Um *Middleware* P2P Descentralizado para a Computação de *Workflows*

Thiago S. Siqueira[1], **Edmundo R. M. Madeira**[1]

[1]Instituto de Computação – Universidade Estadual de Campinas - Unicamp
Caixa Postal 6176 – Campinas – SP – Brasil

thiago.siqueira@students.ic.unicamp.br, edmundo@ic.unicamp.br

Abstract. *P2P Computing has been raised as an alternative and complementary solution to Grid Computing. The use of P2P technology is able to provide a flexible and decentralized execution and management of Grid workflows. In this paper we present a completely decentralized P2P middleware for workflow computing. The middleware collects the shared processing power of the peers in order to execute workflows, modeled as DAG structures, composed of a set of dependent tasks. Through a distributed scheduling algorithm and a leasing-based fault tolerance mechanism, the middleware achieves high execution parallelism and efficient execution recovery in failure occurrences. The middleware is implemented in Java, through RMI and the JXTA library. The obtained experimental results show the efficiency of the middleware in the distributed execution of workflows as well as the fast execution recovery.*

Resumo. *A computação sobre P2P tem surgido como uma solução alternativa e complementar às grades computacionais. O uso da tecnologia P2P é capaz de prover a flexibilização e descentralização dos processos de execução e gerenciamento de workflows nas grades computacionais. Neste trabalho é apresentado um middleware P2P completamente descentralizado para a computação de workflows. O middleware coleta o poder de processamento compartilhado pelos peers para possibilitar a execução de workflows, modelados como DAGs, compostos por um conjunto de tarefas dependentes. Através do processo distribuído de escalonamento de tarefas e do mecanismo de tolerância a faltas baseado em leasing, o middleware atinge um nível alto de paralelismo na execução e eficiência na recuperação de execuções em ocorrência de falhas. O middleware é implementado em Java, juntamente com RMI e a biblioteca JXTA. Os resultados experimentais obtidos mostram a eficiência do middleware na execução distribuída dos workflows assim como a recuperação rápida de execução em cenários com faltas.*

1. Introdução

As redes *Peer-to-Peer* (P2P) surgiram como fenômenos sociais e tecnológicos significantes na última década. Um número cada vez maior de aplicações tem feito uso da tecnologia P2P, dentre elas as mais conhecidas são as de compartilhamento de arquivos (por exemplo, BitTorrent), *instant messengers* (MSN) e VoIP (Skype). Uma classe de aplicações P2P que vem ganhando importância nos últimos anos é aquela que utiliza o ambiente P2P para suportar computação distribuída, as chamadas aplicações de *P2P*

Computing, onde os ciclos de processamento ociosos, disponíveis em computadores localizados nas bordas da Internet, são coletados e utilizados na computação de problemas complexos, como os de bioinformática, astronomia, física, etc.

Como soluções concorrentes e complementares às grades computacionais, as aplicações de *P2P Computing* também têm sido utilizadas na resolução dos mesmos tipos de problemas tradicionalmente realizados pelas grades. Um exemplo destes problemas, que tem se tornado um dos mais importantes e desafiadores em grades, são as aplicações que envolvem execução e gerenciamento de *workflows* [T. Fahringer 2005]. Para tirar proveito dos recursos distribuídos através de organizações virtuais multi-institucionais, o desenvolvimento de mecanismos de execução descentralizada de *workflows* se torna um aspecto importante para as grades computacionais, assim como um importante tema de pesquisa. A execução e gerenciamento de *workflows* baseados em P2P desponta como uma solução viável. Através da abordagem P2P é possível desenvolver uma infra-estrutura de execução de *workflow* descentralizada, podendo inclusive ser utilizada em conjunto com as grades computacionais para suportar, de forma eficiente e escalável, a execução de *workflows* ao longo da grade.

O objetivo deste trabalho é propor uma infra-estrutura P2P completamente descentralizada para a execução de *workflows*. Por meio do desenvolvimento de um *middleware* P2P, possibilita-se a coleção e utilização do poder de processamento dos *peers* participantes da rede. Para atingir tal objetivo, o *middleware* faz uso de mecanismos eficientes de escalonamento distribuído de tarefas e de tolerância a faltas.

Este trabalho está organizado da seguinte maneira: na Seção 2 são apresentados os conceitos básicos envolvidos nas redes P2P e em *workflows*. A Seção 3 aborda alguns trabalhos relacionados. Na Seção 4 são tratadas questões referentes à arquitetura do *middleware*, como os algoritmos, escalonamento e o mecanismo de tolerância a faltas. As questões relativas à implementação são apresentadas na Seção 5. Já na Seção 6 são apresentados os resultados experimentais observados na execução do *middleware*. Finalmente, a Seção 7 aborda as considerações finais e uma visão geral de trabalhos futuros.

2. Conceitos Básicos

Duas abordagens relativamente recentes de computação distribuída emergiram nos últimos anos, ambas com o objetivo de solucionar o problema da organização, reunião e coordenação de ambientes computacionais de grande escala: P2P e grades computacionais. Tais abordagens tiveram rápida evolução, ampla disseminação e aplicações de sucesso. Entretanto, apesar das semelhanças, P2P e grades computacionais são baseadas em diferentes ambientes e possuem diferentes requisitos [I. T. Foster 2003].

