Um Middleware Peer-to-Peer Descentralizado para a Computação de Workflows

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Thiago Senador de Siqueira e aprovada pela Banca Examinadora.

Campinas, 14 de Março de 2008.

Prof. Dr. Edmundo Roberto Mauro Madeira
(Orientador)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP
Bibliotecária: Maria Júlia Milani Rodrigues – CRB8a / 2116

Siqueira, Thiago Senador de
Si75m Um middleware Peer-to-Peer descentralizado para a computação de workflows / Thiago Senador de Siqueira -- Campinas, [S.P. :s.n.], 2008.

Orientador : Edmundo Roberto Mauro Madeira
Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

1. Sistemas distribuídos. 2. Computação distribuída. 3. Middleware. 4. Peer-to-Peer. I. Madeira, Edmundo Roberto Mauro. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

(mjmr/imecc)

Título em inglês: A decentralized P2P middleware for workflow computing.

Palavras-chave em inglês (Keywords): 1. Distributed systems. 2. Distributed computing. 3. Middleware. 4. Peer-to-Peer.

Área de concentração: Ciência da Computação

Titulação: Mestre em Ciência da Computação

Banca examinadora:

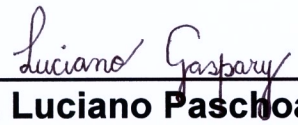
Prof. Dr. Luciano Paschoal Gaspary (Instituto de Informática – UFRGS)
Prof. Dr. Ricardo de Oliveira Anido (IC – UNICAMP)
Profa. Dra. Islene Calciolari Garcia (IC – UNICAMP)

Data da defesa: 14/03/2008

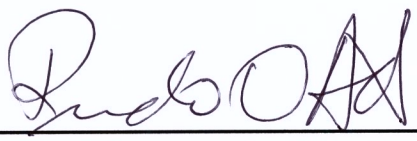
Programa de Pós-Graduação: Mestrado em Ciência da Computação

TERMO DE APROVAÇÃO

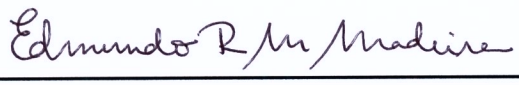
Dissertação Defendida e Aprovada em 14 de março de 2008, pela Banca examinadora composta pelos Professores Doutores:



Prof. Dr. Luciano Paschoal Gaspary
Instituto de Informática – UFRGS



Prof. Dr. Ricardo de Oliveira Anido
IC – UNICAMP



Prof. Dr. Edmundo Roberto Mauro Madeira
IC – UNICAMP

Instituto de Computação
Universidade Estadual de Campinas

Um Middleware Peer-to-Peer Descentralizado para a Computação de Workflows

Thiago Senador de Siqueira[1]

Março de 2008

Banca Examinadora:

- Prof. Dr. Edmundo Roberto Mauro Madeira (Orientador)
- Prof. Dr. Luciano Paschoal Gaspary - Instituto de Informática - UFRGS
- Prof. Dr. Ricardo de Oliveira Anido - IC - Unicamp
- Profa. Dra. Islene Calciolari Garcia - IC - Unicamp

[1]Suporte financeiro de Bolsa do CNPq - 2006/2008

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 faltas. 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.

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 work 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.

Agradecimentos

Gostaria de agradecer primeiramente à minha família, em especial à minha mãe, Patrícia, e à minha irmã, Ane, pelo apoio incondicional durante todas as etapas da minha vida;

Ao meu orientador, prof. Edmundo, que esteve sempre disponível e conduziu a orientação do meu trabalho com grande profissionalismo, compreensão e principalmente amizade;

Aos meus amigos de Aiuruoca, Rafael, Javan, Jeferson, Carlos Henrique e Natanael, pelos grandes momentos compartilhados, mesmo à distância;

Aos meus amigos da UFJF, em especial ao Daniel, Élder, Fábio, Vinícius, Rafael, Júlio e Rogério, pessoas com as quais aprendi muito e que me auxiliaram com dicas valiosas na implentação deste trabalho;

Aos meus amigos da Unicamp, Renato, Javier, Eliza, Rodrigo, Luiz, Neumar, Thiago, Douglas e Alonso *(in memoriam)*, pessoas que foram muito importantes nos momentos que passei em Campinas;

À minha sobrinha Julia, que, com apenas 4 anos, me proporciona momentos super divertidos;

À minha namorada Angélica, pessoa com a qual cresci e aprendi muito, principalmente no aspecto humano e pessoal;

A Jesus.

Sumário

Resumo **v**

Abstract **vi**

Agradecimentos **vii**

1 Introdução **1**

2 Conceitos Básicos **4**
2.1 Redes P2P 4
2.1.1 Redes P2P Estruturadas 6
2.1.2 Redes P2P Desestruturadas 8
2.2 Redes P2P X Grades Computacionais 9
2.3 Grades P2P 10
2.4 *Workflows* 11

3 Trabalhos Relacionados **14**

4 Arquitetura e Algoritmos do *Middleware* **18**
4.1 Análise de Requisitos 18
4.2 Arquitetura do *Middleware* 19
4.3 Escalonamento e Coordenação de Execução 20
4.4 Tolerância a Faltas 27
4.4.1 *Leasing* 27
4.4.2 Recuperação de Execução 29
4.4.3 Algoritmos de Recuperação de Execução 30

5 Implementação e Resultados Experimentais **37**
5.1 JXTA 37
5.1.1 Protocolos JXTA 39

5.1.2 JXTA em Execução 40
5.1.3 JXTA no *Middleware* 42
5.2 Java RMI 43
5.3 RMI + JXTA 45
5.4 Modelagem e Seqüência de Eventos 47
5.5 Construção, Particionamento e População do DAG 51
5.6 Resultados Experimentais 54

6 Conclusão **57**

Bibliografia **59**

Lista de Tabelas

5.1 Relação entre módulos e componentes que os implementam 50

Lista de Figuras

2.1 Rede *overlay* formada sobre a Internet 6
2.2 Napster - Hierarquia de *peers* além de comunicação cliente-servidor e P2P . 7
2.3 Geometria e busca na CAN 8
2.4 Disposição dos *peers* na rede Gnutella 9
2.5 *Workflow* modelado como DAG 12

4.1 Módulos que compõem o *middleware* 19
4.2 Exemplo de DAG de entrada 21
4.3 Organização de tarefas e *DAG slices* 23
4.4 Árvore de execução gerada após o escalonamento 26
4.5 Passos envolvidos na recuperação de execução 36

5.1 Pilha de protocolos JXTA 40
5.2 Cenário básico de uma aplicação JXTA 41
5.3 PeerMI no estabelecimento de conexão RMI 47
5.4 Diagrama de classe UML do *middleware* 48
5.5 Diagrama de seqüência de eventos UML em um cenário sem faltas 51
5.6 Sequência de eventos com a ocorrência de falta 52
5.7 Estruturas de criação e composição de DAGs 53
5.8 Avaliação da execução local e no *middleware* proposto 55
5.9 Avaliação da intermitência e efeitos de faltas 56

Capítulo 1

Introdução

As redes *Peer-to-Peer* (P2P) surgiram como fenômenos sociais e tecnológicos significativos na última década. Através do crescimento da Internet, maior disponibilização de conectividade e aumentos progressivos de largura de banda, as redes P2P continuam ganhando popularidade e vêm se despontando como uma forma econômica de computação e compartilhamento de recursos. Um número cada vez maior de aplicações tem feito uso da tecnologia P2P, dentre elas as mais conhecidas são as de compartilhamento de arquivos, como BitTorrent, Emule, dentre outras. Outras aplicações como *instant messengers* (por exemplo, MSN), VoIP (Skype) e ferramentas colaborativas (Groove) também têm tido crescimentos expressivos. 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.

De acordo com as características dos ambientes P2P, as aplicações de *P2P Computing* podem ser vistas como soluções alternativas e complementares às grades computacionais, visto que ambas abordagens possuem objetivos semelhantes, como reunião, organização e coordenação de recursos espalhados na rede. Para suportar computação distribuída em tais ambientes várias questões devem ser abordadas. Por causa do tamanho, autonomia e alta volatilidade dos recursos, estas plataformas provêem a oportunidade de revisitar aspectos importantes de sistemas distribuídos como protocolos, infra-estruturas, segurança, tolerância a faltas, escalonamento, desempenho, serviços, aplicações e incentivos à cooperação.

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, é o caso das aplicações que envolvem execução e gerenciamento de *workflows* [12]. Muitas aplicações científicas complexas requerem a criação de um sistema de gerenciamento e execução de *workflows* como parte de seus sofisticados processos de solução de problemas em grades computacionais. 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. Diante das características e requisitos destes cenários, 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. Através dos mecanismos de escalonamento do *middleware*, o *workflow*, modelado como um grafo de dependências, ou DAG (*Directed Acyclic Graph*), é particionado em estruturas menores (*DAG slices*) independentes umas das outras, as quais são enviadas para execução em *peers* remotos. O *middleware* também conta com um mecanismo de coordenação de execução, que assegura a correta computação do *workflow* através das dependências entre tarefas e *DAG slices*. O principal diferencial do *middleware* com relação às soluções de grades e P2P existentes é o mecanismo de tolerância a faltas descentralizado. Através da utilização de *leasing*, o *middleware* é capaz de detectar a falha de *peers* (saída inesperada da rede), além de recuperar e retomar a execução do *workflow*. Na implementação do *middleware* foi utilizada Java como linguagem de programação e, para as funcionalidades P2P, a biblioteca JXTA, que oferece um conjunto de protocolos e mecanismos para a construção de aplicações P2P. De acordo com os resultados obtidos, o *middleware*, através do particionamento do DAG de entrada em *DAG slices*, consegue uma melhoria de até 45% no tempo de execução total. Outro importante resultado observado na execução do *middleware* diz respeito à eficiência na recuperação de falhas durante a computação do DAG. Os mecanismos de tolerância a falhas garantem ao middleware alta resiliência a falhas e propiciam uma rápida detecção e recuperação de execuções perdidas em decorrência de falhas.

Este trabalho está organizado da seguinte maneira: No Capítulo 2 são apresentados os conceitos básicos envolvidos nas redes P2P, grades computacionais e *workflows*, necessários ao entendimento do *middleware* proposto e seus objetivos. O Capítulo 3 aborda os trabalhos relacionados à computação de *workflows*, tanto em ambientes de grades com-

putacionais quanto P2P. No Capítulo 4 são tratadas questões referentes à arquitetura do *middleware*, como os algoritmos desenvolvidos, escalonamento e o mecanismo de tolerância a faltas. As questões relativas à implementação do *middleware*, como tecnologias utilizadas, mecanismos de comunicação e justificativas de projeto são apresentadas no Capítulo 5. Neste capítulo também são apresentados os resultados experimentais observados na execução do *middleware*. Finalmente, o Capítulo 6 aborda as considerações finais e possíveis trabalhos futuros.

Capítulo 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 [13].

Neste capítulo são apresentadas as principais características, semelhanças e diferenças entre redes P2P e grades computacionais, além das implicações da união das duas tecnologias, as chamadas grades P2P. Também são abordadas as questões referentes à execução e gerenciamento de *workflows* em ambos os ambientes.

