações estará dentro de um intervalo de tamanho $\mathcal{O}(\frac{|\beta_0|}{C})$. O número de identificadores de instâncias que cabem nesse intervalo pode ser considerado um limite superior para o espalhamento de item, uma vez que nenhum outro identificador de instância pode estar mais perto, em qualquer configuração.

Como o número de identificadores de instâncias é $\mathcal{O}(C \log C)$, em todas as visões, pode-se mostrar $\mathcal{E}(e) = \mathcal{O}(|\beta_0| \log C)$ como limite superior para o espalhamento de um item e qualquer, para todas as visões e com probabilidade maior que $1 - 1/C^{\Omega(1)}$. Por fim, uma vez que todo item estará associado a pelo menos uma instância temos $\mathcal{E}(e) = \Omega(1)$ (sempre ≥ 1) como limite inferior para espalhamento de um item.

5.6.4 Carga

A carga de uma função *hash* variável é a quantidade máxima de itens diferentes que estão associados a uma instância. Como exemplo, vamos supor as três configurações expostas na Tabela 5.2.

(a) Conf. 0		(b) Conf. 1		(c) Conf. 3	
I_0	$1, 3, 7, 5$	I_0	$1, 3$	I_0	1
I_1	2	I_1	$2, 5$	I_1	2
		I_2	7	I_2	$7, 5$
				I_3	3

Tabela 5.2. Seqüência de três reconfigurações mostrando a distribuição do estado.

Nesse caso, na primeira configuração, temos a instância I_0 assinalada a quatro itens e a instância I_1 a apenas um item. Analisando as demais configurações vemos que quatro é a maior quantidade de itens que a instância I_0 foi associada, assim $c(I_0) = 4$. No caso da instância I_1, houve uma variação entre as três configurações e o máximo ocorreu na segunda configuração, levando a uma carga $c(I_1) = 2$. A mesma análise é válida para as demais instâncias: $c(I_2) = 2$ e $c(I_3) = 1$.

Definindo mais formalmente a carga c em uma instância, temos:

$$c(I) = \left| \bigcup_{0 \leq j \leq |\mathcal{B}|} f_{\beta_j}^{-1}(I) \right|$$

onde $f_{\beta_j}^{-1}(b)$ é a função inversa de f e retorna o conjunto de itens que são associados a um determinado recipiente, numa determinada configuração. Uma boa função de *hash* consistente deve também possuir baixa carga

A demonstração informal da carga de f segue a mesma linha da demostração do espalhamento. Dessa forma, pode-se mostrar $c(I) = \mathcal{O}\left(|\beta_0| \log C\right)$ é o limite superior para o tamanho do intervalo ao qual uma instância I qualquer será "dona" com probabilidade maior que $1 - (C)^{-\Omega(1)}$. Esse intervalo é a contagem todos os pontos que estão mais perto de todos os identificadores desta instância I.

5.7 Resumo

Este capítulo apresentou uma discussão que mostra que a função de dispersão padrão utilizada pelo mecanismo de fluxo rotulado do Anthill é ineficiente no cenário em que instâncias são adicionadas em tempo de execução. Foi apresentada também uma solução para esse problema. Essa solução é baseada no protocolo de *hash* consistente e tem como principal objetivo minimizar o número de reassinalamentos causados pela reconfiguração. Foi apresentada uma discussão informal sobre as características de distribuição consistente atingidas com a utilização dessa função e mostradas as mo-

dificações apropriadas no mecanismo de fluxo rotulado para utilização dessa função. Por fim, foram mostrados os detalhes de implementação e a *interface* de programação exportada.

No próximo capítulo apresentaremos uma avaliação experimental realizada com as implementações propostas. Serão descritas as aplicações utilizadas e as configurações experimentais de *hardware* e *software*. Por fim, os resultados obtidos serão apresentados e discutidos.

Capítulo 6

Avaliação Experimental

Até aqui foram apresentados os componentes que fazem parte da solução de suporte para adição dinâmica de instâncias no Anthill. Este capítulo descreve a avaliação experimental realizada com as implementações propostas. O restante do capítulo possui a seguinte organização: a Seção 6.1 descreve as aplicações utilizadas nessa avaliação experimental e justifica a utilização das mesmas. Em seguida as configurações experimentais de *hardware* e *software* são descritas na Seção 6.2. Por fim, os resultados obtidos são apresentados e discutidos na Seção 6.3.

6.1 Aplicações

Esta seção descreve as quatro aplicações escolhidas para execução nessa avaliação experimental. Elas representam quatro grandes classes de aplicações Anthill e são apresentadas em ordem de complexidade: a Seção 6.1.1 descreve a aplicação Extração de Características de Células Neuroblásticas, representando as aplicações que não possuem estado. Em seguida, as Seções 6.1.2 e 6.1.3 descrevem as paralelizações filtro-fluxo de dois algoritmos de mineração de dados que visam analisar aplicações síncronas e assíncronas que realizam somente operações de leitura no estado. Por fim, a Seção 6.1.4 descreve a aplicação utilizada para análise de assíncronas que modificam o estado.

6.1.1 Extração de Características de Células Neuroblásticas (ECCN)

Nos últimos anos, vários trabalhos científicos vem estudando a utilização de sistemas computacionais de análise de imagens na área biomédica. Em particular, uma aplicação desses conceitos foi utilizada para auxiliar o prognóstico, diagnóstico e classificação de diferentes tipos de patologias [Kobatake et al., 1994; Nasser Esgiar et al., 1998;

Doyle et al., 2006]. Esses métodos geralmente envolvem o processamento de grandes bancos de dados de imagens de alta resolução, uma tarefa computacionalmente muito intensiva. Por outro lado, altas taxas de erros e/ou imprecisão normalmente não são aceitáveis, devido à natureza biomédica dessas aplicações. Assim, diversas propostas de paralelização dos algoritmos foram apresentadas, com o objetivo de executar os mesmos mantendo altas taxas de precisão e tempos de respostas aceitáveis.

Neste trabalho, utilizamos a implementação Anthill de um estágio do classificador para Tumores Neuroblásticos Periféricos (pNTs) do tipo Neuroblastoma, como aplicação alvo para avaliação do comportamento de aplicações sem estado na presença de reconfigurações. Essa escolha se deu pelos seguintes motivos: alta relevância da aplicação, alto custo computacional e pelo fato dos dados de entrada podem ser facilmente particionados.