2.1. Redes P2P

Devido ao constante crescimento da Internet e à disponibilização cada vez maior de conectividade e largura de banda, as redes P2P continuam ganhando popularidade como uma forma barata de computação e compartilhamento de recursos [M. Ripeanu 2002]. Ao contrário dos sistemas distribuídos tradicionais, as redes P2P agregam um grande número de computadores que entram e saem frequentemente da rede e podem não possuir endereço de rede (IP) permanente. Em [Lichun Ji 2005] redes P2P são definidas como uma classe de sistemas e/ou aplicações que utilizam recursos distribuídos de forma

descentralizada e autônoma para alcançar um determinado objetivo, como realizar uma computação ou compartilhar arquivos. Os membros de uma rede P2P sempre estão em total controle de seus recursos locais. Tal autonomia torna o modelo P2P diferente de outras abordagens distribuídas, como o modelo cliente-servidor, por exemplo. Ao invés de possuir papéis estáticos e pré-definidos para os participantes, as redes P2P se baseiam em papéis dinâmicos, como resultado da auto-organização dos *peers* em provedores e consumidores de recursos. Conseqüentemente, redes P2P tendem a ser descentralizadas, altamente dinâmicas e heterogêneas.

A implementação das redes P2P geralmente envolve a criação de redes *overlay*, ou seja, redes virtuais sobrepostas, auto-organizadas e independentes da estrutura da Internet subjacente. Isto significa que as redes *overlay* e Internet são diferentes na maioria das vezes. Por exemplo, dois nós (*peers*) podem estar conectados diretamente na rede *overlay* e separados por dezenas de nós na Internet. As características intrínsecas às redes *overlay* oferecem uma série de funcionalidades às aplicações P2P como arquitetura de roteamento em escala global, detecção de permanência, autenticação, anonimato e alta escalabilidade [Keong et al. 2005].

2.2. *Workflows*

Workflows são estruturas que permitem a automatização de procedimentos de negócios através da passagem de documentos, informações ou tarefas de um participante para outro, de acordo com um conjunto de regras bem definidas [Yolanda Gil 2007]. A utilização de *workflows* tem sido importante para as grades computacionais pelo fato de que estes permitem organizar os vários serviços da grade, combinando e propiciando aos usuários novos tipos de serviços. Tais benefícios também têm sido incorporados nas aplicações P2P que fazem uso de *workflows*.

No escopo deste trabalho, os *workflows* utilizados são modelados como grafos de dependências acíclicos, ou DAGs (*Directed Acyclic Graph*). Neste tipo de estrutura, cada nó do grafo representa uma tarefa computacional a ser processada, enquanto que as arestas direcionadas determinam o fluxo de execução e estabelecem as dependências entre as várias tarefas componentes do grafo.

3. Trabalhos Relacionados

A grande maioria das soluções de grades computacionais não suporta a execução coordenada de um conjunto de tarefas dependentes. Entretanto, aplicações de grades de sucesso têm sido desenvolvidas para o gerenciamento e execução de *workflows*. Em [Cao et al. 2003] é apresentado um *framework* baseado em agentes responsável pelo gerenciamento, execução e escalonamento de *workflows*, tanto de forma local (mesma organização) quanto global (todas as organizações). Em [Churches et al. 2006] é abordado um vasto conjunto de ferramentas que possibilitam a composição eficiente e flexível de *workflows*. Nesta solução também é possível especificar mecanismos responsáveis em distribuir sub-grafos do *workflow* para execução remota em recursos da grade. Já em [Oinn et al. 2006] é implementado um *middleware* para suportar experimentos do tipo *data intensive* de biologia molecular. Através da utilização de mais de 1000 serviços pré-desenvolvidos é possível compor mecanismos de escalonamento de tarefas, partição e execução de *workflows*. Apesar destes mecanismos complexos e eficientes de gerência

e execução de *workflows*, tais soluções não abordam as faltas resultantes da intermitência do ambiente das grades computacionais, sendo assim, nenhum mecanismo de recuperação de execução é especificado.

De acordo com as características dos ambientes P2P, novas aplicações baseadas em *workflows* também têm sido desenvolvidas sobre tal tecnologia. Uma plataforma P2P especializada na execução de aplicações baseadas em DAG é apresentada em [Hantz 2006], cujo objetivo visa a coleta de recursos para a computação do DAG. Nesta solução o escalonamento é realizado de forma estática e centralizada através de *peers* especializados, e nenhum tipo de falha é tratado. Em [Jun Yan 2006] é desenvolvida uma infra-estrutura P2P descentralizada para o gerenciamento e execução de *workflows*. Neste tipo de solução verifica-se a existência de repositórios locais em todos os *peers* como forma primária de tolerância a faltas, entretanto, não existe tratamento para situações onde *peers* falham e saem da rede. Funcionalidades interessantes desta infra-estrutura são o escalonamento e controle de execução do *workflow*, realizados de forma completamente descentralizada. Outra aplicação P2P que trata a execução de *workflows* é apresentada em [Joan Esteve Riasol 2006], a qual é baseada na especificação JXTA. Nesta solução nota-se a modificação dos protocolos básicos JXTA a fim de se alcançar um melhor desempenho na transmissão e execução de tarefas do *workflow* ao longo da rede.

4. Arquitetura do *Middleware*

O principal objetivo do *middleware* proposto neste trabalho é utilizar a flexibilidade e o dinamismo que os ambientes P2P oferecem a fim de suportar computação distribuída e cooperativa. Além do mais, o *middleware*, diferentemente da maioria das soluções de grades computacionais e P2P, suporta a execução distribuída e coordenada de *workflows*, ou seja, gerencia a execução de um conjunto de tarefas dependentes de acordo com regras bem definidas.

No desenvolvimento da arquitetura do *middleware* foram levados em consideração aspectos tanto de aplicações P2P quanto de aplicações de computação distribuída. Na Figura 1 são apresentados os módulos constituintes do *middleware*, identificados durante a fase de análise de requisitos.