2.1 Redes P2P

As redes P2P surgiram como um fenômeno social e tecnológico significativo nos últimos anos. Devido ao constante crescimento da Internet e a disponibilização cada vez maior de conectividade e largura de banda, as redes P2P continuam ganhando popularidade como uma forma barata de computação, armazenamento e compartilhamento de recursos [24]. Ao contrário dos sistemas distribuídos tradicionais, as redes P2P agregam um grande número de computadores, ou *peers*, que entram e saem da rede frequentemente e podem não possuir endereço de rede (IP) permanente.

Sendo estruturas relativamente recentes, várias definições têm sido utilizadas no intuito de caracterizar redes P2P. A definição apresentada em [26] caracteriza de forma simples e completa redes P2P como:

> Uma arquitetura de rede distribuída pode ser chamada de rede *Peer-to-Peer* se os participantes compartilham parte de seus próprios recursos de hardware (*i.e.*

> poder de processamento, capacidade de armazenamento, largura de banda de rede, impressoras, etc). Estes recursos compartilhados são necessários para prover serviços e conteúdo através da rede (*i.e.* compartilhamento de arquivo, compartilhamento de área de trabalho para colaboração, etc). Tais recursos são acessíveis por outros *peers* diretamente, sem a necessidade de passagem por entidades intermediárias. Os participantes de redes P2P são tanto provedores quanto consumidores de recursos.

A definição de redes P2P a seguir, apresentada em [18], reforça outra importante característica de tais redes, a autonomia dos *peers* participantes.

> Redes P2P referem-se a 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 (*peers*) sempre estão em total controle de seus recursos locais e podem impor ou modificar políticas de utilização destes recursos. 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 (*i.e.* cliente-servidor), as redes P2P se baseiam em papéis dinâmicos, 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. A Figura 2.1 ilustra a formação de uma rede *overlay* sobre a Internet.

Como mostra a Figura 2.1, as redes *overlay* podem ser vistas como redes virtuais em nível de aplicação, as quais possuem seus próprios mecanismos de roteamento e permitem que computadores individuais compartilhem informações e recursos diretamente, sem a necessidade de servidores dedicados. Estas características intrínsecas às redes *overlay* oferecem uma série de funcionalidades às aplicações P2P como arquitetura de roteamento em escala global, busca eficiente de dados, seleção de *peers* vizinhos, armazenamento redundante, detecção de permanência, autenticação, anonimato, alta escalabilidade e tolerância a falhas [19]. Com a ausência de controle centralizado e nenhuma (ou pequena) organização hierárquica, as redes *overlay* proporcionam às aplicações P2P serviços além dos oferecidos pelos tradicionais sistemas cliente-servidor, atribuindo aos participantes simetria de papéis, onde clientes também atuam como servidores.

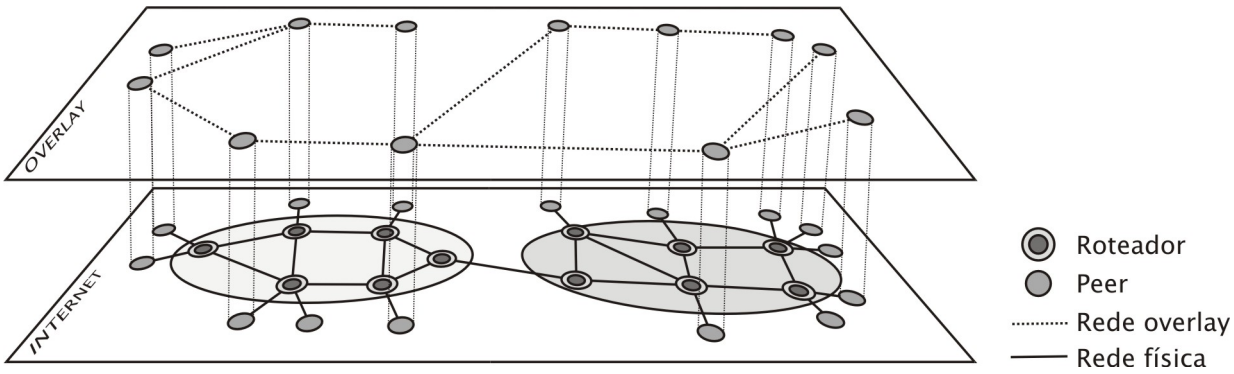


Figura 2.1: Rede *overlay* formada sobre a Internet

A topologia das redes *overlay* e os mecanismos de roteamento utilizados têm um impacto significante nas propriedades das aplicações como performance, confiabilidade, escalabilidade, e, em alguns casos, anonimato dos participantes. Além destes atributos, a topologia *overlay* também é responsável pela determinação dos custos de comunicação associados à execução das aplicações, tanto em *peers* individuais quanto em aglomerados de *peers*. As características das topologias *overlay* são um dos principais fatores utilizados na classificação das diferentes redes P2P. A classificação mais utilizada, e apresentada em [26], diferencia as redes P2P com relação ao determinismo com que é feita a inserção, busca e recuperação de conteúdo ao longo da rede. Em tal classificação, as redes P2P são agrupadas em dois grandes grupos: As redes P2P Estruturadas e Desestruturadas.

2.1.1 Redes P2P Estruturadas

Uma rede P2P é classificada como Estruturada quando se verifica o rígido controle da rede *overlay* associada, ou seja, dados e conteúdo são colocados não em *peer* aleatórios mas em posições específicas, o que tornará buscas e acessos posteriores mais eficientes. A implementação de redes estruturadas é geralmente realizada através de mecanismos denominados *Distributed Hash Table* (DHT), responsáveis em estabelecer a relação entre dados e localização ao longo da rede, isto é, a informação da localização dos dados é colocada, deterministicamente, em *peers* com identificadores que correspondem univocamente à chave que representa o dado. Redes P2P baseadas em DHT possuem a propriedade de atribuir identificadores de *peers* de maneira uniforme e randômica a um conjunto de *peers* em um grande espaço de identificadores. Aos dados são atribuídos identificadores únicos, chamados *chaves*, escolhidos no mesmo espaço de identificadores. As *chaves* são mapeadas por protocolos da rede *overlay* para um único *peer* ativo na rede. Através de primitivas do tipo *(put(chave, valor))* e *(valor = get(chave))*, os mecanismos de DHT suportam de forma escalável o armazenamento e a recuperação de dados na rede, que são

realizados através do roteamento de requisições aos *peers* que possuem identificadores correspondentes aos das chaves desejadas. Entretanto, devido aos altos custos de atualização das estruturas DHT, as redes P2P estruturadas são bastante sensíveis à intermitência de *peers* na rede, o que atua como fator limitante a uma utilização mais ampla das redes baseadas em DHT.

No universo das redes P2P estruturadas existe ainda uma sub-divisão baseada na ausência ou presença de entidades centrais. As chamadas Redes P2P Estruturadas Centralizadas, também conhecidas como Redes P2P Híbridas, são aquelas que possuem uma infra-estrutura centralizada responsável por alguns tipos de serviços exclusivos, como por exemplo, propiciar a entrada de *peers* na rede, realizar o processo de busca, indexar conteúdo, dentre outros. Neste tipo de rede P2P, verifica-se dois modelos de comunicação concomitantes: comunicação cliente-servidor entre *peers* e entidades centrais, e comunicação P2P entre *peers*. Como representantes desta subclasse destacam-se o Napster [3] e Emule [2], que utilizam servidores centrais para a manutenção e determinação da localização de conteúdo. A Figura 2.2 apresenta a estrutura do Napster, onde, além dos dois modelos de comunicação, verifica-se o agrupamento hierárquico dos participantes da rede entre *peers* e *super-peers*.

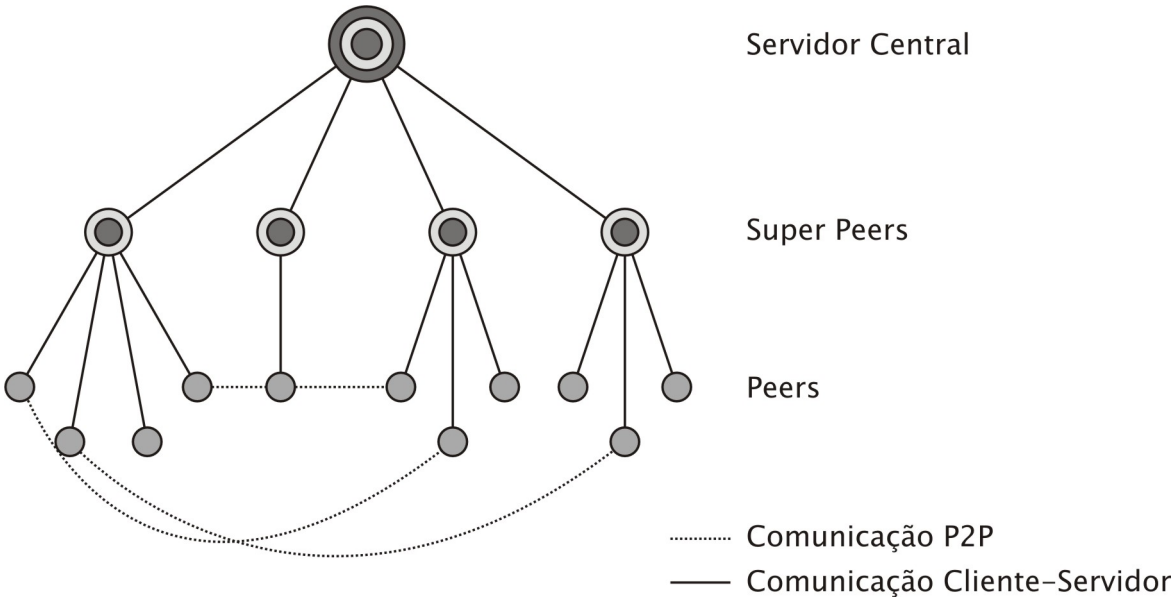


Figura 2.2: Napster - Hierarquia de *peers* além de comunicação cliente-servidor e P2P

Por outro lado, as Redes P2P Estruturadas Descentralizadas são aquelas baseadas em DHT e que não possuem uma entidade ou infra-estrutura centralizada. Nesta subclasse de redes P2P destacam-se o *Content Addressable Network* (CAN) [22], Tapestry [32] e Chord [28], que, além da utilização de DHT, possuem particionamento de espaço de identifica-

dores diferentes e adicionam o conceito de geometria de rede na determinação da relação entre dados e localização de conteúdo. A Figura 2.3 apresenta um exemplo da geometria e como são realizadas as buscas na CAN, uma rede baseada em DHT. Neste tipo de rede, os identificadores de dados são representados como um espaço cartesiano multi-dimensional virtual, no caso da Figura 2.3 bidimensional. Este espaço coordenado é puramente lógico, o qual é dinamicamente particionado entre todos os *peers* de forma que cada *peer* possua sua própria zona coordenada individual. Através de DHT, os *peers* da CAN mantêm os endereços IP e coordenadas virtuais das zonas de cada vizinho no espaço coordenado. De posse das coordenadas dos vizinhos, os *peers* roteiam as mensagens aos destinatários, encaminhando-as para o *peer* vizinho que está mais próximo das coordenadas destino.