Essa aplicação, anteriormente descrita por Kong et al. [2007], Gurcan et al. [2007] e Ruiz et al. [2007], recebe como entrada imagens em formato RGB devidamente colorizadas e compactadas. Essas são convertidas para o formato LA*B*, adotado por fornecer uma percepção uniforme das cores, o que permitiu a utilização de operadores lineares, tornando o o processo mais simples e eficiente. Esses operadores lineares são aplicados a cada canal L, A*, B* e são extraídas as características estatísticas contraste, correlação, homogeneidade e energia. Além dessas características, uma medida complementar de contraste local é extraída, o *Local Binary Pattern* (LBP). O resultado final é um vetor de 13 dimensões, pois temos 4 características para cada canal, mais o resultado do operador LBP. Na versão Anthill foi definido um filtro para a leitura da imagem, um filtro para a conversão das imagens e um filtro para o cálculo das estatísticas e composição do vetor resultado. Esses filtros são ligados por um fluxo que utiliza a política de roteamento *round robin*. A Figura 6.1 ilustra o funcionamento descrito e os retângulos pontilhados delimitam o comportamento dos filtros na implementação Anthill.

6.1.2 Classificação

Classificação é um processo de predição onde se deseja associar classes (ou categorias) à objetos desconhecidos (chamados de base de teste), usualmente, utilizando um modelo baseado em objetos pré-classificados (chamados de base de treinamento). Um algoritmo clássico para classificação de objetos é o K-Means [Pelleg e Moore, 1999], o qual baseia-se na premissa de que a vizinhança de um objeto contém objetos semelhantes. Desta forma, através da análise da vizinhança poderia ser estimada a classe de um objeto desconhecido. Na prática, vizinhança (proximidade) é um conceito inerente a cada domínio de aplicação. Assim, para fins de experimentação foi utiliza a função de

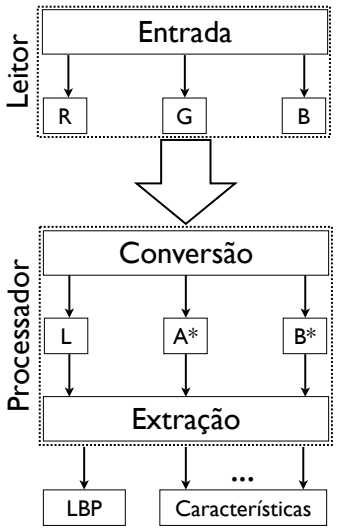


Figura 6.1. Arcabouço da aplicação para extração de características de células neuroblásticas

distância Euclidiana entre os atributos para determinação de proximidade entre os objetos.

Em linhas gerais o algoritmo pode ser descrito em quatro passos: (i) O algoritmo é iniciado com um número de pontos escolhidos aleatoriamente, chamados de centróides; (ii) visitam-se todos os pontos de um conjunto de dados de treinamento e, para cada ponto p, é encontrado o centróide c_i mais próximo de p e é feita uma associação entre p e o agrupamento do qual c é o centro; (iii) c_i é movido para na direção à p; (iv) Os passos ii e iii são repetidos até que a condição de precisão ou o número de iterações máximo seja atingido.

Uma versão do K-Means foi implementada no Anthill. O algoritmo foi modelado no paradigma filtro-fluxo usando os Assinalador e Agregador, e um filtro Final, como pode ser visto na Figura 6.2. O filtro Assinalador mantém partições da base de treinamento. Esse filtro determina quais objetos estão mais próximos de quais centróides, atualiza a posição de cada centróide e envia as novas posições dos centróides para o filtro Agregador. Esse, uma vez que possui uma visão agregada dos centróides locais, calcula os centróides globais e os envia para os Assinaladores. Quando um número máximo de iterações é atingido ou a melhora na precisão dos centróides globais atinge um certo limiar, os centróides globais são enviados para o filtro Final, que escreve o resultado num arquivo, encerrando a aplicação. Essa aplicação foi escolhida pois, mesmo possuindo um comportamento de fácil compreensão, é largamente utilizada na prática. Além disto, é caso onde ilustramos a existência de sincronia (barreiras).

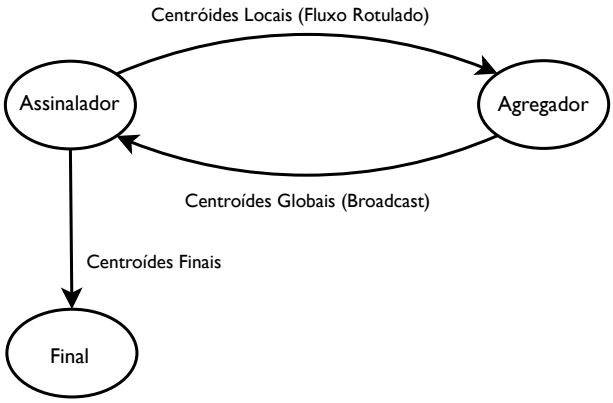


Figura 6.2. Filtros da Aplicação K-Means

6.1.3 Análise de Associações

Análise de associações consiste na identificação de relações de causalidade entre itens que freqüentemente co-ocorrem em uma base de dados. Essas relações recebem o nome de regras de associação e o primeiro passo da sua computação é a determinação de todos os conjuntos de itens que ocorrem na base com uma freqüência maior que um limite determinado pelo usuário. Autores como Agrawal e Srikant [1994]; Zaki et al. [1997], descrevem soluções para esse problema. Neste trabalho, usaremos o algoritmo Apriori [Agrawal e Srikant, 1994].

O Apriori é baseado no princípio de que, para um conjunto com k itens ser freqüente, todos seus subconjuntos de tamanho $k-1$ também devem ser. O algoritmo inicia a execução com um conjunto composto por todos os subconjuntos de itens de tamanho unitário e calcula a freqüência desses subconjuntos na base de dados. Os candidatos freqüentes de tamanho unitários são combinados e produzem novos candidatos de tamanho dois, que serão verificados na próxima iteração do algoritmo. Assim, os ciclos continuam, assinalando candidatos e verificando suas freqüências, assincronamente, até que não existam mais candidatos.

A paralelização do Apriori, conforme apresentada na Figura 6.3, é composta por três filtros: i) O filtro Contador é responsável por verificar a freqüência dos conjuntos de itens candidatos na base de dados. ii) O Verificador recebe via fluxo rotulado as contagens locais enviadas por cada instância do filtro Contador, reduzindo as mesmas em um valor global que é utilizado para verificar se o conjunto de itens é freqüente. O Gerador de candidatos, por sua vez, combina os elementos dos conjuntos freqüentes criando novos conjuntos candidatos. O algoritmo Apriori implementado no Anthill explora a assincronia inerente nesta aplicação, uma vez que que o Contador pode processar simultaneamente conjuntos candidatos oriundos de diversas iterações (de

diversos tamanhos).