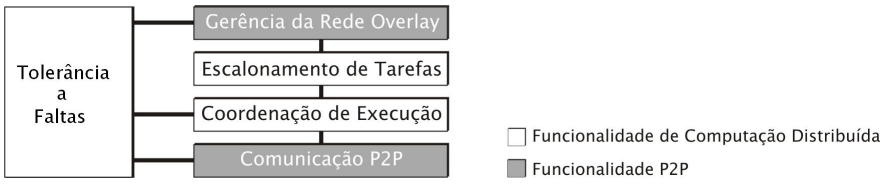

Figura 1. Módulos que compõem o *middleware*

O módulo de **Gerência da Rede Overlay** contém um conjunto de procedimentos comuns às aplicações P2P e é responsável pela descoberta e adição de novos *peers* à rede. Já o módulo de **Comunicação P2P**, também comum à maioria das aplicações P2P, lida com os aspectos referentes à comunicação propriamente dita entre os *peers* participantes da rede. Por se tratarem de aspectos basicamente atrelados às tecnologias utilizadas, estes módulos são tratados em mais profundidade na Seção 5, que aborda as questões associadas à implementação.

O módulo de **Escalonamento de Tarefas**, juntamente com o módulo **Coordenação de Execução**, é o encarregado em distribuir as tarefas componentes do DAG aos *peers* da rede e fazer com que as dependências entre tarefas sejam respeitadas. Estes dois módulos são apresentados em detalhes na Seção 4.1. Já o módulo de **Tolerância a Faltas** encapsula os mecanismos que propiciam ao *middleware* resiliência alta a faltas e recuperação de execução. Tal módulo é discutido na Seção 4.2.

4.1. Escalonamento e Coordenação de Execução

O problema do escalonamento de tarefas é uma das questões mais pesquisadas no escopo das aplicações de computação distribuída. O escalonamento se torna um tema importante visto que o tempo de execução de uma aplicação está diretamente relacionado às estratégias e à forma com que o escalonamento é realizado.

No *middleware*, o módulo **Escalonamento de Tarefas** encapsula as técnicas e procedimentos que gerenciam como a distribuição (escalonamento) das tarefas será realizada entre os *peers*. Visto que o *middleware* se destina à execução de *workflows*, que por sua vez são compostos por um conjunto de tarefas dependentes, os mecanismos de escalonamento também são os responsáveis em assegurar que as regras (dependências) impostas pelo *workflow* sejam obedecidas. A gerência de tais regras e o controle de execução das tarefas são definidos no módulo **Coordenação de Execução**, que, juntamente com os mecanismos de escalonamento, garante a correta execução do *workflow*.

No escopo deste trabalho, como discutido na Seção 2.2, os *workflows* são modelados como um grafo de dependência de tarefas, a partir de agora referenciados simplesmente como DAG. Este DAG é utilizado como entrada para o processo de escalonamento no *middleware* e é composto por um conjunto de tarefas, as quais possuem seus respectivos custos computacionais. Estes custos podem, por exemplo, representar a quantidade de FLOPS necessária para a execução da tarefa. Para aumentar o paralelismo na execução, o DAG de entrada pode ser particionado em sub-DAGs, ou ***DAG slices***, que representam subconjuntos de tarefas independentes uns dos outros. É importante ressaltar que esta independência se verifica em *DAG slices* de mesma hierarquia. Dentro de cada *DAG slice*, as tarefas são identificadas e numeradas univocamente em ordem crescente, de acordo com a relação de dependência entre elas. A Figura 2 apresenta um exemplo de DAG de entrada do *middleware*.

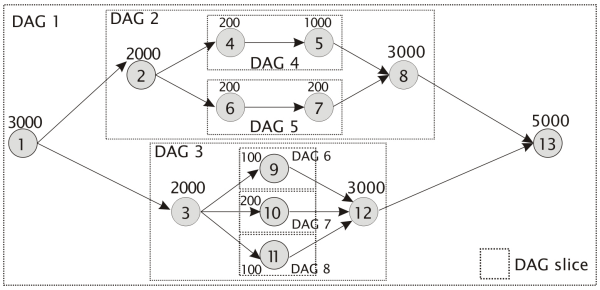

Figura 2. Exemplo de DAG de entrada

Na Figura 2 é possível notar a forma com que o DAG é particionado em *DAG slices*, as dependências entre tarefas e os respectivos custos computacionais de cada tarefa componente do DAG.

Não é objetivo do *middleware* alcançar um escalonamento ótimo das tarefas do DAG, mas sim fazer uso do poder de processamento disponibilizado pelos *peers* de maneira cooperativa, a fim de agregar recursos computacionais. Logo, o processo de escalonamento no *middleware* leva em consideração apenas o poder de processamento que cada *peer* quer compartilhar e o custo computacional das tarefas envolvidas na computação.

Todos os *peers* envolvidos na computação de um DAG são responsáveis pelo escalonamento de suas tarefas, ou seja, o processo de escalonamento é realizado de forma descentralizada e distribuída. Em decorrência desta igualdade de funcionalidades, todos os *peers* também podem submeter DAGs para serem executados. O *peer* que submete um DAG para execução no *middleware* é denominado ***DAG owner***. Cabe ao *DAG owner* a estruturação, população e estabelecimento das dependências entre as tarefas e *DAG slices*.

Algoritmo de Escalonamento

O algoritmo ***DAG Slice Distribution***, apresentado no Algoritmo 1, é o algoritmo de escalonamento utilizado no *middleware* e é executado em todos os *peers* que realizam alguma computação.