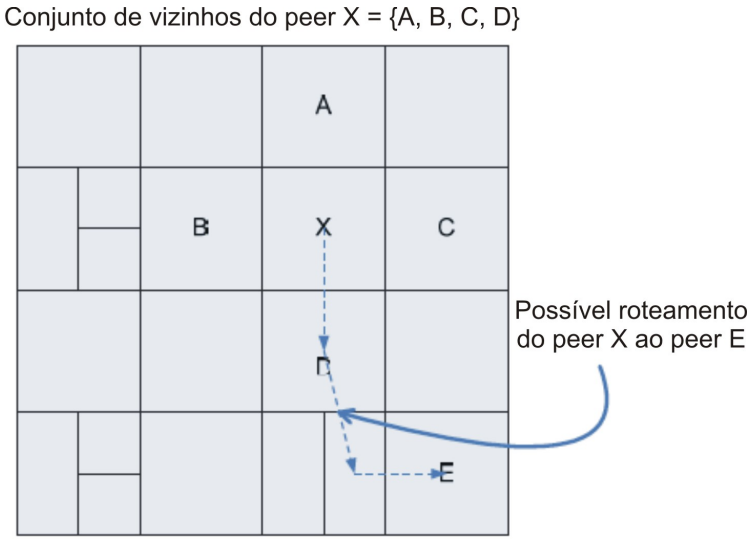


Figura 2.3: Geometria e busca na CAN

2.1.2 Redes P2P Desestruturadas

As Redes P2P Desestruturadas, também chamadas de Redes P2P Puras, são aquelas onde *peers* se juntam à rede sem nenhum conhecimento prévio da topologia da mesma e são organizados em grafos aleatórios, de forma hierárquica (*i.e. Super-Peers*) ou não. Redes desta classe utilizam inundação de mensagens (com escopo limitado) ou algoritmos randômicos como mecanismos de busca ao longo da *overlay*, além de serem bastante resilientes à alta intermitência de *peers* na rede. Quando um *peer* recebe uma determinada requisição, este envia uma lista de todo o conteúdo que satisfaz a requisição ao *peer* de origem. Claramente esta abordagem não é escalável, visto que à medida que novos *peers* se juntam à rede, a carga sobre cada *peer* e o número total de requisições crescem exponencialmente.

Enquanto que as técnicas baseadas em inundação de mensagens são eficazes para localizar itens altamente replicados, elas são bastante ineficientes na localização de itens raros espalhados na rede, cenário onde as redes estruturadas apresentam melhor desempenho. A principal e mais conhecida representante das redes P2P desestruturadas é a Gnutella, que utiliza inundação de mensagens na localização de arquivos na rede. A Figura 2.4 apresenta a disposição e o agrupamento de peers na rede Gnutella.

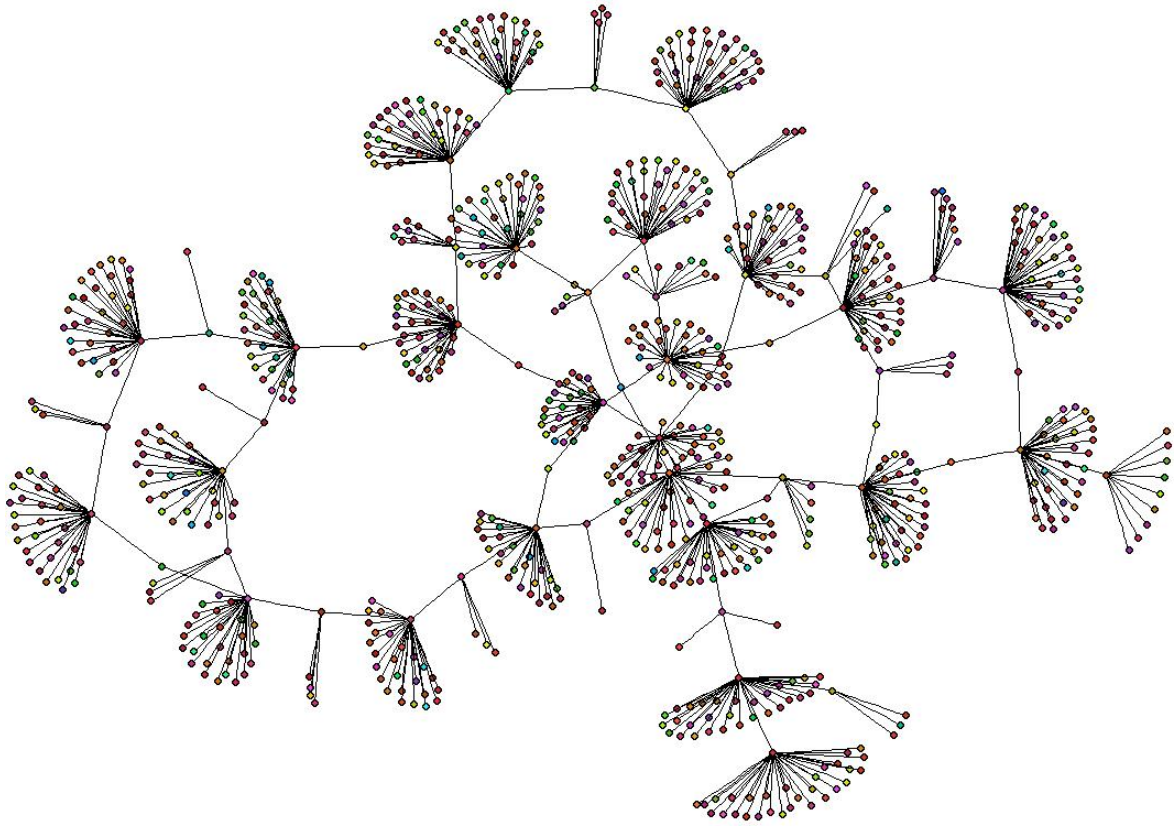


Figura 2.4: Disposição dos *peers* na rede Gnutella

A Figura 2.4 sugere que de acordo com o escopo das requisições de busca consegue-se atingir mais ou menos *peers* ao longo da rede através de inundações de mensagens. Na Figura 2.4, uma única requisição pode, por exemplo, alcançar de 500 a 2.000 computadores na rede.

2.2 Redes P2P X Grades Computacionais

Como quaisquer outras tecnologias concorrentes, existem semelhanças e diferenças entre redes P2P e grades computacionais. Pode-se ressaltar que estas se assemelham devido ao

fato de que o principal objetivo de ambas é prover mecanismos de organização de recursos compartilhados dentro de comunidades virtuais, além de utilizarem basicamente a mesma abordagem na solução destes problemas: a criação de uma rede *overlay* que coexiste com as diferentes estruturas físicas subjacentes.

Embora tais abordagens sejam utilizadas basicamente com o mesmo objetivo, existem diferenças significativas entre ambas, o que as tornam tecnologias concorrentes. Tanto em redes P2P quanto em grades computacionais existem também limitações cruciais a uma maior disseminação de tais tecnologias. Enquanto que grades computacionais possuem uma melhor gerência de infra-estrutura, estas geralmente não possuem mecanismos de tolerância a faltas eficientes, já redes P2P possuem melhores mecanismos de tolerância a faltas mas não usufruem de mecanismos de gerência de infra-estrutura [13].

Atualmente, outros fatores que também são utilizados na diferenciação de redes P2P e grades computacionais são o número de participantes e os tipos de serviços providos em cada ambiente. Em grades computacionais os serviços compartilhados são mais poderosos (supercomputadores, telescópios, etc), mais diversos, estão melhor conectados e são utilizados por comunidades de porte médio. Em tais ambientes existe a preocupação com a integração dos diferentes recursos, para que seja possível o fornecimento de serviços mais complexos e de níveis de qualidade de serviço (QoS) não triviais, dentro de um ambiente com o mínimo de confiança entre os participantes. Por outro lado, ambientes P2P lidam com muito mais participantes, mas oferecem serviços limitados e especializados, onde não há uma garantia mínima de QoS e nada se pode afirmar sobre laços de confiança entre os participantes [17].

Os papéis desempenhados entre os participantes dos dois modelos também podem ser diferentes. Enquanto que no modelo P2P todos os *peers* possuem as mesmas funcionalidades e características, em ambientes de grades computacionais pode haver distinção de papéis entre os participantes. Neste tipo de ambiente pode-se definir papéis específicos aos participantes, como coordenadores de grupos, gerenciadores e escalonadores de processos, além de nós dedicados exclusivamente à computação.

2.3 Grades P2P

Grades computacionais e redes P2P, ambas lidam com o gerenciamento e manutenção de recursos em ambientes distribuídos e são construídas através de estruturas *overlay* que operam independentemente de infra-estruturas institucionais. Apesar de suas diferenças, a aproximação destas tecnologias pode propiciar o desenvolvimento de novas aplicações e a solução de novos problemas. Neste contexto, surgem as Grades Computacionais P2P.

Com a fusão de modelos P2P e grades computacionais tornar-se-á possível a criação de uma tecnologia que agregue o que há de melhor em cada modelo. O modelo P2P

pode, por exemplo, ser utilizado para garantir escalabilidade em um ambiente de grades computacionais, ou seja, pode-se fazer uso da filosofia e de técnicas P2P para implementar grades descentralizadas e não hierárquicas (ausência de hierarquias de participantes). Tais técnicas podem auxiliar também na flexibilização da estrutura rígida das grades no que diz respeito à expansão da estrutura. Com a utilização de protocolos P2P que lidam com a intermitência de usuários poder-se-á aumentar de forma simples e transparente a estrutura da grade. Os ambientes de grades computacionais também seriam auxiliados com a incorporação de modelos distribuídos de busca de recursos, como os utilizados em redes P2P, o que eliminaria papéis centralizados na grade e conseqüentemente um único possível ponto de falha.

Embora o desenvolvimento de grades P2P seja vantajoso tanto do ponto de vista financeiro (uma única estrutura física) quanto do ponto de vista técnico (suporte para novas aplicações), ainda faltam muitos aspectos a serem tratados [30]. Ambos modelos necessitam de requisitos comuns como alta escalabilidade, auto-configuração e tolerância a faltas. Entretanto, algumas divergências filosóficas (e técnicas) envolvendo tais modelos dificultam uma maior integração dos mesmos, como por exemplo, a questão da confiança entre participantes. Enquanto em ambientes de grades computacionais necessita-se de um mínimo de confiança entre os participantes, em modelos P2P é priorizado o anonimato e autonomia dos *peers*, onde nenhum laço de confiança deve ser estabelecido. Sendo assim, para uma maior disseminação das grades computacionais P2P será necessária a solução de tais questões, onde cada modelo contribui com determinadas características e absorve outras do modelo concorrente [13].

2.4 *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 [14]. Durante as últimas décadas, as aplicações de *workflows* têm sido um dos tópicos de maior interesse na área de sistemas distribuídos, visto que o desenvolvimento de *workflows* pode resultar no aumento de flexibilidade e melhoria na estruturação e gerência de processos.

Assim como as redes P2P e grades computacionais, várias definições têm sido utilizadas no intuito de caracterizar *workflows*. A definição apresentada em [33], e que será utilizada no escopo deste trabalho, caracteriza *workflow*, de forma simples e objetiva como:

> O termo *workflow* pode ser expresso como a orquestração de um conjunto de tarefas, de acordo com regras bem definidas, a fim de se alcançar um objetivo sofisticado maior ou um processo de negócio.