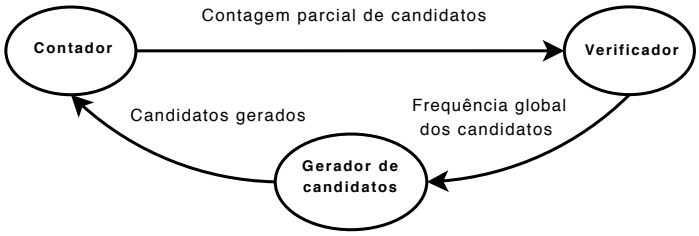


Figura 6.3. Paralelização Filtro-Fluxo do algoritmo Apriori

6.1.4 Emissor-Processador

Para avaliar o impacto que a consistência no assinalamento pode causar no desempenho de aplicações Anthill foi criada uma aplicação fictícia, a qual chamamos de Emissor-Processador. Essa aplicação é composta de dois filtros: o (i) Emissor, que envia um número pré-determinado de tarefas via fluxo rotulado e o (ii) Processador, que recebe essas tarefas, executa um processamento de tamanho fixo e responde avisando do término. A Figura 6.4 ilustra o projeto desta aplicação Anthill.

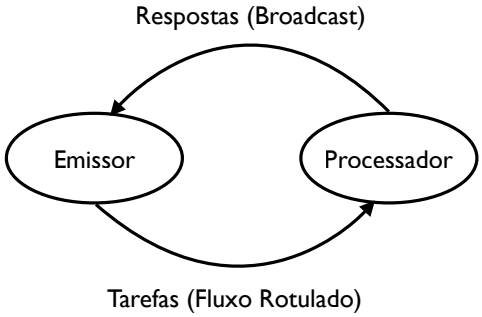


Figura 6.4. Paralelização Filtro-Fluxo da aplicação Emissor-Receptor

As mensagens enviadas pelo processador são prefixadas por números, os quais poderiam ser interpretados como as chaves primárias dos registros de um banco de dados. Esses identificadores são utilizados como rótulos no envio através do fluxo rotulado.

6.2 Configuração Experimental

As avaliações experimentais ocorreram em duas plataformas:

- Plataforma 1: composta por 8 computadores conectados por ligações Gigabit Ethernet e cada um equipado com um processador Core 2 Duo ™ 2.13 GHz, 4 MB de memória cache e 2GB de memória principal.

- Plataforma 2: composta por 16 PCs conectados via interfaces de rede Fast Ethernet. Cada componente da plataforma possui um processador AMD Athlon ™ 3200+, 2 MB de memória cache e 2 GB de memória principal.

Os experimentos para avaliação da aplicação de extração de características de células neuroblásticas foram executados na plataforma 1. Uma amostra de tecido neuroblástico foi utilizada como entrada e essa imagem foi dividida em 12.600 segmentos de mesmo tamanho e mesma resolução. Para avaliar o impacto da resolução dos segmentos no tempo de execução da aplicação, os experimentos foram divididos em dois cenários de acordo com a resolução processada: 256x256 e 512x512. Para simplificar a análise, o tempo para particionamento da imagem foi ignorado.

Os demais experimentos foram realizados na plataforma 2. Em termos de configuração de *software*, experimentos com o K-Means e o Apriori foram realizados utilizando bases de dados sintéticas geradas segundo o procedimento descrito por Veloso et al. [2004]. Ambas são compostas por 1.000.000 registros. No caso do K-Means, cada ponto possui 50 dimensões e essas podem representar, por exemplo, variáveis sócio-demográficas de regiões do brasil. Já para a aplicação de análise de associação, cada transação contém 70 atributos.

Para aplicação Emissor-Receptor realizamos testes do tipo *scaleup*. Nesses experimentos o tamanho da base de dados foi aumentado de forma diretamente proporcional ao número inicial de instâncias do experimento. A tabela 6.1 mostra as configurações utilizadas.

Tabela 6.1. Configuração dos experimentos da aplicação Emissor-Processador

#Instâncias	Tamanho da Base de Dados
1	683.6 MB
2	1.33 GB
4	2.67 GB
8	5.34 GB

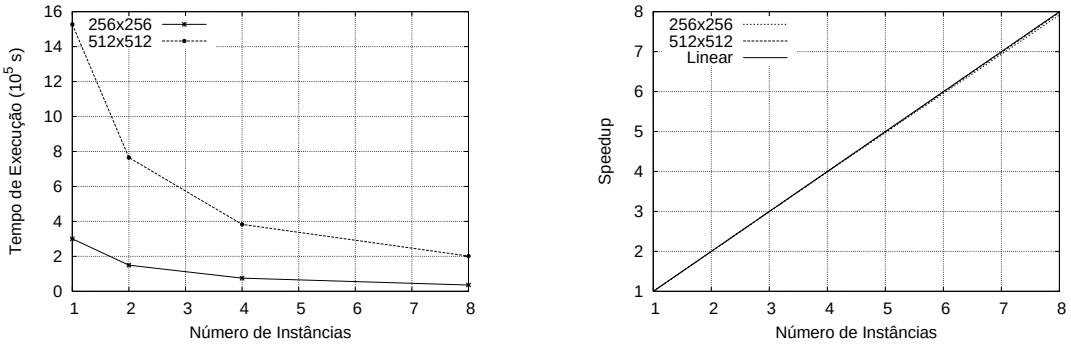
6.3 Resultados

6.3.1 Escalabilidade

Com o objetivo de caracterizar o potencial de paralelismo das implementações utilizadas nesta avaliação experimental e assim nortear as demais conclusões, realizamos uma série de experimentos estáticos. Por estático, entende-se que não houve mudanças de configuração em tempo de execução. Nestes experimentos verificamos os tempos de execução das aplicações à medida que mudamos o cenário da execução.

6.3.1.1 ECCN

Após a realização de uma análise inicial foi identificado que o filtro Processador é o mais custoso em termos computacionais, sendo o melhor candidato para ter o número de instâncias variado. A Figura 6.5(a) ilustra os tempos de execução nos cenários 256x256 e 512x512. Nela percebemos que existe uma queda no tempo de execução proporcional ao aumento do tamanho da plataforma. A Figura 6.5(b) confirma essa primeira através da quase linearidade nos *speedups*.