Algorithm 1 *DAG Slice Distribution* - Visão Geral

```
1:  peer P recebe DAG slice S do peer Q
2:  t = tarefa com menor ID de S
3:  while P é capaz de executar t do
4:      while P não receber todos os operandos de t do
5:          fica bloqueado até receber os operandos de t
6:      if t é a última tarefa de S then
7:          execute t e marca S como executado
8:          envia resultados de t para Q
9:          FINISH
10:     execute t
11:     propague resultados de t para tarefas e/ou DAG slice sucessores
12:     for cada DAG slice sucessor, S_t, de t do
13:         operandos de S_t = resultados de t
14:         envia S_t para U_t
15:         fica bloqueado até receber resultados de U
16:     t = tarefa com menor ID de S
```

O algoritmo de escalonamento pode ser entendido da seguinte forma: Um dado *peer* P recebe um *DAG slice* S de um *peer* Q (linha 1). No momento do recebimento de S, P armazena a tarefa de menor identificador de S, no caso t (linha 2). Por causa da dependência entre tarefas, um *peer* é capaz de executar apenas uma única tarefa por vez em um dado *DAG slice*. Sendo assim, para minimizar os custos de comunicação e transferência de dados, enquanto P puder computar tarefas de S, ele o fará (linha 3), resultando em uma melhor utilização do poder de processamento compartilhado pelos *peers* da rede. O algoritmo também mostra que tarefas dependentes podem bloquear a execução do DAG, ou seja, enquanto uma tarefa não receber todos os operandos necessários ela não pode retomar a computação (linhas 4 e 5). Se t é a última tarefa de S, t é executada (em P) e, após a computação de t, todo o *DAG slice* S é marcado como executado com sucesso. Após a marcação de S, seus resultados são retornados ao *peer* que possui o *DAG*

slice pai de S, ou seja, o *peer* Q. Sendo assim, encerra-se a participação do *peer* P na computação de S (linhas 6 a 9). Se t não é a última tarefa de S, t também será executada em P e seus resultados serão propagados para tarefas e/ou *DAG slices* posteriores (linhas 10 e 11). Para cada *DAG slice* sucessor de t, no caso S_t, P inicializa uma *thread* para gerenciar a execução de S_t (linha 12), e o processo de escalonamento escolhe um dado *peer* U_t para realizar sua computação (linha 14). Após o recebimento dos resultados de todos S_t o processo se repete, atribuindo-se a t a próxima tarefa de S com menor identificador (linha 16).

Um exemplo de escalonamento é apresentado na Figura 3. Neste caso, o DAG de entrada utilizado é o da Figura 2.

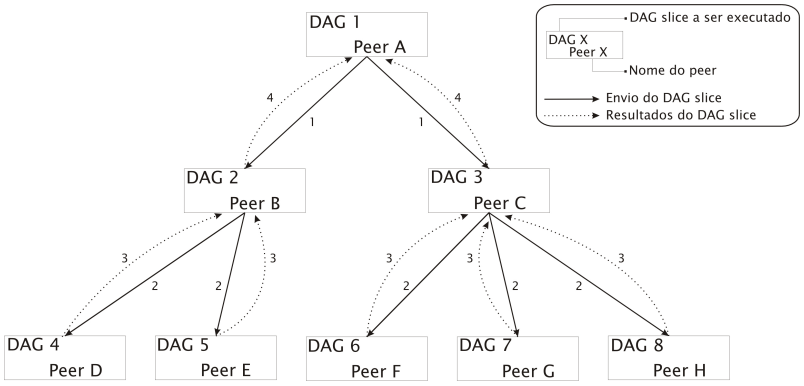

Figura 3. Árvore de execução gerada após o escalonamento

Como mostra a Figura 3, após o escalonamento (e execução) do DAG é formada uma árvore de execução, onde cada nó representa um *peer* computando um *DAG slice*. A formação desta árvore leva em consideração a hierarquia entre *DAG slices* (*DAG slices* pais e filhos) do DAG. Qualquer *peer* presente na *overlay* e que possua poder de processamento suficiente para executar um dado *DAG slice* é um candidato à sua computação. Logo, os *peers* escolhidos na Figura 3 são capazes de executar seus respectivos *DAG slices*. A Figura 3 também mostra a ordem com que o envio de *DAG slices* e seus resultados é realizada na execução do DAG.

4.2. Tolerância a Faltas

O principal diferencial deste trabalho com relação às soluções existentes de grades e aplicações P2P são os mecanismos utilizados para garantir a recuperação de execuções perdidas por causa de faltas. No *middleware* os mecanismos de tolerância a faltas são realizados de forma completamente descentralizada, característica esta que provê flexibilidade, alta resiliência a faltas e ausência de entidades centrais.

O módulo **Tolerância a Faltas** é o responsável em localizar, identificar e tratar as faltas que possam vir a ocorrer durante a execução do DAG. No escopo deste trabalho, as faltas consideradas são aquelas oriundas da intermitência dos ambiente P2P.

Através de mecanismos de *leasing*, o *middleware* é capaz de determinar rapidamente os estados dos *peers* participantes da computação de um DAG. *Leasing* pode ser definido como um padrão de projeto que simplifica o gerenciamento de recursos através da especificação de como os usuários ganham acesso a estes recursos de uma entidade

provedora por um período de tempo pré-definido [Michael Schneider 2007]. O esquema de *leasing* nos mecanismos de tolerância a faltas permite aos *peers* concederem *leases* (permissões) a outros *peers*. Estas *leases* devem ser renovadas de tempos em tempos. Se uma *lease* não é renovada, o *peer* que a concedeu a outro *peer* sabe que aquele *peer* não mais pertence à rede.

Um *peer* cliente permanece conectado a um *peer* servidor até que este último termine a computação do *DAG slice* correspondente; se o *peer* servidor falha ou sai da rede, o cliente é imediatamente notificado. Logo, em ambas as direções na árvore de execução do DAG é possível determinar os estados dos *peers* da rede.