Os esforços mais recentes com relação à utilização de *workflows* visam a descentralização das estruturas de coordenação e execução destes, visto que a grande maioria dos sistemas baseados em *workflows* utiliza a arquitetura cliente-servidor, a qual provê controle centralizado. Apesar de possuir benefícios, como a facilidade de monitoramento e utilização de *"thin clients"*, o gerenciamento centralizado da estrutura cliente-servidor não se mostra uma tecnologia ideal para suportar as novas aplicações de *workflow* distribuído, como as utilizadas em grades computacionais e mais recentemente em aplicações P2P [31]. Além de ser vulnerável, por possuir um único ponto de falha, a abordagem cliente-servidor subutiliza o potencial dos nós clientes, além de sobrecarregar os servidores.

Com o amadurecimento e a descentralização progressiva das aplicações baseadas em *workflows*, outras famílias de sistemas distribuídos têm feito larga utilização de tais estruturas, em especial as grades computacionais. Um *workflow* de grade também pode ser visto como um conjunto de tarefas, ou serviços de grade, executados em uma ordem bem definida a fim de alcançar um objetivo específico. A utilização de *workflows* também tem sido importante para as grades pelo fato de que estes permitem organizar os vários serviços da grade, combinando e propiciando aos usuários novos tipos de serviços, que poderão ser utilizados na solução de novos problemas complexos que venham a surgir. 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. Neste tipo de estrutura, cada nó do grafo representa uma tarefa computacional (com entrada e saída de dados) a ser processada, enquanto que as arestas direcionadas determinam o fluxo de execução e estabelecem as dependências entre as várias tarefas componentes do grafo. A Figura 2.5 apresenta um exemplo de DAG utilizado neste trabalho.

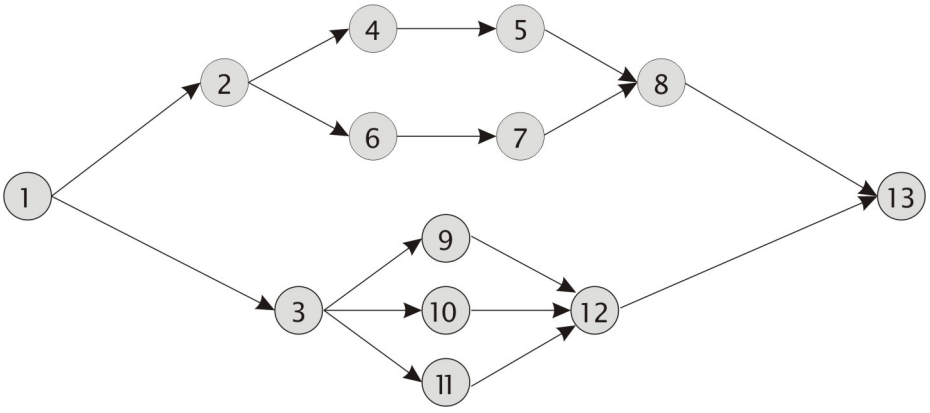


Figura 2.5: *Workflow* modelado como DAG

Na Figura 2.5, composta de treze tarefas, as arestas orientadas definem a ordem e a precedência na execução das tarefas. Por exemplo, a tarefa 1 deve ser executada com

sucesso para que, de posse dos resultados da tarefa 1, as tarefas 2 e 3 possam ser executadas e assim por diante. A tarefa 1, que não possui tarefas predecessoras, é chamada tarefa de entrada e é responsável pela inicialização dos processos e inserção dos dados necessários à computação do DAG. A tarefa 13, que não possui tarefas sucessoras, é denominada tarefa de saída. A tarefa de saída, geralmente, tem o objetivo de agrupar e apresentar os resultados obtidos ao longo da execução do DAG, no entanto, em alguns casos, também pode realizar alguma computação. Quando a tarefa de saída é executada com sucesso, conclui-se que a computação do DAG está terminada.

Capítulo 3

Trabalhos Relacionados

Neste capítulo são apresentadas aplicações e soluções relacionadas ao *middleware* proposto no presente trabalho. Dentre as diversas soluções existentes e para fins de comparação, no escopo deste capítulo, considera-se apenas aquelas que apresentam mecanismos de suporte e execução distribuída de *workflows*, que são os objetivos do *middleware* proposto. Diante da importância e do amadurecimento das aplicações baseadas em *workflows*, são abordadas soluções adotadas em ambientes computacionais diferentes, que têm feito uso de tais aplicações, como as grades, redes P2P e mais recentemente a web. Levando em consideração as decisões de projeto e implementação do *middleware*, também são apresentadas técnicas de *leasing* e de coordenação de procedimentos em ambientes P2P.

Workflows em Grades e na Web

Nas aplicações de grades computacionais, na maioria das vezes, o problema a ser solucionado é composto por um conjunto de pequenas tarefas que podem ser dependentes ou independentes umas das outras. Com a computação particionada em tarefas, um melhor balanceamento de carga pode ser realizado pelo processo de escalonamento. As aplicações que utilizam tarefas independentes, as chamadas *Bag-of-Tasks*, são aquelas aplicações paralelas onde não se verifica nenhuma dependência de dados e de execução entre as tarefas [11]. Atualmente, a maioria das soluções de grades e P2P suportam somente a execução isolada de tarefas, como as aplicações *Bag-of-Tasks*, mas não consideram a interdependência de tarefas como em um *workflow*. Esta deficiência restringe o desenvolvimento de melhores algoritmos de escalonamento, coordenação e recuperação de execução distribuída [10].

Para superar a limitação das aplicações *Bag-of-Tasks* e alcançar um maior poder de expressão na construção de processos estruturados, algumas aplicações têm feito uso de tarefas dependentes na solução de problemas computacionais complexos. A utilização de tarefas dependentes em tais aplicações possibilita mecanismos mais robustos e eficientes

de escalonamento e coordenação de execução, entretanto, tais funcionalidades introduzem um *overhead* maior no tempo de execução total do que nas aplicações *Bag-of-Tasks*.

Dentre as aplicações de grades computacionais que tratam a dependência entre tarefas, geralmente modeladas como *workflows*, pode-se citar a solução apresentada em [8], onde é implementado um sistema de gerenciamento e execução de *workflows*. Tal solução inclui um portal para usuários submeterem processos, além de serviços de gerência e escalonamento de *workflows*. Através de um mecanismo de escalonamento de tarefas baseado em predição de performance o sistema consegue realizar de forma eficiente a distribuição de tarefas do *workflow* para os recursos disponíveis na grade. Em [9] é apresentado um ambiente gráfico para a solução de problemas científicos, que provê uma ferramenta de construção e composição de *workflows*. Usuários compõem *workflows* graficamente através da combinação de componentes programáveis pré-desenvolvidos. Nesta solução também se verifica a existência de mecanismos responsáveis pela descoberta, escalonamento e coordenação de recursos da grade. Outra ferramenta gráfica para a construção e execução de *workflows* é proposta em [5], cujo objetivo visa o gerenciamento e análise de dados de *workflows* de aplicações biológicas. A abordagem utilizada em tal solução é baseada em modelos orientados a atores e permite modelagem hierárquica e estabelecimento de informações semânticas para as tarefas componentes do *workflow*. Um *middleware* para execução de processos estruturados em grades é apresentado em [10]. Neste *middleware*, onde a dependência de tarefas é modelada como DAG, utiliza-se controladores que coordenam a execução de fragmentos do DAG nos diversos recursos da grade. Papéis específicos como escalonamento e gerência de máquinas dentro de uma organização virtual são desempenhados por nós especializados.

Mais recentemente tem-se notado esforços com relação à integração de aplicações baseadas em *workflows* com a tecnologia de serviços web. Através dos protocolos e linguagens padronizadas oferecidos pelos serviços web, novas soluções de gerência e execução de *workflows* têm sido desenvolvidas, com o objetivo de tirar proveito da flexibilidade e interoperabilidade que tal tecnologia provê. Um representante desta nova classe de aplicação é apresentado em [21], que utiliza mecanismos de composição, coordenação e orquestração de serviços. Através da composição de mais de 1000 serviços disponíveis, usuários podem compor *workflows* para a solução de problemas relacionados à bio-informática.

Apesar dos mecanismos complexos e eficientes de gerência e execução de *workflows*, as soluções anteriormente apresentadas não abordam as falhas resultantes da intermitência do ambiente das grades computacionais, sendo assim, nenhum mecanismo de recuperação de execução é especificado. Além do mais, em tais soluções verifica-se a diferenciação de papéis entre os participantes da aplicação, onde nós especiais são responsáveis unicamente em prover determinado serviço, como por exemplo o escalonamento de tarefas. Esta característica pode ser vista como uma centralização de serviços fundamentais, o que

resulta em caso de falhas no comprometimento de todo o sistema.

Workflows em ambientes P2P

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 [16], 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 falta é tratado. Em [31] 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 [23], 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.

O trabalho apresentado em [30] mostra como a convergência de grades computacionais e aplicações P2P podem produzir soluções flexíveis e escaláveis, as chamadas grades P2P. Neste trabalho é realizada uma extensão de uma aplicação de P2P, através de um *framework*, para possibilitar a computação em grade na Internet. Através da integração de funcionalidades de soluções P2P, como a busca dinâmica de recursos, e de grades, como a publicação e registro de recursos, o trabalho proposto consegue bons resultados tanto na computação de *workflows* quanto na computação de problemas modelados como *Bag-of-Tasks*.

Leasing e Coordenação P2P

As técnicas de *leasing* têm sido cada vez mais utilizadas em aplicações que necessitam de um forte controle sobre o consumo e uso de recursos. Em [25] são abordadas várias questões sobre a utilização de *leasing* em sistemas distribuídos, visto que nenhuma análise detalhada sobre as variantes e os campos de aplicação de *leasing* foi realizada anteriormente. Em tal trabalho é apresentada uma classificação sistemática das possíveis utilizações de *leasing*. Também são propostos, através de algoritmos, possíveis mecanismos de concessão, gerenciamento e renovação de *leases*, mecanismos estes que permitem decisões mais complexas e eficientes em sistemas que fazem uso de *leasing*.

De acordo com a descentralização dos ambientes P2P, aplicações construídas sobre

tais ambientes têm de lidar com sérios problemas de coordenação, seja de execução ou de comunicação. Em [18] são apresentadas soluções para propriciar melhores mecanismos de coordenação em ambientes P2P. No trabalho é apresentado um *framework/middleware* para suportar a execução coordenada de aplicações distribuídas em ambientes P2P. Além do mais, são propostos modelos e linguagens de coordenação para possibilitar uma melhor orquestração de procedimentos distribuídos na rede P2P.

Capítulo 4

Arquitetura e Algoritmos do *Middleware*

Neste trabalho é proposto um *middleware* cujo objetivo principal visa a utilização da flexibilidade e do dinamismo que os ambientes P2P oferecem, a fim de suportar computação distribuída e cooperativa. A definição apresentada em [29], e que será utilizada ao longo deste trabalho, caracteriza *middleware* da seguinte maneira:

> *Middleware* representa uma camada de software intermediária entre o sistema operacional e a aplicação, o qual implementa funcionalidades adicionais para suportar o desenvolvimento de novas soluções distribuídas.

Com o desenvolvimento do *middleware*, usuários poderão usufruir dos recursos computacionais disponibilizados pelos participantes da rede. Além disso, o *middleware*, diferentemente da maioria das soluções de grades computacionais e P2P, suporta a execução distribuída e coordenada de *workflows*, gerenciando e coordenando a execução de um conjunto de tarefas dependentes de acordo com regras bem definidas. No decorrer deste capítulo são abordadas as questões referentes à arquitetura, mecanismos e algoritmos utilizados na implementação do *middleware*, que possui como principais características a descentralização de procedimentos e alta resiliência a faltas.