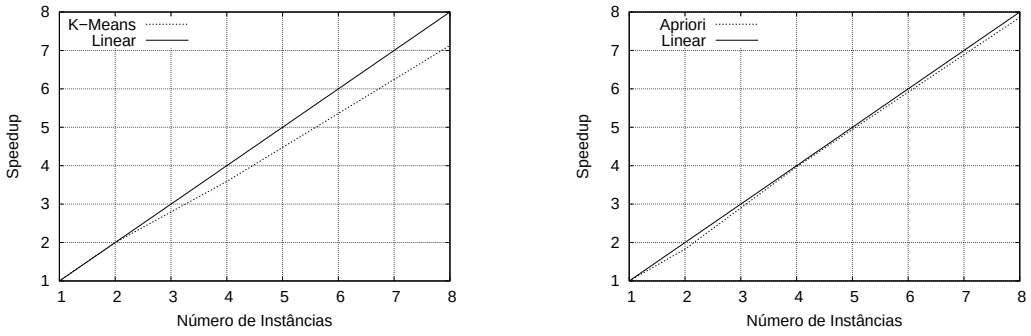
(a) Tempo de Execução (b) Speedup

Figura 6.5. Tempos de execução e speedup da aplicação ECCN

6.3.1.2 K-Means e Apriori

Medições preliminares foram efetuadas com o intuito de identificar os estágios mais intensivos em processamento, os quais comporiam um conjunto de candidatos a uma possível reconfiguração. Na aplicação K-Means identificou-se o filtro Assinalador e na aplicação Apriori o filtro Contador como os estágios mais custosos, sendo escolhidos para terem seu número de instâncias variado.

A Figura 6.6 mostra o *speedup* destas aplicações quando o número de instâncias cresce até 8. Nela podemos ver que ambas possuem boa escalabilidade, estando sempre muito próximas da escalabilidade linear.



(a) K-Means (b) Apriori

Figura 6.6. Speedups das aplicações K-Means e Apriori

6.3.1.3 Emissor-Processador

Os testes de escalabilidade da aplicação Emissor-Processador foram realizados variando o número de instâncias do filtro Processador. A Figura 6.7 mostra o comportamento do *scaleup* da aplicação para 1,2,4 e 8 instâncias. Nela podemos ver que a aplicação possui uma escalabilidade muito próxima da linear.

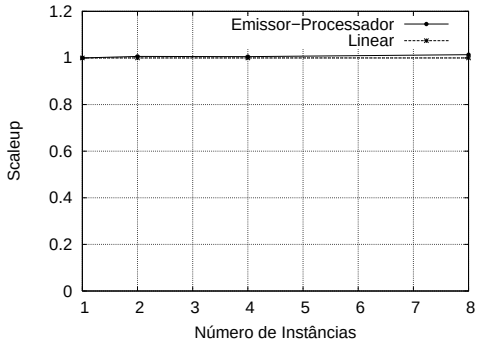


Figura 6.7. Scaleup da aplicação Emissor-Receptor

6.3.2 Adição Dinâmica de Instâncias

Para uma primeira análise da implementação do suporte à adição dinâmica de instâncias foram executados experimentos onde o número de instâncias dobra em pontos pré-determinados do progresso execução. Visando tornar a análise mais refinada, definimos três pontos: 25%, 50% e 75%. Por exemplo, envia-se 25% do trabalho total a ser realizado e depois que esse trabalho é processado ocorre adição das demais instâncias. Não há nenhuma razão específica para escolha da quantidade de instâncias adicionadas (o dobro) nem para escolha dos pontos de reconfiguração. Apenas que esses nos forneceriam uma idéia suficientemente ampla do impacto da reconfiguração.

6.3.2.1 Aplicações Sem Estado

Iniciamos a análise pelo caso mais simples: onde o sucesso da reconfiguração depende apenas da implementação do mecanismo de adição dinâmica de instâncias. Assim, a aplicação de extração de características de células neuroblásticas foi escolhida pois, além de ter um comportamento regular para cada tamanho de entrada, não possui nenhum dado que precisa ser armazenado entre execuções do filtro Processador. Em outras palavras, não possui estado. Dessa forma pretendemos isolar as análises, facilitando assim o entendimento.

A Figura 6.8 mostra a variação do tempo de execução para o processamento de segmentos de resolução 256x256. Nota-se que o tempo de execução decrescendo quase linearmente em função do número de instâncias, resultado que demonstra a efetividade da solução de adição dinâmica de instâncias para caso de aplicações sem estado. Os pontos marcados dentro das barras indicam o limiar teórico, calculado sem levar em consideração custos de reconfiguração.

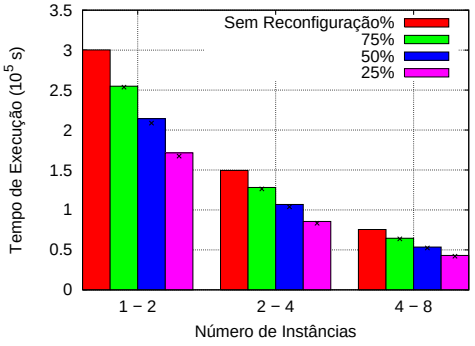
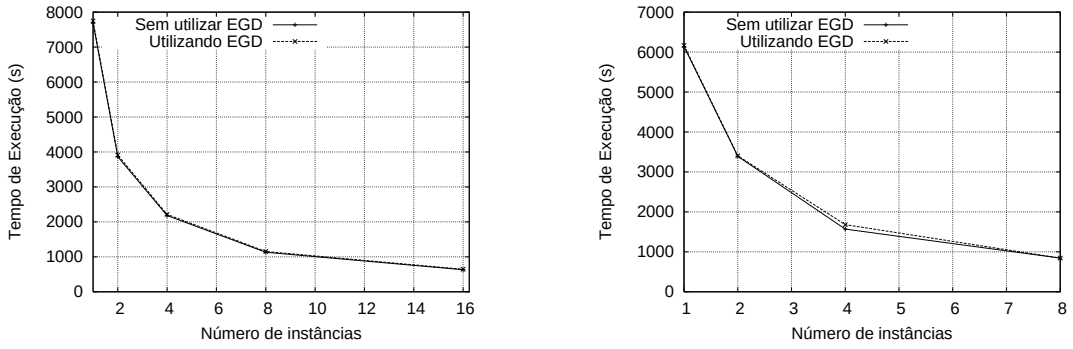


Figura 6.8. Tempos de execução da aplicação de ECCN em face a adição dinâmica de instâncias

6.3.2.2 Utilização do Estado Global Distribuído

Para uma segunda série de análises adicionamos a utilização do estado global distribuído. Para tal, uma versão reconfigurável das aplicações de K-Means e Apriori foi criada, onde os filtros mais caros continuam sendo os candidatos a reconfiguração. No KNN, o filtro Assinalador passou a armazenar todos os objetos da base no estado global distribuído, permitindo que a migração de dados decorrentes da reconfiguração ocorra de forma transparente. Na aplicação Apriori, mudamos o filtro Contador. Esse passou a armazenar a base de dados no estado global distribuído. Assim, para facilitar a utilização do EGD, a base de dados foi disposta horizontalmente, onde cada linha representa uma transação e, para cada uma delas, os itens que a compõe.