A outra funcionalidade importante suportada pelos mecanismos de tolerância a faltas do *middleware* é a recuperação de execução em decorrência de faltas. Para suportar esta funcionalidade, o *middleware* faz uso de marcação de tarefas (*acknowledgment*). Todas as tarefas e *DAG slices* são marcados pelos respectivos *peers* que os computaram quando executados com sucesso. Para lidar com a ocorrência de faltas, todo *DAG slice* carrega consigo uma lista ordenada de *peers* pais na árvore de execução. Esta lista é formada durante o processo de escalonamento e leva em consideração a hierarquia de *DAG slices* e a distribuição dos peers. Para suportar o esquema de marcação de tarefas, e conseqüentemente a recuperação de execução assume-se que o *DAG owner* nunca falha.

Na ocorrência de faltas na computação de um DAG, e, de acordo com a árvore de execução gerada pelo escalonamento, pode-se identificar dois cenários distintos no processo de recuperação de execução: **1)** O *peer* que falha é um nó folha na árvore de execução. Neste caso, a computação realizada até então pelo *peer* que falhou não pode mais ser recuperada. Quando detectada este tipo de falha, o *peer* pai, localizado imediatamente acima na árvore de execução do *peer* que falhou, simplesmente requisita um novo escalonamento do *DAG slice*. **2)** O *peer* que falha é um nó intermediário na árvore de execução. Neste cenário, procedimentos mais complexos devem ser realizados para assegurar a correta retomada da execução do DAG, visto que a árvore de execução é desconectada. A falha é detectada simultaneamente através de dois acontecimentos: o cancelamento da conexão entre os *peers* pai e filhos (direção *top-down* na árvore de execução) e a não renovação da *lease* por parte do *peer* pai para com os *peers* filhos (*lease* expira - direção *bottom-up* na árvore de execução).

Nos Algoritmos 2 e 3 são apresentadas as atividades realizadas pelos *peers* clientes e servidores, respectivamente, no processo de recuperação de execução para os dois possíveis cenários abordados anteriormente. Tais algoritmos podem ser explicados através de um exemplo, no qual é utilizada como base a Figura 3. A seqüência de atividades envolvidas neste exemplo é enumerada na Figura 4. Por simplicidade, apenas os *peers* que participam do processo de recuperação de execução são mostrados.

Se Peer B falha (Figura 4 (1)) depois de distribuir os *DAG slices* 4 e 5 para Peer D e Peer E respectivamente, a *lease* concedida pelos dois *peers* a Peer B irá expirar (Algoritmo Servidor - linha 4, Figura 4 (2)), o que significa que Peer B não mais pertence à rede. Depois de computar seus respectivos *DAG slices*, Peer D e Peer E enviam seus resultados ao pai de Peer B, neste caso, Peer A, alertando-o sobre a falha de Peer B (Algoritmo Servidor - linhas 5 e 6, Figura 4 (4) e (5)). Estes avisos são necessários para determinar se o Peer B falhou antes ou depois de distribuir os *DAG slices* filhos a outros *peers*. Com os

resultados dos *DAG slices* 4 e 5, Peer A pode então continuar com a computação do *DAG slice* 2, assumindo que tarefas e *DAG slices* anteriores foram executados com sucesso, marcando-os (Algoritmo Cliente - linha 8). Logo, Peer A pode escolher entre computar o restante do *DAG slice* 2 ou enviá-lo a outro *peer* (Algoritmo Cliente - linha 9).

Algorithm 2 Peer "cliente"no mecanismo de recuperação de execução

```
1: for cada DAG-slice filho S_i do
2:     envia S_i para o peer P_i
3:     fica bloqueado até receber resultados de S_i
4:     if P_i desconecta ou falha then
5:         if S_i possui DAG slices filhos S_Si then {P_i é nó intermediário na árvore de
           execução}
6:             espera aviso e resultados dos peers filhos de P_i
7:             if recebeu resultados dos peers filhos de P_i then
8:                 ACK S_Si e tarefas anteriores em S_i
9:     escalona ou executa S_i
```

Algorithm 3 Peer "servidor"no mecanismo de recuperação de execução

```
1: recebe DAG-slice S do peer P
2: inicializa mecanismo de lease para P
3: computa ou escalona S
4: if lease expira(cliente saiu ou falhos) then
5:     avisa peer pai de P
6:     entrega resultados de S para peer pai de P
```

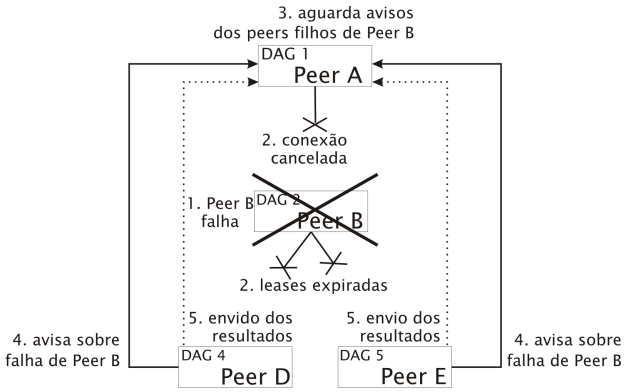

Figura 4. Etapas na recuperação de execução

5. Implementação

Para alcançar os requisitos de interoperabilidade, o *middleware* foi implementado através da linguagem de programação Java e outros produtos e tecnologias da família Java.

As funcionalidades P2P suportadas pelo *middleware*, como a descoberta de *peers* e gerenciamento da rede *overlay*, são implementadas através da especificação JXTA, baseada em Java. JXTA é um conjunto de protocolos abertos que permitem que quaisquer dispositivos conectados à rede comuniquem e colaborem entre si de forma P2P [Joan Esteve Riasol 2006].