4.1 Análise de Requisitos

A fase de análise de requisitos da arquitetura do *middleware* tem como objetivo a identificação das funcionalidades necessárias a fim de prover uma infraestrutura P2P para suportar computação distribuída e cooperativa. Para a identificação destas funcionalidades é necessário primeiramente situar o *middleware* tanto como uma aplicação P2P quanto

uma aplicação de computação distribuída. A combinação dos aspectos fundamentais de cada uma destas classes de aplicação propicia o desenvolvimento de aplicações altamente dinâmicas, flexíveis e resilientes a faltas.

Como um representante da classe de aplicações P2P típicas, no *middleware*, todos os nós (*peers*) participantes da rede possuem os mesmos papéis e funcionalidades - todos os *peers* agem como cliente e servidor. Entretanto, a prática concomitante destes dois papéis não é o suficiente para caracterizar o *middleware* como uma aplicação P2P. Um outro aspecto importante na caracterização de aplicações P2P é a criação de redes *overlay*, que, no *middleware*, possibilita a entrada de novos *peers* na rede, busca, comunicação e publicação de recursos por parte dos *peers*.

Além de uma aplicação P2P, o *middleware* também se enquadra como uma aplicação de computação distribuída, devendo para isso, possuir características e mecanismos essenciais a este tipo de aplicação. Como exemplo destas características pode-se citar o escalonamento, que consiste nas técnicas e procedimentos de distribuição de tarefas e recursos; coordenação de execução, que provê mecanismos responsáveis em assegurar a correta execução da aplicação em ambientes distribuídos; e tolerância a faltas, que propicia às aplicações alta resiliência na ocorrência de faltas (rede, hardware e software) e permite, na maioria das soluções, recuperação de execução.

4.2 Arquitetura do *Middleware*

No desenvolvimento da arquitetura do *middleware* foram levados em consideração aspectos tanto de aplicações P2P quanto de aplicações de computação distribuída. Na Figura 4.1, através de um diagrama de blocos, são apresentados os módulos constituintes do *middleware*, identificados durante a fase de análise de requisitos, assim como seus relacionamentos e suas naturezas com relação às classes de aplicações.

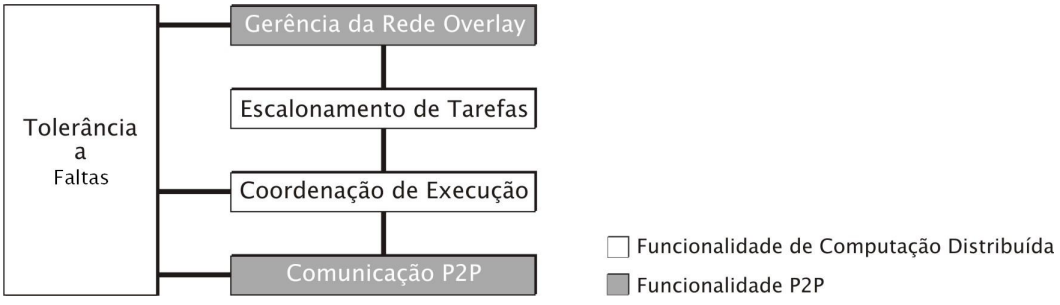


Figura 4.1: Módulos que compõem o *middleware*

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. Neste módulo também são tratados os aspectos envolvidos na criação, atualização e auto-organização da rede *overlay*. Através destes procedimentos, um *peer* que deseja se juntar à rede se torna visível aos demais participantes e pode publicar recursos a serem compartilhados, tornando-se a partir de então um novo membro da rede e podendo ser utilizado em computações futuras no *middleware*.

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. Neste módulo são especificados os procedimentos, requisitos e parâmetros envolvidos na comunicação entre *peers*, além das questões de serialização, transferência de dados e tarefas entre participantes de uma determinada computação. Por se tratarem de aspectos basicamente atrelados às tecnologias utilizadas, ambos os módulos, **Gerência de Rede *Overlay*** e **Comunicação P2P**, são tratados em mais profundidade no Capítulo 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 4.3. Já o módulo de **Tolerância a Faltas** encapsula os mecanismos que propiciam ao *middleware* alta resiliência a faltas e recuperação de execução. Tal módulo é discutido na Seção 4.4.

4.3 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. Neste tipo de problema, a entidade que manipula a aplicação e atribui cada um de seus componentes a um recurso é chamada de **escalonador**. O principal objetivo do escalonador é minimizar o tempo de execução das aplicações, escalonando seus componentes de forma a maximizar o paralelismo na execução e minimizar a comunicação, conseqüentemente otimizando a utilização dos recursos [6]. Em sistemas dinâmicos e voláteis como as grades computacionais e aplicações P2P, um dado recurso pode ficar indisponível a qualquer momento, o que dificulta e torna cada vez mais complexo o processo de escalonamento em tais ambientes. Além do mais, a natureza heterogênea dos recursos destes ambientes (redes, hardwares e softwares diferentes) também deve ser levada em consideração a fim de se alcançar um bom escalonamento.

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* participantes da rede. 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 precedências de execução destas tarefas sejam respeitadas, obedecendo assim as regras (dependências) impostas pelo *workflow*. Tais regras de gerência de dependência e controle de execução das tarefas são definidas no módulo **Coordenação de Execução**, que, juntamente com os mecanismos de escalonamento, garante a execução correta, distribuída e coordenada do *workflow*.

No escopo deste trabalho, como discutido na Seção 2.4, 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. Tais custos podem ser definidos através de uma métrica qualquer, como por exemplo, a quantidade aproximada de FLOPS que cada tarefa necessita para ser executada. 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, ou seja, como o particionamento do DAG em *DAG slices* gera um aninhamento destas estruturas (*DAG slices* pais e filhos), os *DAG slices* de mesmo nível (por exemplo, filhos de um mesmo *DAG slice* pai) devem ser dispostos de forma a não haver dependência entre eles. Dentro de cada *DAG slice*, as tarefas são identificadas e numeradas univocamente em ordem crescente, de acordo com a relação de dependência entre elas, ou seja, uma determinada tarefa não pode possuir identificador maior que sua sucessora (arestas orientadas). Tarefas paralelas também são consideradas *DAG slices*. A Figura 4.2 apresenta um exemplo de DAG de entrada do *middleware*.

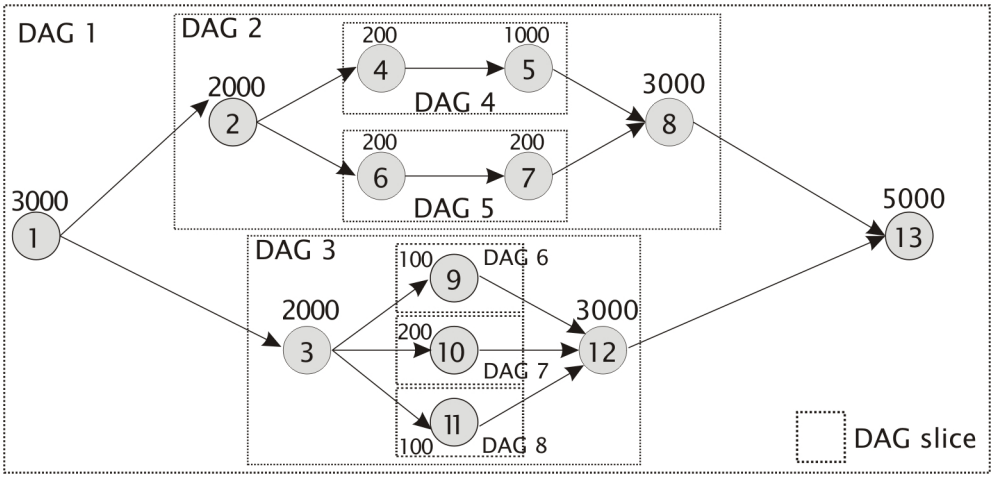


Figura 4.2: Exemplo de DAG de entrada

Na Figura 4.2 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. No *middleware*, para assegurar a correta submissão do DAG, recepção, organização e apresentação dos resultados, tanto a tarefa de entrada quanto a tarefa de saída são de responsabilidade do *peer* que submete o DAG, cabendo exclusivamente a este *peer* a computação de tais tarefas.

Diferentemente da maioria das aplicações baseadas em *workflow*, no DAG de entrada do *middleware* não são levados em consideração os custos de comunicação entre as tarefas (custo das arestas). Tal decisão se justifica pela dificuldade em medir o custo de comunicação nas redes *overlay*, nas quais as aplicações P2P são construídas. Por exemplo, dois *peers* são vizinhos diretos na *overlay* mas podem estar separados por dezenas de nós intermediários na Internet, o que resulta num alto custo de comunicação. Por outro lado, um terceiro *peer* pode estar distante na *overlay* e estar na mesma rede local que os primeiros, o que sugere um baixo custo de comunicação. Outra justificativa para a não adoção de custos de comunicação no DAG de entrada reside no fato de que o *middleware* foi inicialmente desenvolvido tendo em vista aplicações do tipo *CPU intensive*, ou seja, aplicações que necessitam principalmente de grande volume de processamento. Além do mais, principalmente em aplicações P2P, nada se pode dizer sobre a qualidade dos *links* sobre os quais os dados são enviados. A análise da quantidade de dados a ser transferida entre *peers*, do estado dos *links* que os conectam e consequentemente a introdução de custos de comunicação no DAG de entrada pode otimizar o processo de escalonamento são propostos como trabalhos futuros.

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 e possibilitar a execução do DAG. Logo, o processo de escalonamento no *middleware* leva em consideração apenas o poder de processamento que cada *peer* compartilha, valor este definido quando os *peers* se juntam à rede, e o custo computacional das tarefas envolvidas na computação. Como trabalho futuro, a fim de otimizar o processo de escalonamento, poder-se-á introduzir os custos de comunicação entre tarefas do DAG como uma variável adicional, possibilitando uma melhor e mais eficiente distribuição das tarefas entre os participantes da rede.

Como ressaltado no início do capítulo, no *middleware* todos os *peers* possuem as mesmas características e funcionalidades. Sendo assim, todos os *peers* envolvidos na computação de um dado 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* participantes da rede também podem submeter DAGs para serem executados. Ao longo deste trabalho, 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. De acordo com as características do processo representado pelo DAG, cabe também ao *DAG owner* particioná-lo em *DAG slices*, agrupando convenientemente as tarefas relacionadas a fim de obter um maior paralelismo na execução e um menor número de dependências entre *DAG slices*.

Algoritmo de Escalonamento

Nesta seção é apresentado o algoritmo de escalonamento utilizado no *middleware*, responsável em distribuir as tarefas entre os *peers* participantes e controlar e coordenar as dependências impostas pelo DAG. Entretanto, para o entendimento do algoritmo é necessário primeiramente especificar a forma com que as tarefas e *DAG slices* são modelados no *middleware*, a qual é apresentada na Figura 4.3.