A partir desta versão, primeiramente tentamos isolar o custo de utilização EGD. Para tal, executamos um conjunto de experimentos sem adição de instâncias em tempo de execução. A Figura 6.9 mostra a diferença nos tempos de execução entre a versões as versões com e sem utilização do EGD. Como podemos ver, a perda de desempenho ocasionada pela utilização do estado global distribuído é muito pequena. Nossos resultados mostram que essa é de aproximadamente $1,5\%$ e $1,58\%$ em média, para as aplicações K-Means e Apriori, respectivamente.

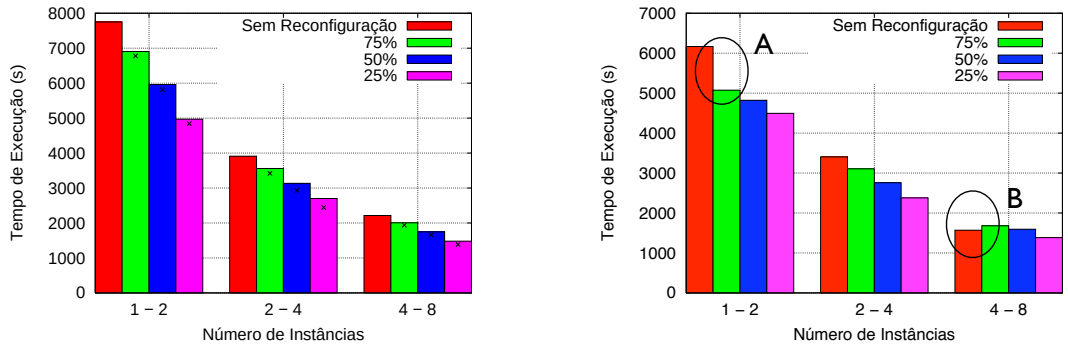


(a) K-Means (b) Apriori

Figura 6.9. Custo da utilização do EGD nas aplicações K-Means e Apriori

Outra série de experimentos foi efetuada visando avaliar o impacto da adição dinâmica de instâncias no desempenho da aplicação. Da mesma forma que na aplicação ECCN, foram executados experimentos onde o número de instâncias dobrava em três pontos pré-determinados da execução: 25%, 50% e 75%. Como pode ser notado na Figura 6.10(a), a aplicação K-Means apresentou um padrão muito regular de ganho. Diversos fatores contribuem para esse fato, entre eles a regularidade no padrão de execução da aplicação, ou seja, o tempo de execução das tarefas é proporcional ao tamanho

da entrada. Dessa forma, se depois da reconfiguração os Assinaladores têm que processar uma base de metade do tamanho, eles o farão em metade do tempo. Novamente, os pontos marcados dentro das barras indicam o limiar teórico, calculado sem levar em consideração custos de reconfiguração e migração do estado.



(a) K-Means (b) Apriori

Figura 6.10. Tempos de execução da aplicação K-Means e Apriori em face a adição dinâmica de instâncias

A aplicação Apriori, no geral, apresentou um padrão regular de ganho. Como pode ser notado na Figura 6.10(b), dois pontos chamam atenção neste gráfico: i) Um grande ganho no desempenho, ocorrido quando no cenário de reconfiguração de uma pra duas instâncias (marcado com A); e ii) uma perda de desempenho, ocorrida no cenário onde partimos de quatro para oito instâncias (marcado com B).

Uma vez que essa aplicação possui um padrão de execução irregular, ou seja, a demanda por processamento pode variar durante a execução, apenas medir o tempo de execução não é suficiente para explicar o comportamento da aplicação. Assim, realizou-se um novo conjunto de experimentos, onde foi medida a taxa com que novos conjuntos candidatos são verificados freqüentes e o resultado é mostrado na Figura 6.11. No cenário apresentado, o conjunto inicial de instância é de tamanho dois e o final de tamanho quatro, e as instâncias são adicionadas com 25% da execução completa. Sendo escolhido exatamente pelo comportamento menos irregular, levando em conta o gráfico da Figura 6.10(b).

Esse experimento mostra que a taxa de verificação cresce após a adição de novas instâncias e cai muito nos últimos 20% de execução. Desta forma, mesmo possuindo menos tarefas a executar, o custo com a migração de dados pode ser bem menor que a quantidade de trabalho a ser executada por apenas 2 instâncias. O que explica o comportamento mostrado no ponto A da Figura 6.10(b). A situação inversa ocorre quando já existem várias processadores dividindo essa carga. Neste cenário, o custo

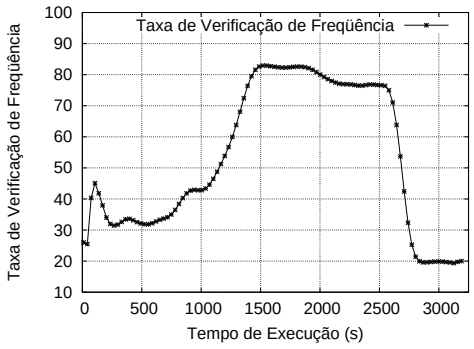


Figura 6.11. Taxa de verificação de freqüência de novos conjuntos candidatos

de migração dos dados, necessário para o rebalanceamento da carga, passa a ser maior que o benefício trazido pela adição das mesmas.