Apesar das diversas funcionalidades oferecidas pela especificação JXTA, o único mecanismo JXTA utilizado é o serviço de descoberta de *peers*, através do qual o *middleware* é capaz de localizar e adicionar novos *peers* à rede *overlay*, onde serão utilizados em computações posteriores. Toda a adição de novos *peers* ou recursos na rede é feita através da publicação, via *broadcast*, de *advertisements*. O serviço de descoberta utiliza estes *advertisements* para localizar e adicionar *peers* à *overlay*. Quando um *peer* se junta à rede, este publica um *advertisement* contendo sua localização, identificação e a quantidade de processamento que deseja compartilhar com os outros *peers*.

Todos os procedimentos descritos anteriormente são encapsulados no módulo de **Gerência da Rede Overlay** (Figura 1). Tais procedimentos também são responsáveis pela configuração e introdução de um novo *peer* à *overlay*.

Os procedimentos envolvidos na comunicação ponto-a-ponto, encapsulados no módulo **Comunicação P2P** (Figura 1), e os mecanismos de tolerância a faltas foram implementados através de RMI. RMI provê funcionalidades úteis para aplicações distribuídas em Java, tal como *download* dinâmico de código, comunicação segura, mecanismo de *leasing*, dentre outros. A utilização de RMI na comunicação é justificada pelo maior desempenho que ele apresenta em relação aos esquemas de comunicação nativos da especificação JXTA, os chamados *pipes*. No *middleware*, utilizou-se o *download* automático de código provido pelo RMI para transferir as tarefas do cliente (apenas o *DAG owner* possui as classes que implementam as tarefas) para os *peers* cliente. Também foi utilizado o mecanismo eficiente de serialização e de gerenciamento de objetos Java através da rede, o que é um processo complexo, caro e demorado com a utilização de *pipes* JXTA. O mecanismo de *leasing* do RMI possibilita a simples, eficiente e transparente determinação dos estados dos nós envolvidos na comunicação. Através do comportamento bloqueante das chamadas RMI (cliente permanece bloqueado e conectado ao servidor até o final da chamada) e do esquema de *leasing* providos pelo RMI, desenvolveu-se os mecanismos de tolerância da faltas, que permitem a rápida recuperação de execução e determinação dos estados dos *peers* da rede.

A Figura 5 mostra o diagrama de classes UML dos componentes identificados durante a fase de modelagem do *middleware*.

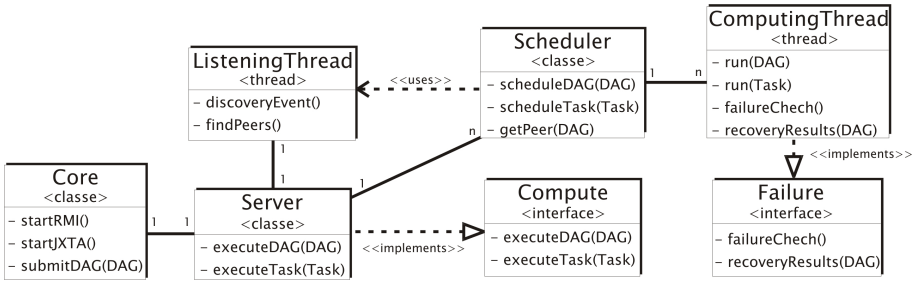

Figura 5. Diagrama de classe UML do *middleware*

A classe `Core` é a responsável em estabelecer os parâmetros de configuração iniciais tanto do ambiente JXTA quanto do RMI e inserir o novo *peer* na rede *overlay*. A classe `Server` representa o lado servidor RMI do *middleware*, a qual implementa a interface remota `Compute`, que provê métodos para a execução de *DAG slices* e tarefas. Através da invocação dos métodos da interface remota `Compute`, *peers* "clientes"podem

submeter *DAG slices* e/ou tarefas para computação em *peers* "servidores". A classe `Server` também realiza outra tarefa importante: a inicialização dos mecanismos de *leasing* do *middleware*. Se alguma *lease* expira, falha na direção *bottom-up* na árvore de execução, a classe `Server` se encarrega em obter informações sobre o ponto onde a execução foi interrompida para que os mecanismos de recuperação possam retomar a execução do *DAG slice*. A *thread* `ListeningThread` é responsável em "ouvir"a rede *overlay* JXTA a procura de novos *peers*. Implementada como uma *thread* que é executada indefinidamente, `ListeningThread` executa os procedimentos envolvidos no serviço de descoberta da especificação JXTA. A classe `Scheduler` implementa os procedimentos e algoritmos envolvidos nos processos de escalonamento e coordenação de execução de tarefas. `ComputingThread` é a estrutura que realmente realiza a comunicação entre *peers* no *middleware*. Implementada como *threads* independentes, cada instância de `ComputingThread` é responsável pelo estabelecimento e gerenciamento da conexão RMI entre um par de *peers* cliente e servidor para a computação de um *DAG slice*. Em outras palavras, `ComputingThread` representa o lado cliente RMI. `ComputingThread` também se responsabiliza em inicializar o processo de recuperação de execução na ocorrência de falha no *peer* servidor (falha na direção *top-down* na árvore de execução) através da implementação da interface remota `Failure`, que possui métodos empregados na notificação e envio de resultados em situações de falha.

6. Resultados Experimentais

Visto que a maioria das soluções de grades computacionais e aplicações P2P não considera a interdependência de tarefas, para avaliar a desempenho do *middleware* apresentado no presente trabalho, compara-se os efeitos e impactos de duas variáveis no tempo de execução total de um DAG: o número e nível de paralelismo de *DAG slices*, e a intermitência dos ambientes P2P.