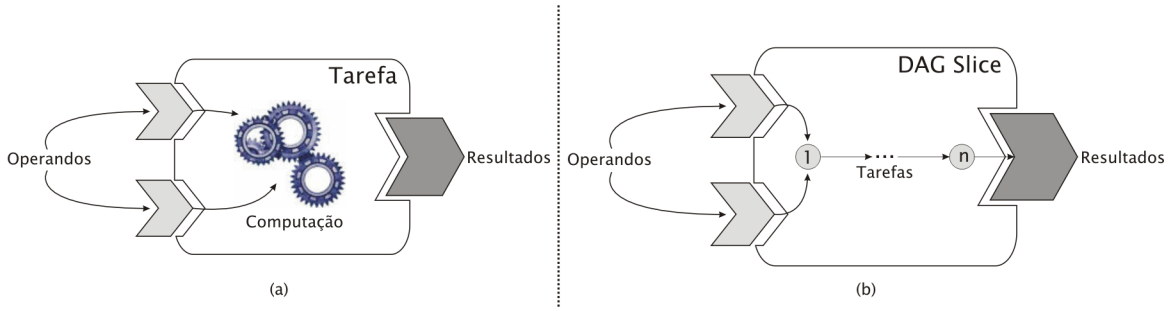


Figura 4.3: Organização de tarefas e *DAG slices*

Como mostra a Figura 4.3, tanto as tarefas quantos os *DAG slices* possuem organizações semelhantes. No caso das tarefas, Figura 4.3 - (a), estas possuem um conjunto de operandos, os quais são necessários para realizar a computação e conseqüentemente gerar resultados. Estes operandos são os resultados das execuções das tarefas e/ou *DAG slices* predecessores, ou seja, a saída de uma tarefa e/ou *DAG slice* serve como entrada para as suas tarefas sucessoras. Já no caso dos *DAG slices*, Figura 4.3 - (b), assim como nas tarefas, o conjunto de operandos é formado pelos resultados de tarefas e/ou *DAG slices* predecessores, no entanto, no início do processo de escalonamento, tais operandos são transferidos para os operandos da primeira tarefa do *DAG slice*, que é de fato a entidade que realiza computação e inicializa a execução do *DAG slice*. Já os resultados do *DAG slice* são os resultados produzidos na execução de sua última tarefa.

Após a definição da estrutura das tarefas e *DAG slices*, no Algoritmo 1 é apresentado o algoritmo ***DAG Slice Distribution***, que é executado em todos os *peers* que realizam alguma computação no *middleware*.

Algoritmo 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: se operandos de S ≠ vazio então
4:    operandos de t = operandos de S
5:    operandos de S = vazio
6: enquanto P é capaz de executar t faça {P possui poder de processamento maior que
   o custo de t}
7:    enquanto P não receber todos os operandos de t faça
8:       fica bloqueado até receber os operandos de t
9:    se t é a última tarefa de S então
10:      execute t e marca S como executado
11:      envia resultados de t para Q
12:      FINISH
13:   execute t
14:   propague resultados de t para tarefas e/ou DAG slice sucessores
15:   para cada DAG slice sucessor, S_t, de t faça
16:      operandos de S_t = resultados de t
17:      heavierTask = tarefa de maior custo de S_t
18:      se existe peer U_t com processamento ≥ custo de heavierTask então
19:         envia S_t para U_t
20:      senão
21:         envia S_t para peer U_t com processamento ≥ custo da tarefa de menor ID de S_t
22:      fica bloqueado até receber resultados de U
23:   marca t como executada
24:   t = tarefa com menor ID de S
```

A seqüência de procedimentos apresentada no Algoritmo 1 pode ser melhor explicada, de forma textual, da seguinte maneira:

> Um dado *peer* P recebe um *DAG slice* S de um *peer* Q (linha 1). Q possui o *DAG slice* pai de S, e durante o processo de escalonamento realizado em Q, este escolhe P para computar S, visto que P possui poder de processamento para tal. No momento do recebimento de S, P armazena a tarefa de menor identificador (tarefas numeradas univocamente em ordem crescente) de S, no caso t (linha 2). Sendo S um *DAG slice* filho, S conterá o resultado da execução de tarefas e/ou *DAG slices* predecessores, armazenados no seu conjunto de operandos, resultados estes que serão utilizados como operandos da primeira tarefa de S, ou seja, de t (linhas 3 e 4). 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 6), diminuindo conseqüentemente o tempo total de execução do *DAG slice* - resultado de uma melhor utilização do poder de processamento compartilhado pelos *peers* da rede. O algoritmo também mostra um entrave na utilização de tarefas dependentes - estas bloqueiam a execução do DAG, ou seja, enquanto uma tarefa não receber todos os operandos necessários de suas tarefas e/ou *DAG slices* predecessores, ela não pode retomar a computação (linhas 7 e 8).

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 9 a 12), entretanto, P pode ainda ser escolhido para a computação de *DAG slices* posteriores. 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 13 e 14).

Para cada *DAG slice* sucessor de t, no caso S_t, P inicializa uma *thread* para gerenciar a execução de S_t (linha 15). Os resultados da execução de t são transferidos para os operandos de S_t (linha 16). Esta ação é necessária a fim de possibilitar a execução de S_t imediatamente após seu recebimento por parte do *peer* responsável pela computação de S_t, ainda a ser escolhido pelo processo de escalonamento. Logo, quando um *peer* recebe um dado *DAG slice* para computação, o *peer* já dispõe de todos os pré-requisitos para inicializar a execução do *DAG slice*.

Com o intuito de escolher os melhores *peers* e minimizar os custos de comunicação e de transferência de dados, S_t é enviado a um *peer* U_t (se houver) tal que, U_t possua poder de processamento maior ou igual ao custo da tarefa de maior custo computacional de S_t. Se não houver um *peer* com tal poder de processamento, S_t é enviado a um *peer* U_t que possua capacidade de processar pelo menos a primeira tarefa de S_t (linhas 17 a 21).

Finalmente t é marcada como executada com sucesso e o processo se repete, atribuindo-se a t a próxima tarefa de S com menor identificador (linhas 23 e 24). As marcações realizadas nas linhas 10 e 23 são necessárias para suportar os mecanismos de recuperação de execução em decorrência de faltas, e serão discutidas na Seção 4.4.

Um exemplo de escalonamento é apresentado na Figura 4.4. Neste caso, o DAG de entrada utilizado é o da Figura 4.2.

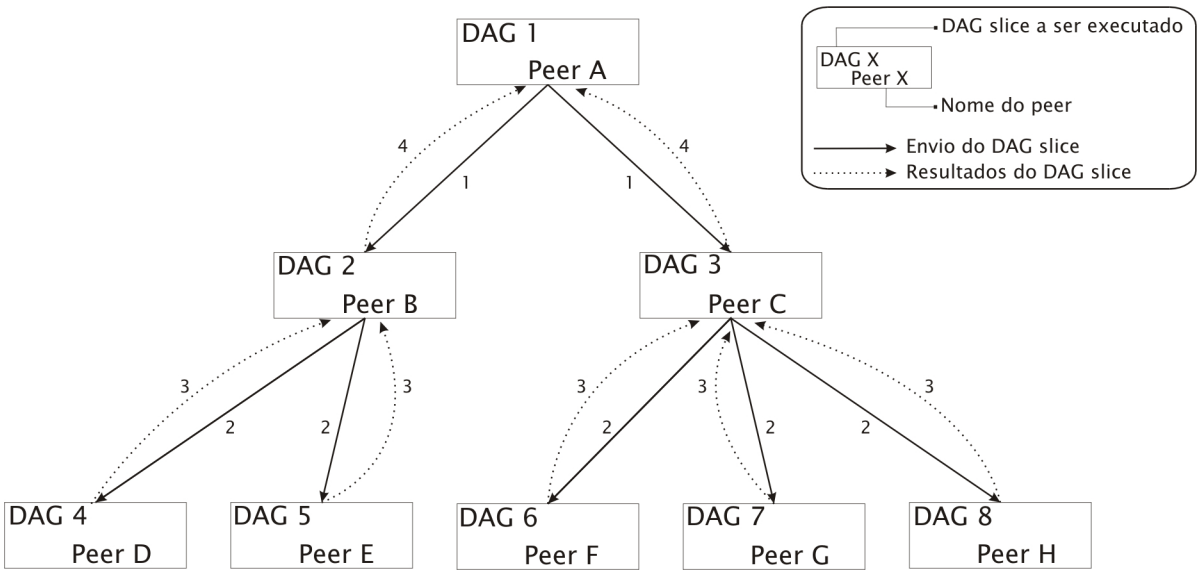


Figura 4.4: Árvore de execução gerada após o escalonamento

Como mostra a Figura 4.4, 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. A estrutura e formação da árvore de execução também são de fundamental importância nos mecanismos de tolerância a faltas do *middleware*, que a utiliza na determinação dos estados dos *peers* envolvidos em uma computação. A Figura 4.4 também mostra a ordem com que o envio de *DAG slices* e seus resultados devem ser realizados.

Como discutido na Seção 4.3, o critério utilizado na escolha dos *peers* para a computação de um dado *DAG slice* leva em consideração o poder de processamento compartilhado pelos *peers* na rede e o custo computacional dos *DAG slices* e tarefas. Sendo assim, os *peers* escolhidos no exemplo da Figura 4.4 possuem processamento suficiente para executarem seus respectivos *DAG slices*. Além do mais, cada *peer* pode conhecer um conjunto diferente de *peers* da rede em um dado momento, característica esta que confere um caráter não determinístico na escolha dos *peers*.

De acordo com a arquitetura modular do *middleware*, algoritmos de escalonamento mais robustos (e especializados) podem ser utilizados a fim de otimizar a distribuição de tarefas entre os participantes da rede. Tal modularidade confere ao *middleware* um caráter extensível com relação aos mecanismos de escalonamento. Desta forma, diferentes algoritmos podem, por exemplo, considerar diferentes conjuntos de informações durante a realização do escalonamento, de acordo com os objetivos de cada aplicação.

4.4 Tolerância a Faltas

Tolerância a faltas é a propriedade que permite que sistemas (em geral computacionais) continuem a operar adequadamente mesmo após a ocorrência de faltas em alguns de seus componentes. A tolerância a faltas é propriedade inerente em sistemas de alta disponibilidade ou aplicações críticas como as dedicadas à medicina e as aplicações de tempo real [20]. Entretanto, aplicações de computação distribuída têm utilizado cada vez mais mecanismos e estratégias que assegurem a continuidade de suas execuções em decorrência de faltas, visto que tais aplicações são suscetíveis a faltas de diversas naturezas.

O principal diferencial deste trabalho com relação às soluções existentes de grades computacionais e aplicações P2P são os mecanismos de tolerância a faltas utilizados para garantir a correta execução do DAG e retomar execuções perdidas por causa de faltas. Visto que, na maioria das aplicações de grades e P2P, os mecanismos de tolerância são implementados de forma centralizada, no *middleware*, estes são realizados de forma completamente descentralizada, característica esta que provê flexibilidade, alta resiliência a faltas e ausência de entidades centrais (pontos únicos de falha).

No *middleware*, 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. Após o tratamento da falta, cabe também ao módulo de Tolerância a Faltas recuperar, se possível, as computações e resultados perdidos em decorrência da falta. Quando tal recuperação não é possível, um novo escalonamento é requisitado pelos mecanismos de tolerância para que a computação do DAG possa ser retomada.

No escopo deste trabalho, as faltas consideradas são aquelas oriundas da intermitência dos ambiente P2P, ou seja, a alta volatilidade de tais ambientes, que concede aos *peers* participantes a autonomia de entrar e sair da rede quando desejarem. Em outras palavras, estas faltas ocorrem quando um determinado *peer*, responsável por alguma computação em andamento do DAG, sai de maneira inesperada da rede, ou por vontade própria ou por alguma falha de rede, hardware ou software, o que resulta na perda das informações processadas e em um desarranjo na estrutura da árvore de execução do DAG.

4.4.1 *Leasing*

Como abordado na Seção 4.3, os mecanismos de tolerância a faltas são construídos tendo como base a árvore de execução gerada pelo processo de escalonamento. Através da estrutura da árvore, o *middleware* trata de forma eficiente importantes aspectos de tolerância a faltas em sistemas distribuídos: a determinação dos estados dos nós da rede e recuperação de execução.

Com base na árvore de execução e através de mecanismos de *leasing*, o *middleware* pode determinar rapidamente quando um determinado *peer* deixa a rede de forma inesperada.

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 [25]. Geralmente, os sistemas que fazem uso de *leasing* são aqueles onde a utilização de recursos precisa ser controlada, para permitir a liberação de tempos em tempos de recursos não utilizados ou ociosos.

Sistemas escaláveis e robustos devem gerenciar seus recursos de forma eficiente. Um recurso pode ser de várias naturezas distintas, incluindo serviços locais ou distribuídos, sessões de banco de dados, dentre outros. Em um caso típico de uso, um usuário recupera a interface de um provedor de recursos e então solicita a este provedor o acesso a um ou mais recursos. Assumindo que o provedor garanta o acesso ao recurso, o usuário pode então começar a utilizá-lo. Entretanto, após um determinado período de tempo, o usuário pode não mais necessitar destes recursos. A menos que o usuário libere os recursos após terminar explicitamente seu relacionamento com o provedor, os recursos não utilizados continuariam sendo desnecessariamente consumidos. Este consumo de recursos desorganizado pode resultar na degradação de desempenho tanto do usuário quanto do provedor dos recursos. Além do mais, pode afetar também a disponibilidade de recursos para outros usuários.

Em sistemas onde usuários e provedores estão distribuídos, ainda é possível que após um determinado período de tempo o provedor possa falhar (sair da rede) ou possa não mais oferecer um dado recurso. Se os usuários não forem explicitamente notificados sobre a indisponibilidade destes recursos, tais usuários podem continuar a manter referências inválidas a recursos que não existem mais.

A solução destes problemas através de *leasing* consiste na introdução de ***leases*** (entenda-se permissões) em todos os recursos mantidos por um usuário. A *lease* é concedida pelo provedor dos recursos e obtida pelo usuário que deseja o recurso. As *leases* especificam um período de tempo no qual o usuário pode utilizar o recurso. Uma vez que este período de tempo expire, a *lease* é dita expirada e os recursos correspondentes mantidos pelo usuário são liberados. Antes da *lease* expirar, o usuário pode tentar renová-la com o provedor do recurso. Se a *lease* é renovada o recurso correspondente continua disponível ao usuário.

A utilização de *leasing* no *middleware* tem como objetivo auxiliar na determinação dos estados dos *peers* participantes da computação de um determinado DAG. O esquema de *leasing* nos mecanismos de tolerância a faltas permite aos *peers* concederem *leases* a outros *peers*. Neste caso, os recursos em questão são a utilização dos próprios *peers*, ou seja, um dado *peer* "servidor"concede sua utilização a um *peer* "cliente". Estas *leases* de utilização 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, ou por conseqüência de uma falha ou se desligou da rede por vontade própria. Considerando que

o número de pedidos e concessões de *leases* cresce proporcionalmente com o aumento do número de *peers*, e, devido ao alto custo de comunicação nos ambientes P2P, a utilização de *leases* é realizada apenas na direção ***bottom-up*** na árvore de execução do DAG. Por exemplo, na Figura 4.4, os *peers* D e E concedem suas utilizações ao *peer* B através de *leases* (*peers* D e E desempenham papéis de servidores para o *peer* B). Se os *peers* D e E ainda não tiverem terminado a execução de seus respectivos *DAG slices*, e, por algum motivo as *leases* do *peer* B não forem renovadas, tanto o *peer* D quanto o *peer* E saberão que o *peer* B saiu da rede, o que desconectará a árvore de execução.

O *middleware* também faz uso do comportamento bloqueante da execução de tarefas dependentes para prover uma maneira "natural"de saber se um *peer* está ativo na árvore de execução (direção ***top-down***): 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. Este esquema bloqueante é semelhante às chamadas RPC (*Remote Procedure Call*), onde cliente e servidor permanecem conectados (e cliente bloqueado) até que a chamada remota seja atendida, executada e finalizada pelo servidor. Logo, em ambas as direções na árvore de execução do DAG é possível determinar os estados dos *peers* da rede.

4.4.2 Recuperação de Execução

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, como apresentado no Algoritmo 1. 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 *peer* que submete o DAG para computação no *middleware*, o *DAG owner*, nunca falha. A introdução desta exigência reside no fato de que, enquanto o DAG não for computado, o principal interessado na completude de sua execução, o *DAG owner*, não se desligará da rede, aguardando o término do processo. Além do mais, como raiz da árvore de execução, o *DAG owner* é a única entidade que possui conhecimento global do DAG em questão. Esta visão global é necessária para a recuperação e/ou reescalonamento de qualquer componente do DAG. Se, por motivo de falha em rede, hardware ou software, o *DAG owner* sair da rede, todo o processo de execução do DAG será abortado, pois, mesmo que todos os *DAG slices* tenham sido distribuídos antes da falta

do *DAG owner*, a tarefa de saída do DAG, responsável pelo agrupamento e processamento final do DAG e que é de responsabilidade do *DAG onwer*, foi perdida, impossibilitando a completude da execução do DAG.

A visão global do DAG que somente o respectivo *DAG owner* possui provê ao *middleware* uma característica altamente desejável em aplicações distribuídas: gerência descentralizada. Isto quer dizer que o *middleware* suporta a execução simultânea de mais de um DAG, submetidos por diferentes *peers* (diferentes *DAG owners*). Sendo assim, cada *DAG owner* gerencia independentemente a execução de seu respectivo DAG.

4.4.3 Algoritmos de Recuperação de Execução

Na ocorrência de faltas em *peers* participantes da 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:

- O *peer* que falha é um nó folha na árvore de execução ou
- O *peer* que falha é um nó intermediário na árvore de execução.

No primeiro caso, o mais simples, onde a falta ocorre em um *peer* folha na árvore de execução, a computação realizada até então por este *peer* não pode mais ser recuperada. Quando detectada este tipo de falta, através do cancelamento abrupto da conexão (RPC) entre os *peers* pai e filho, o *peer* pai localizado imediatamente acima na árvore de execução do *peer* que falhou, simplesmente requisita um novo escalonamento do *DAG slice*, atribuindo-o a outro *peer* na rede.

No segundo caso, onde o *peer* que falha é um nó intermediário na árvore de execução, procedimentos mais complexos devem ser realizados para assegurar a retomada correta da execução do DAG, visto que a árvore de execução é desconectada neste cenário. Além do mais, tanto o *peer* pai quanto o(s) *peer*(s) filho(s) de um *peer* que falhou devem se organizar a fim de identificar e tratar a falta. Neste cenário, a falta 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). Para o melhor entendimento do procedimento de recuperação de execução neste caso, são apresentados nos Algoritmos 2 e 3 as atividades realizadas pelos *peers* clientes e servidores, respectivamente, com o objetivo de assegurar a coesão e correção na execução do DAG.

A Figura 4.5 apresenta, passo-a-passo, todos os procedimentos envolvidos num cenário onde um *peer* intermediário na árvore de execução falha. Neste caso, considera-se novamente o DAG da Figura 4.2 e a árvore de execução correspondente, apresentada na Figura

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

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

Algoritmo 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: se lease expira(cliente saiu ou falhos) então
5:    avisa peer pai de P
6:    entrega resultados de S para peer pai de P
```