6.3.2.3 Assinalamento baseado em Hash Consistente

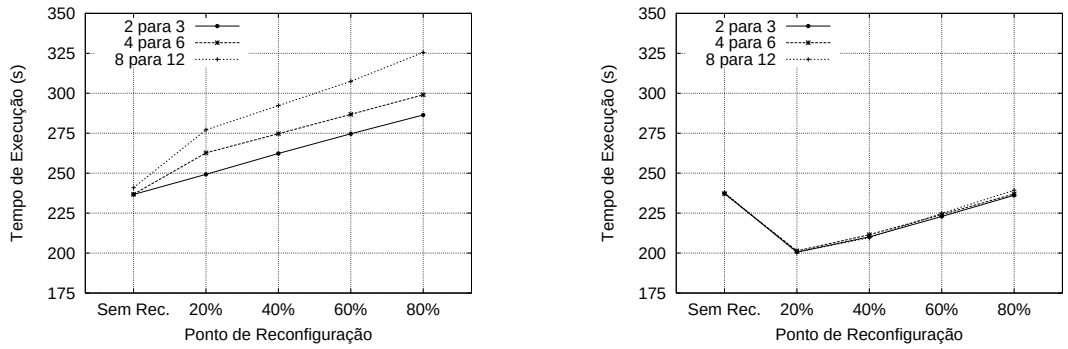
Visando medir o desempenho da utilização do assinalamento consistente para roteamento via fluxo rotulado realizamos outra série de experimentos utilizando a aplicação Emissor-Receptor. Essa aplicação foi modificada e a base de dados passou a ser armazenada no estado global distribuído. Um trecho do pseudocódigo do filtro Processador é mostrado na Figura 6.12. A chamada de função da linha 2 garante que, caso o dado já tenha sido armazenado, o dado estará em memória local para seu processamento (linha 4). Por fim, na linha 6, o dado armazenado no EGD é atualizado e, a partir deste do final desta chamada, pode ser acessado por qualquer outro processo. Novamente, a migração transparente dos dados armazenados no estado foi atingida de maneira simples.

```
1 foreach msg received do
2     egd_get(VAR, msg.id, dataSize, data);
3     if data != NULL then
4         processa(msg, dataSize, data);
5     end
6     egd_put(VAR, msg.id, dataSize, data);
7 end
```

Figura 6.12. Pseudocódigo filtro Processador

Ainda seguindo a idéia de avaliar de forma mais precisa o impacto do momento da reconfiguração no desempenho geral da aplicação, foram executados experimentos onde

o número de instâncias aumenta em 50% em pontos pré-determinados da execução. Com essa escolha também diferenciamos um pouco das análises realizadas até aqui, onde dobramos a quantidade de máquinas. Visando tornar tornar a análise ainda mais refinada, aumentamos a quantidade de pontos de reconfiguração para 4: 20%, 40%, 60% e 80% da execução. A Figura 6.13 mostra os tempos de execução, comparando o desempenho utilizando o assinalamento baseado nas funções mod e consistente.



(a) Hash Mod

(b) Hash Consistente

Figura 6.13. Tempos de execução da aplicação Emissor-Receptor para o hash mod e consistente

É possível observar que os gráficos mostrados nas Figuras 6.13(a) e 6.13(b) possuem a mesma escala e faixa nos eixos das abcissas e das ordenadas, facilitando a comparação. Em cada gráfico são mostradas três curvas, uma para cada número de instâncias: quando partimos de 2 para 3 instâncias, de 4 para 6 e de 8 para 12. Notamos que, para o *hash* mod, em todos os casos, a adição de instâncias leva a uma perda de desempenho. Essa perda chega a aproximadamente 38% quando partimos de 8 para 12 instâncias, com 80% da execução completa. O principal motivo para essa perda é o que o tempo para realização das migrações devido ao rearranjo dos assinalamentos superou os ganhos de processamento oriundos da utilização de mais recursos computacionais.

Uma análise assintótica do comportamento da função *mod* nestes cenários mostra que a manutenção do balanceamento pode resultar num tráfego da ordem de $\Theta\left(nk/(1.5 * n)\right)$. Uma vez que estamos realizamos experimentos do tipo *scaleup* e o tamanho do estado, k, aumenta proporcionalmente ao número de instâncias, n, teremos um número crescente de migrações. Esse resultado analítico é confirmado pela Figura 6.14, onde são mostrados os tempos médios de migração do estado para o assinalamento realizado com ambos os tipos de *hash*.

A utilização do reassinalamento consistente leva a ganhos de desempenho em quase todos os casos, os quais podem ser observados na Figura 6.13(b). No melhor caso,

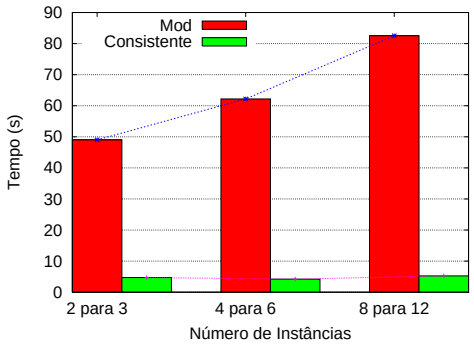


Figura 6.14. Tempo médio de transferência com reassinalamentos

quando a reconfiguração acontece com 20% da execução completa, é obtido aproximadamente 20% de ganho, em média. Já no pior caso, último ponto de reconfiguração, temos que não existe ganho. Isto pois o tempo gasto com migração de dados foi similar ao ganho obtido a adição de máquinas.

Outro ponto que chama atenção na Figura 6.13(b) é a quase sobreposição das curvas. Isto se deve ao fato que o número de migrações é $O(k/n)$, para k chaves e n instâncias. Uma vez que o numerador e o denominador desta divisão crescem segundo uma mesma taxa, um mesmo impacto é sentido em todos os cenários, independente do número de instâncias. Essa análise também é comprovada pela Figura 6.14, uma vez que o custo de migrações permanece quase constante, para todos casos.

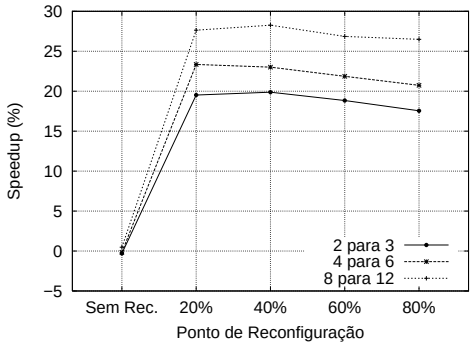


Figura 6.15. Speedup atingido com a utilização do hash consistente em relação ao hash mod

Um resumo dos ganhos do *hash* consistente em relação ao *mod* é mostrado na Figura 6.15. Nele observamos que, no melhor caso, temos uma ganho de aproximada-

mente 28%. Uma vez que esses ganhos são proporcionais tanto ao número de instâncias quanto ao tamanho da base de dados, podemos extrapolar as análises analíticas e experimentais feitas até aqui e concluir que esse ganho de desempenho cresce se qualquer destes parâmetros crescer. Além disso, dado que no cenário consistente o custo com a migração permanece constante, esse ganho no desempenho é devido somente à piora do padrão de comunicação obtido com a utilização do *hash mod*.