Na primeira avaliação, considera-se o DAG da Figura 2, entretanto, cada nó do grafo é populado com uma mesma tarefa. Sendo assim, todos os nós do grafo possuirão uma tarefa de mesmo custo computacional. Primeiramente, o DAG é executado particionado em apenas um *DAG slice*, composto de treze tarefas seqüenciais. A partir de então, particiona-se este DAG inicial em DAGs com 3, 5, 7 e finalmente 8 *DAG slices*, da mesma forma como o DAG original, mas sempre mantendo o mesmo custo computacional de todo o DAG. O objetivo desta avaliação é medir o impacto e a eficiência do particionamento do DAG em *DAG slices*. Para esta avaliação utilizou-se seis *peers* (máquinas reais) distribuídos em duas sub-redes distintas. Além do mais, neste cenário, tais *peers* não apresentam faltas durante as simulações. Cada teste foi executado cinco vezes, sendo o resultado final de cada teste obtido pela média das cinco execuções. A Figura 6 mostra a comparação entre a execução local, onde apenas um *peer* é utilizado (apenas o *DAG owner*) e a execução do *middleware* proposto, em ambos os casos com números diferentes de *DAG slices*.

Os resultados da Figura 6 mostram a eficiência no particionamento do DAG de entrada em *DAG slices*. No melhor caso, com 8 *DAG slices*, o tempo de execução é reduzido em 45% em relação à execução local. O resultado consideravelmente melhor com 3 *DAG slices* pode ser possivelmente explicado através do menor custo de processamento em um mesmo *peer* que o custo de comunicação entre diferentes *peers*. A partir do momento em que menos comunicação é necessária com apenas 3 *DAG slices*, cada *peer* realiza uma

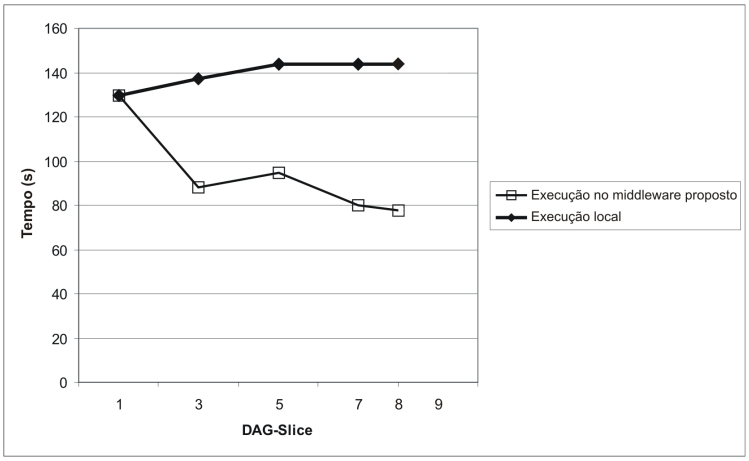

Figura 6. Avaliação da execução local e no *middleware* proposto

quantidade de processamento maior, visto que o DAG é composto por tarefas ***CPU intensive***. Como resultado, os custos de comunicação tenderam a ser minimizados, assim como o tempo total de execução. Apesar deste resultado particular, a Figura 6 mostra que quanto mais *DAG slices* são utilizados no DAG, menos tempo é gasto na sua execução, o que sugere que tanto o processo de escalonamento quanto a técnica de particionar o DAG em *DAG slices* são eficientes.

O segundo teste avalia como a intermitência do ambiente P2P e a ocorrência de faltas afetam o *middleware* e o tempo total de execução do DAG. Para efeito de comparação, uma outra versão do *middleware* foi implementada, na qual não existe a busca automática e dinâmica de novos *peers* na rede *overlay*. Nesta versão do *middleware*, é utilizado um nó central responsável em gerenciar os *peers* participantes. Este nó central mantém uma lista estática de um conjunto de *peers* previamente conhecidos, ou seja, se novos *peers* se juntam à rede após a criação da lista, tais *peers* não serão visíveis ao *middleware* e, se um *peer* presente na lista deixa a rede, nenhum aviso ou atualização na lista do nó central é feita, o que pode ocasionar em uma falha (que poderia ser evitada) em tempo de execução. Nesta versão, quando um *peer* submete um DAG para execução, ele primeiro contacta o nó central e recupera a lista de *peers* disponíveis naquele momento. Apenas os *peers* participantes na lista do nó central irão participar da computação do DAG. Para medir o dinamismo e a recuperação de execução do *middleware*, causa-se uma falha em um *peer* intermediário na árvore de execução (*peer* que possui *DAG slice* pai e filhos) durante a computação do DAG e adiciona-se dois novos *peers* após esta falha. Em ambas as versões do *middleware*, avalia-se novamente o tempo total de execução do DAG, considerando diferentes números de *peers* iniciais. A Figura 7 mostra os resultados obtidos.

Tanto na versão centralizada quando na versão original do *middleware*, a Figura 7 mostra um grande *overhead* nas execuções com 2 e 3 *peers* iniciais. Estes *overheads* são justificados pela falha em um *peer* "folha"na árvore de execução. Nestes casos, toda a computação realizada no *peer* que falhou é perdida e o DAG necessita ser reescalonado e executado novamente. Nas execuções restantes é possível notar que o comportamento dinâmico do ambiente P2P atenua a ocorrência da falha, minimizando seus efeitos no tempo total de execução do DAG. A Figura 7 também mostra que os resultados da versão original com faltas são bastante próximos dos resultados da versão centralizada sem faltas,