4.4. A cada procedimento relativo à recuperação de execução, são citadas as linhas correspondentes dos algoritmos em questão. Por simplicidade, somente os *peers* envolvidos no processo são apresentados.

Para a compreensão do processo de recuperação de execução, os Algoritmos 2 e 3, juntamente com a Figura 4.5 são explicados, passo-a-passo, de forma textual como se segue:

1. Inicialmente o **Peer A**, que representa o *DAG owner* no exemplo, submete o DAG para execução no *middleware* (Figura 4.5 - (a));

2. **Peer A** executa a **tarefa 1** e adiciona seus resultados nos operandos dos *DAG slices* sucessores da **tarefa 1**, no caso, **DAG 2** e **DAG 3** (Figura 4.5 - (b));

3. **Peer A** escolhe **Peer B** para computar **DAG 2**. **Peer A** fica bloqueado até a conclusão da execução do **DAG 2** por **Peer B**. Apesar de não mostrado na figura, **Peer A** também tem de esperar a computação de **DAG 3** realizada por **Peer C** (Figura 4.5 - (c));

4. **Peer B** executa **tarefa 2** e, como **tarefa 2** possui outros 2 *DAG slices* posteriores, adiciona os resultados de **tarefa 2** nos operandos de **DAG 4** e **DAG 5** (Figura 4.5 - (d));

5. **Peer B**, após o processo de escalonamento, escolhe **Peer D** e **Peer E** para computarem **DAG 4** e **DAG 5** respectivamente. **Peer B** fica bloqueado até o retorno dos resultados de **DAG 4** e **DAG 5** (Figura 4.5 - (e));

6. **Peer D** executa **tarefa 4** e transfere seus resultados para o conjunto de operandos de **tarefa 5**. Da mesma forma, **Peer E** executa **tarefa 6** e repassa seus resultados para a tarefa posterior, ou seja, **tarefa 7** (Figura 4.5 - (f));

7. **Peer B** falha (Algoritmo 2 - linha 4). Logo, a conexão de **Peer A** com **Peer B** é cancelada e as *leases* de **Peer B** não serão renovadas com **Peer D** e **Peer E** (Algoritmo 3 - linha 4). Enquanto isso, **Peer D** executa **tarefa 5** e **Peer E** executa **tarefa 7**, terminando ambos a execução de seus respectivos *DAG slices* (Figura 4.5 - (g));

8. **Peer A** aguarda aviso e/ou resultados dos *peers* filhos de **Peer B** (Algoritmo 2 - linha 6). Este aviso é necessário para saber se o *peer* que falhou saiu da rede antes ou depois de distribuir seus *DAG slices* filhos. Como os *peers* filhos de **Peer B** (**Peer D** e **Peer E**) já concluíram a execução de seus respectivos *DAG slices*, estes enviam os resultados obtidos em tais execuções para **Peer A**. Se, por acaso, **Peer D** e/ou **Peer E** ainda não tiverem finalizado suas computações, eles primeiramente indicam ao *peer* pai de **Peer B**, **Peer A** no caso, que eles ainda mantêm execuções em andamento. Sendo assim, logo que **Peer D** e **Peer E** finalizem suas execuções, estes enviam seus resultados para **Peer A**. Mas, visto que a árvore de execução se desconectou devido a falta de **Peer B**, como é possível **Peer D** e **Peer E** se comunicarem, localizarem e até mesmo saberem que **Peer A** é o *peer* pai de **Peer B**? A resposta para esta pergunta reside na lista ordenada de *peers* armazenada nos *DAG slices*, formada ao longo do processo de escalonamento. Tanto no **DAG D** quanto no **DAG E** existe informação sobre identidade e localização de todos os *peers* pais acima na árvore de execução. Por exemplo, as informações contidas nesta lista do **DAG D** poderia incluir: **Peer A** → **Peer B** → **Peer D** (Figura 4.5 - (h));