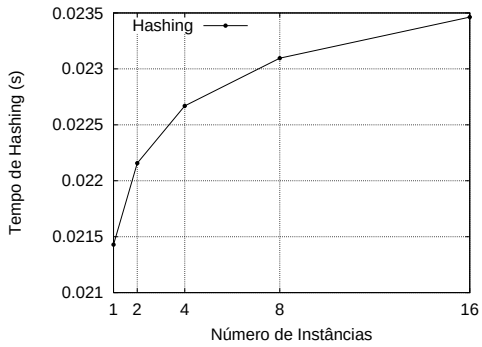


Figura 6.16. Tempo de assinalamento no hash consistente

Por fim, terminamos essa série de análises mostrando o tempo para determinação do assinalamento na política de fluxo rotulado consistente. Na Figura 6.16 vemos que a adição de instâncias leva a um impacto mínimo no tempo do cálculo do assinalamento e que esse pode ser desconsiderado quando comparamos com o tempo de execução da aplicação. Em média, este tempo é uma ordem de grandeza maior que o tempo no caso da função *mod*.

Capítulo 7

Conclusões e Trabalhos Futuros

Neste trabalho apresentamos um conjunto de extensões que adicionam suporte eficiente ao aumento da plataforma de computação em tempo de execução no ambiente Anthill. Acreditamos que esse seja o primeiro passo para utilização eficaz de plataformas dinâmicas no processamento de aplicações que exploram o paralelismo massivo.

Com a implementação do mecanismo de adição de instâncias em tempo de execução, lidar de maneira eficiente com a migração dinâmica de dados passou a ser de fundamental importância para execução de aplicações com estado. Assim, foi criado o Estado Global Distribuído, um módulo do Anthill que tem como principal objetivo esconder do programador detalhes inerentes à redistribuição eficiente do estado das aplicações, mantendo a simplicidade das abstrações oferecidas pelo paradigma filtro-fluxo. Com a utilização desse, o estado distribuído entre as cópias transparentes é exportado como um único espaço de endereçamento, podendo ser acessada a partir de qualquer instâncias como vetor local.

Por fim, foi diagnosticada a ineficiência da função padrão de distribuição do dados via fluxo-rotulado, quando expostos a ambientes dinâmicos. Isto levou a implementação de nova versão desta função de acordo com o protocolo de *hash* consistente, proposto por Karger et al. [1997]. Resultados experimentais atestam sua eficiência no rearranjo das partições do estado, ou seja, na ocorrência de reconfigurações, será movida a menor quantidade de dados possível para balancear a carga dentre as instâncias do filtro.

Essas implementações tiveram seu desempenho medido de forma experimental. Os resultados mostram que essa implementação viabiliza a utilização de mais poder computacional em tempo de execução, mantendo, com um custo baixo, a consistência e a continuidade da exploração assíncrona do paralelismo em diversos níveis.

Nesse trabalhado foram consideradas apenas plataformas de execução homogêneas. Uma extensão natural é levar em consideração heterogeneidade da plataforma na distribuição dos dados. Brinkmann et al. [2002] e Schindelhauer e Schomaker [2005]

propuseram funções de assinalamento variáveis com esse objetivo. Tal proposta poderia ser implementada sobre o arcabouço exposto neste trabalho, habilitando a execução eficiente de aplicações Anthill em plataformas dinâmicas heterogêneas.

Outra área de investigação é a utilização de replicação no estado global distribuído para inserir tolerância a falhas em execução de aplicações Anthill. Uma vantagem nessa abordagem é que o mecanismo pode completamente abstraído, uma vez que toda a implementação pode ficar na no módulo de memória compartilhada distribuída.

Referências Bibliográficas

Abramson, D.; Giddy, J. e Kotler, L. (2000). High performance parametric modeling with Nimrod/G: Killer application for the global grid? In *IPDPS '00: Proceedings of the 14th International Symposium on Parallel and Distributed Processing*, pp. 520–528, Washington, DC, USA. IEEE Computer Society.

Agrawal, R. e Srikant, R. (1994). Fast algorithms for mining association rules in large databases. In *VLDB '94: Proceedings of the 20th International Conference on Very Large Databases*, pp. 487–499, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Bal, H.; Kaashoek, M. e Tanenbaum, A. (1992). Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18:190–205.

Bennett, J. K.; Carter, J. B. e Zwaenepoel, W. (1990). Munin: distributed shared memory based on type-specific memory coherence. In *PPOPP '90: Proceedings of the second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pp. 168–176, New York, NY, USA. ACM.

Bershad, B. N.; Zekauskas, M. J. e Sawdon, W. A. (1993). The Midway distributed shared memory system. Technical report, Pittsburgh, PA, USA.

Beynon, M.; Chang, C.; Catalyurek, U.; Kurc, T.; Sussman, A.; Andrade, H.; Ferreira, R. e Saltz, J. (2002). Processing large-scale multi-dimensional data in parallel and distributed environments. *Parallel Computing*, 28(5):827–859.

Beynon, M. D.; Kurc, T.; Catalyurek, U.; Chang, C.; Sussman, A. e Saltz, J. (2001). Distributed processing of very large datasets with Datacutter. *Parallel Computing*, 27(11):1457–1478.

Brinkmann, A.; Salzwedel, K. e Scheideler, C. (2002). Compact, adaptive placement schemes for non-uniform requirements. In *SPAA '02: Proceedings of the fourteenth*

annual ACM symposium on Parallel algorithms and architectures, pp. 53–62, New York, NY, USA. ACM.

Buyya, R. (1999). *High Performance Cluster Computing: Architectures and Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

Carriero, N. J.; Gelernter, D.; Mattson, T. G. e Sherman, A. H. (1994). The linda alternative to message-passing systems. *Parallel Computing*, 20(4):633–655.

Carter, J. B. (1995). Design of the Munin distributed shared memory system. *Parallel Distribued Computing*, 29(2):219–227.

da Mata, L. L. P.; Pereira, F. M. Q. e Ferreira, R. (2009). Automatic parallelization of canonical loops. Submitted to XIII Brazilian Symposium on Programming Languages.