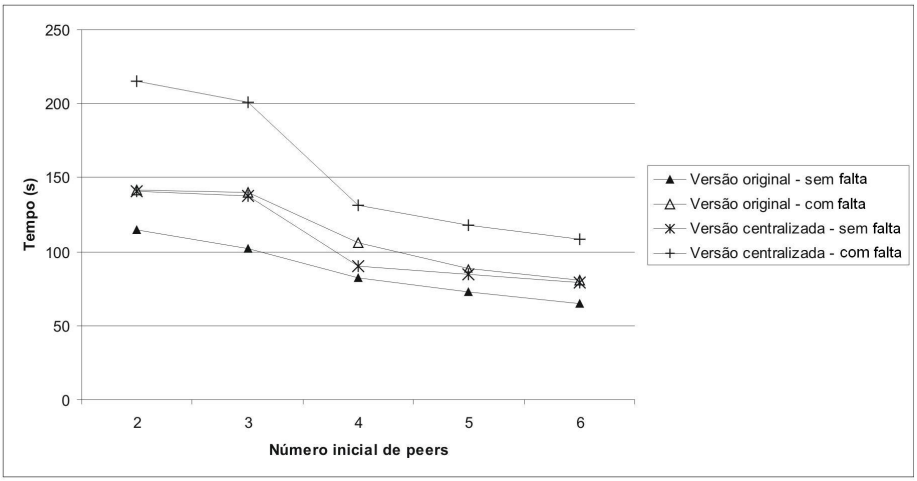

Figura 7. Avaliação da intermitência e efeitos de faltas

o que indica o bom desempenho dos mecanismos de tolerância a faltas e recuperação de execução do *middleware*.

7. Conclusão

O *middleware* apresentado neste trabalho foi desenvolvido com o objetivo de prover um ambiente robusto, eficiente e descentralizado para a computação de *workflows* através das vantagens que a tecnologia P2P oferece às aplicações distribuídas. Com a integração de funcionalidades como alta escalabilidade, ambientes auto-configuráveis e o dinamismo característico das redes P2P, novos tipos de aplicações e soluções complementares às grades computacionais podem ser desenvolvidas. A implementação do algoritmo de escalonamento distribuído e o mecanismo de tolerância a faltas possibilitaram ao *middleware* alcançar um alto nível de paralelismo na execução de *workflows* e a rápida recuperação de execução em ocorrência de faltas. As técnicas utilizadas na tolerância a faltas compõem a funcionalidade que diferencia o *middleware* de outras soluções de *P2P Computing*. Através do mecanismo baseado em *leasing*, o *middleware* trata de forma bastante eficaz a intermitência do ambiente P2P. Os resultados obtidos mostram a eficiência do *middleware* na execução distribuída de *workflows*, onde o *middleware* consegue um tempo de execução 45% menor que o da execução realizada localmente.

Como trabalhos futuros, pretende-se adicionar mais informações, como o custo de comunicação entre tarefas e *peers*, ao processo de escalonamento para possibilitar uma escolha mais eficiente de *peers* na computação do *workflow*; e criar mecanismos de persistência adicionais para a recuperação de execução no caso do *DAG owner* falhar.

Referências

Cao, J., Jarvis, S., Saini, S., and Nudd, G. (2003). Gridflow: workflow management for grid computing. In *Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid, CCGrid 2003*, pages 198–205.

Churches, D., Gombas, G., Harrison, A., Maassen, J., Robinson, C., Shields, M., Taylor, I., and Wang, I. (2006). Programming scientific and distributed workflow with triana services: Research articles. *Concurr. Comput. : Pract. Exper.*, 18(10):1021–1037.

Hantz, F. (2006). Light p2p platform of computing for dag. In *International Conference on Distributed Frameworks for Multimedia Applications (DFMA'06)*, pages 49–54, Penang, Malaysia.

I. T. Foster, A. I. (2003). On death, taxes, and the convergence of peer-to-peer and grid computing. In *2nd International Workshop on Peer-to-Peer Systems (IPTPS'03)*, pages 118–128, Berkeley, Califórnia, EUA.

Joan Esteve Riasol, F. X. (2006). Juxta-cat: a jxta-based platform for distributed computing. In *PPPJ '06: Proceedings of the 4th international symposium on Principles and practice of programming in Java*, pages 72–81, New York, NY, USA. ACM.

Jun Yan, Yun Yang, G. K. R. (2006). Swindew-a p2p-based decentralized workflow management system. *IEEE Transactions on Systems, Man, and Cybernetics, Part A*, 36(5):922–935.

Keong, L., Crowcroft, J., Pias, M., Sharma, R., and Lim, S. (2005). A survey and comparison of peer-to-peer overlay network schemes. *Communications Surveys & Tutorials, IEEE*, pages 72–93.

Lichun Ji, R. D. (2005). Coordination & enterprise wide p2p computing. In *SCC '05: Proceedings of the 2005 IEEE International Conference on Services Computing*, pages 141–148, Washington, DC, EUA. IEEE Computer Society.

M. Ripeanu, I. Foster, A. I. (2002). Mapping the gnutella network: Properties of large-scale peer-to-peer systems and implications for system design. *IEEE Internet Computing Journal*, 6(1).

Michael Schneider, Markus Aleksy, M. S. M. T. (2007). Leasing variants in distributed systems. *First International Conference on Complex, Intelligent and Software Intensive Systems, 2007. CISIS 2007.*, pages 68–73.

Oinn, T., Greenwood, M., Addis, M., Alpdemir, N. M., Ferris, J., Glover, K., Goble, C., Goderis, A., Hull, D., Marvin, D., Li, P., Lord, P., Pocock, M. R., Senger, M., Stevens, R., Wipat, A., and Wroe, C. (2006). Taverna: lessons in creating a workflow environment for the life sciences: Research articles. *Concurr. Comput. : Pract. Exper.*, 18(10):1067–1100.

T. Fahringer, J. Qin, S. H. (2005). Specification of grid workflow applications with agwl: an abstract grid workflow language. In *CCGRID '05: Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05) - Volume 2*, pages 676–685, Washington, DC, EUA. IEEE Computer Society.

Yolanda Gil, Pedro A. González-Calero, E. D. (2007). On the black art of designing computational workflows. In *WORKS '07: Proceedings of the 2nd workshop on Workflows in support of large-scale science*, pages 53–62, Nova York, NY, EUA. ACM Press.