Deatrich, D.; Liu, S.; Payne, C.; Tafirout, R.; Walker, R.; Wong, A. e Vetterli, M. (2008). Managing petabyte-scale storage for the atlas tier-1 centre at triumf. In *HPCS '08: Proceedings of the 2008 22nd International Symposium on High Performance Computing Systems and Applications*, pp. 167–171, Washington, DC, USA. IEEE Computer Society.

Doyle, S.; Rodriguez, C.; Madabhushi, A.; Tomaszeweski, J. e Feldman, M. (2006). Detecting prostatic adenocarcinoma from digitized histology using a multi-scale hierarchical classification approach. *Engineering in Medicine and Biology Society, 2006. EMBS '06. 28th Annual International Conference of the IEEE*, pp. 4759–4762.

Ferreira, R.; Meira Jr., W.; Guedes, D.; Drummond, L.; Coutinho, B.; Teodoro, G.; Tavares, T.; Araujo, R. e Ferreira, G. (2005). Anthill:a scalable run-time environment for data mining applications. In *Symposium on Computer Architecture and High-Performance Computing (SBAC-PAD)*.

Fireman, D.; Teodoro, G.; Cardoso, A. e Ferreira, R. (2008). A reconfigurable runtime system for filter-stream applications. In *SBAC-PAD '08: Proceedings of the 2008 20th International Symposium on Computer Architecture and High Performance Computing*, pp. 149–156, Washington, DC, USA. IEEE Computer Society.

Gurcan, M. N.; Kong, J.; Sertel, O. e Cambazoglu, B. (2007). Computerized pathological image analysis for neuroblastoma prognosis. *Proceedings of the 2007 American Medical Informatics Association Annual Symposium (2007)*, pp. V –525–V –528.

Karger, D.; Lehman, E.; Leighton, T.; Panigrahy, R.; Levine, M. e Lewin, D. (1997). Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *STOC '97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pp. 654–663, New York, NY, USA. ACM.

Keleher, P.; Cox, A. L.; Dwarkadas, S. e Zwaenepoel, W. (1994). Treadmarks: distributed shared memory on standard workstations and operating systems. In *WTEC'94: Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, pp. 10–10, Berkeley, CA, USA. USENIX Association.

Kesselman, C. e Foster, I. (1998). *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers.

Kobatake, H.; Yoshinaga, Y. e Murakami, M. (1994). Automatic detection of malignant tumors on mammogram. *Image Processing, 1994. Proceedings. ICIP-94., IEEE International Conference*, 1:407–410 vol.1.

Kong, J.; Shimada, H.; Boyer, K.; Saltz, J. e Gurcan, M. (2007). Image analysis for automated assessment of grade of neuroblastic differentiation. *Biomedical Imaging: From Nano to Macro, 2007. ISBI 2007. 4th IEEE International Symposium on*, pp. 61–64.

Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Communications ACM*, 21(7):558–565.

Lamport, L. (1979). How to make a multiprocessor computer that correctly executes multiprocess program. *IEEE Transactions on Computers*, 28(9):690–691.

Li, K. (1986). *Shared virtual memory on loosely coupled multiprocessors*. PhD thesis, New Haven, CT, USA.

Li, K. e Hudak, P. (1989). Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359.

Matsumoto, M. e Nishimura, T. (1998). Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions Modelling and Computer Simulation*, 8(1):3–30.

Metropolis, N. e Ulam, S. (1949). The Monte Carlo method. *Journal of the American Statistical Association*, 44(247):335–341.

Narayanan, S.; Kurc, T.; Catalyurek, U. e Saltz, J. (2003). Database support for data-driven scientific applications in the grid. *Parallel Processing Letters*, 13(2):245–271.

Nasser Esgiar, A.; Naguib, R.; Sharif, B.; Bennett, M. e Murray, A. (1998). Microscopic image analysis for quantitative measurement and feature identification of normal and cancerous colonic mucosa. *IEEE Transactions on Information Technology in Biomedicine*, 2(3):197–203.

Pelleg, D. e Moore, A. (1999). Accelerating exact k-means algorithms with geometric reasoning. In *KDD '99: Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 277–281, New York, NY, USA. ACM Press.

Reed, D. A.; Mendes, C. L.; da Lu, C.; Foster, I. e Kesselman, C. (2003). *The Grid 2: Blueprint for a New Computing Infrastructure - Application Tuning and Adaptation*. Morgan Kaufman, San Francisco, CA, second edição. pp.513-532.

Ruiz, A.; Sertel, O.; Ujaldon, M.; Catalyurek, U.; Saltz, J. e Gurcan, M. (2007). Pathological image analysis using the GPU: Stroma classification for neuroblastoma. *IEEE International Conference on Bioinformatics and Biomedicine*, pp. 78–88.

Schindelhauer, C. e Schomaker, G. (2005). Weighted distributed hash tables. In *SPAA '05: Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, pp. 218–227, New York, NY, USA. ACM.

Silva, A.; Almeida, H.; Macambira, T.; Neto, D. G.; Meira Jr., W. e Ferreira, R. (2005). Hashing consistente como ferramenta para distribuição de recursos em sistemas distribuídos reconfiguráveis. *VI Workshop em Sistemas Computacionais de Alto Desempenho WSCAD'2005*.

Snir, M. e Otto, S. (1998). *MPI-The Complete Reference: The MPI Core*. MIT Press, Cambridge, MA, USA.

Spencer, M.; Ferreira, R.; Beynon, M.; Kurc, T.; Catalyurek, U.; Sussman, A. e Saltz, J. (2002). Executing multiple pipelined data analysis operations in the grid. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pp. 1–18, Los Alamitos, CA, EUA. IEEE Computer Society Press.

Sunderam, V. S. (1990). PVM: a framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339.

Tanenbaum, A. S. (1994). *Distributed Operating Systems*. Prentice Hall PTR, 1º edição.

Veloso, A.; Meira Jr., W.; Ferreira, R.; Neto, D. G. e Parthasarathy, S. (2004). 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*, pp. 422–433, Pisa, Italy. Springer-Verlag GmbH.

Zaki, M. J.; Parthasarathy, S.; Ogihara, M. e Li, W. (1997). New algorithms for fast discovery of association rules. In Heckerman, D.; Mannila, H.; Pregibon, D.; Uthurusamy, R. e Park, M., editores, *3rd International Conference on Knowledge Discovery and Data Mining*, pp. 283–296. AAAI Press.