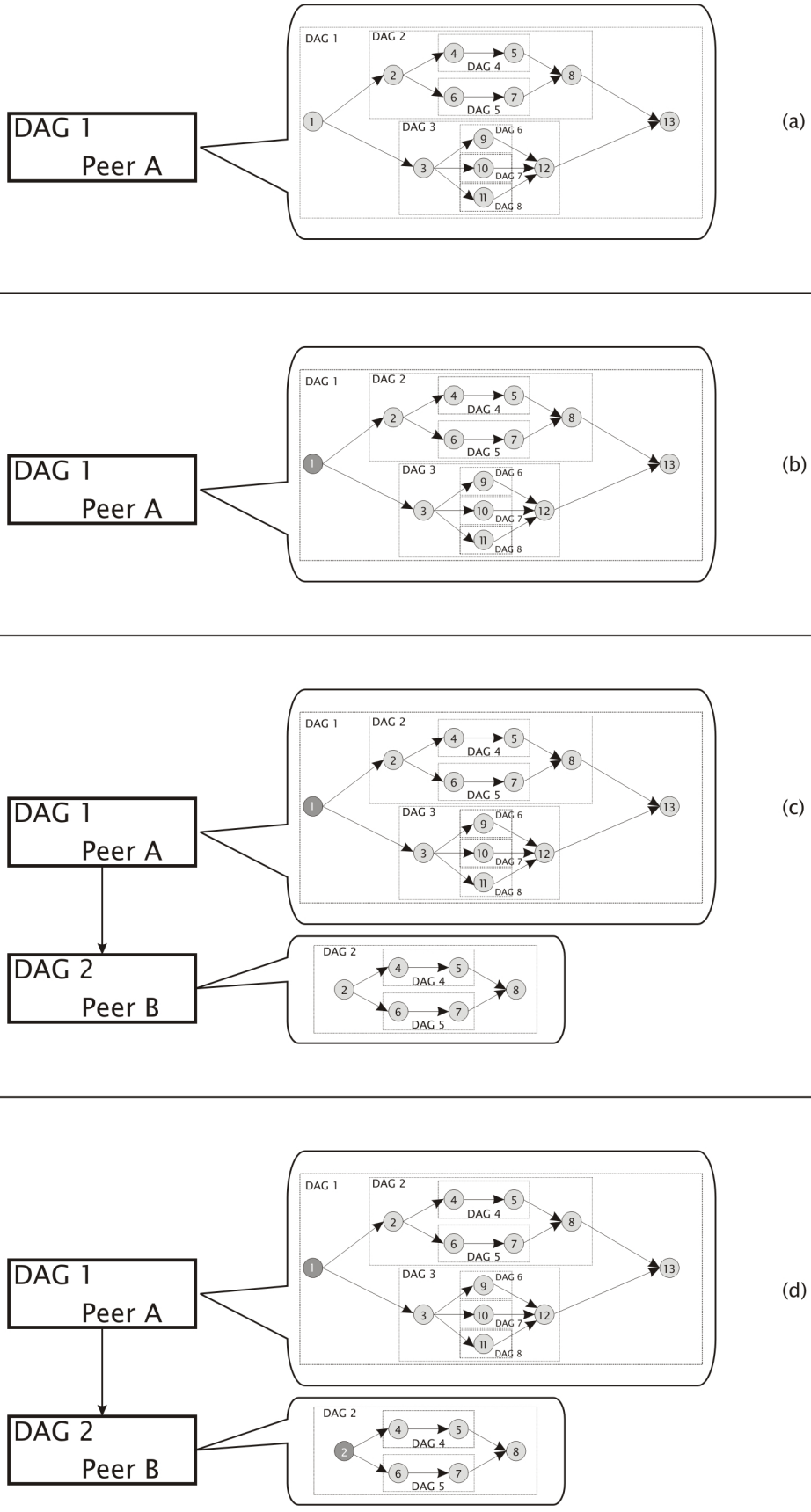
9. **Peer A** recebe resultados de **DAG 4** e **DAG 5** (Algoritmo 2 - linha 6). Como mostrado no Algoritmo 1, **Peer A** marca **DAG 4** e **DAG 5** como executados com sucesso. **Peer A** também marca tarefas e/ou *DAG slices* anteriores a **DAG 4** e **DAG 5** como executados com sucesso (Algoritmo 2 - linhas 7 e 8) (Figura 4.5 - (i));

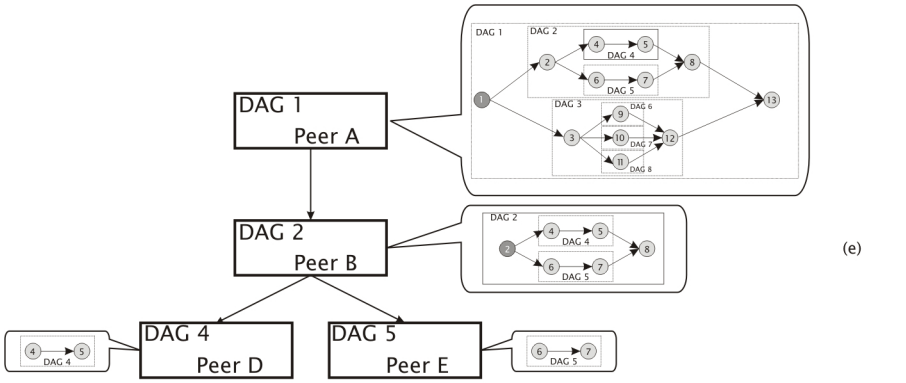
10. **Peer A** reenvia **DAG 2** para um outro *peer* escolhido no processo de escalonamento, **Peer I** no caso (Figura 4.5 - (j));

11. **Peer I** executa **tarefa 8** e finaliza a computação de **DAG 2** (Figura 4.5 - (k));

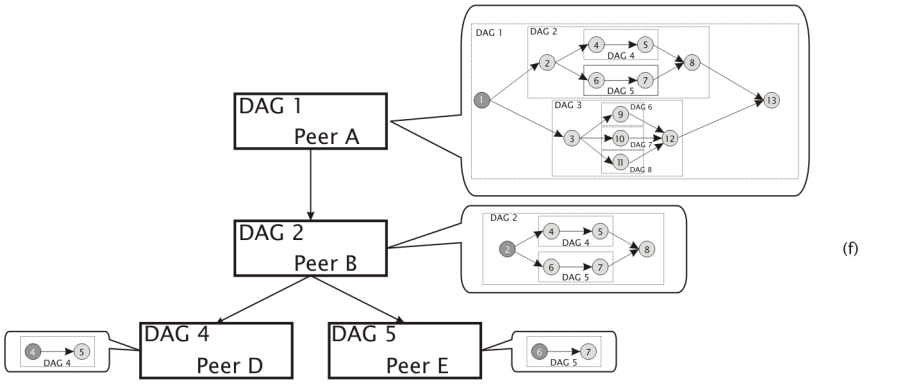
12. **Peer I** envia resultados de **DAG 2** para **Peer A**. Logo, **DAG 2** foi executado com sucesso. Se **DAG 3** também tiver sido finalizado, **Peer A** pode então executar **tarefa 13** e concluir a execução do DAG (Figura 4.5 - (l)).

Neste exemplo pode-se notar que os mecanismos utilizados na recuperação de execução são atrelados ao processo de escalonamento. Assim como o escalonamento, onde todos os *peers* são responsáveis em distribuir as tarefas componentes do DAG, os mecanismos de tolerância a faltas também são realizados de forma completamente descentralizada e distribuída, visto que todos os participantes possuem as mesmas funcionalidades e colaboram entre si, alternando papéis de clientes e servidores a fim de assegurar a correta execução do DAG. De acordo com esta descentralização e distribuição de responsabilidades, o *middleware* é capaz de tratar de forma coesa a ocorrência de mais de uma falta durante a execução do DAG, desde que o *DAG owner* não falhe. Este tratamento distribuído de faltas confere ao *middleware* robustez e flexibilidade diante da intermitência dos ambientes P2P.

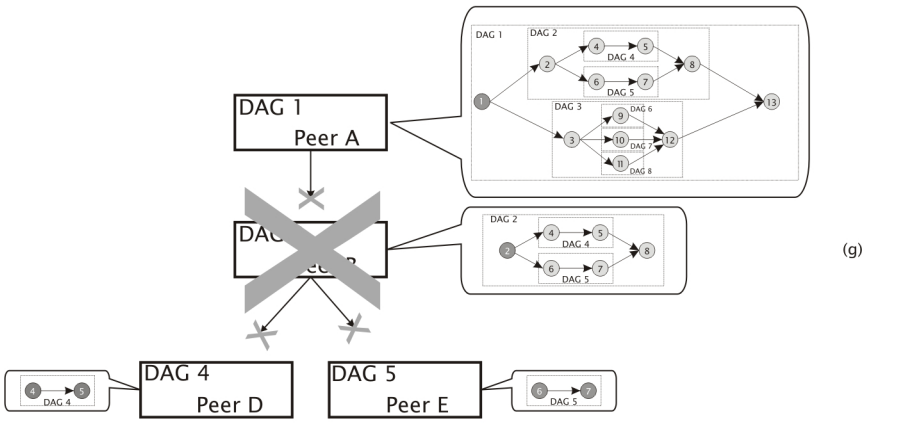

DAG 1
Peer A
DAG 2
DAG 4
DAG 5
DAG 3
DAG 6
DAG 7
DAG 8
(a)
(b)
(c)
DAG 2
Peer B
(d)


DAG 1
Peer A
DAG 2
Peer B
DAG 4
Peer D
DAG 5
Peer E

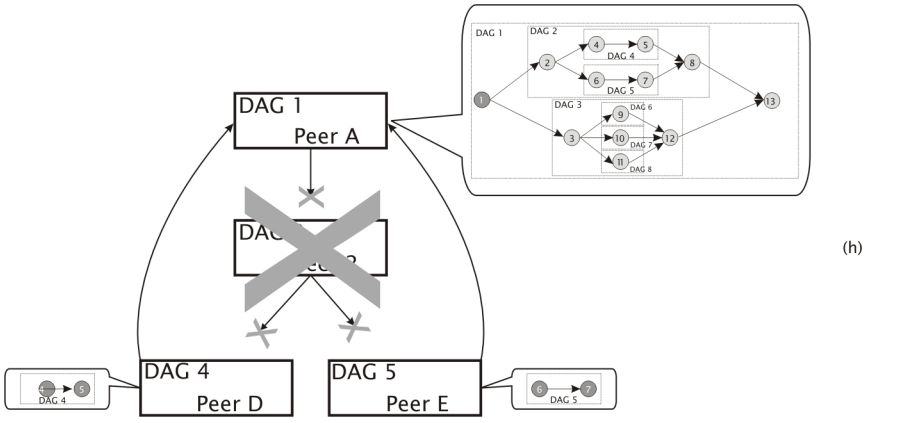
(e)


DAG 1
Peer A
DAG 2
Peer B
DAG 4
Peer D
DAG 5
Peer E

(f)


DAG 1
Peer A
DAG 4
Peer D
DAG 5
Peer E

(g)


DAG 1
Peer A
DAG 4
Peer D
DAG 5
Peer E

(h)

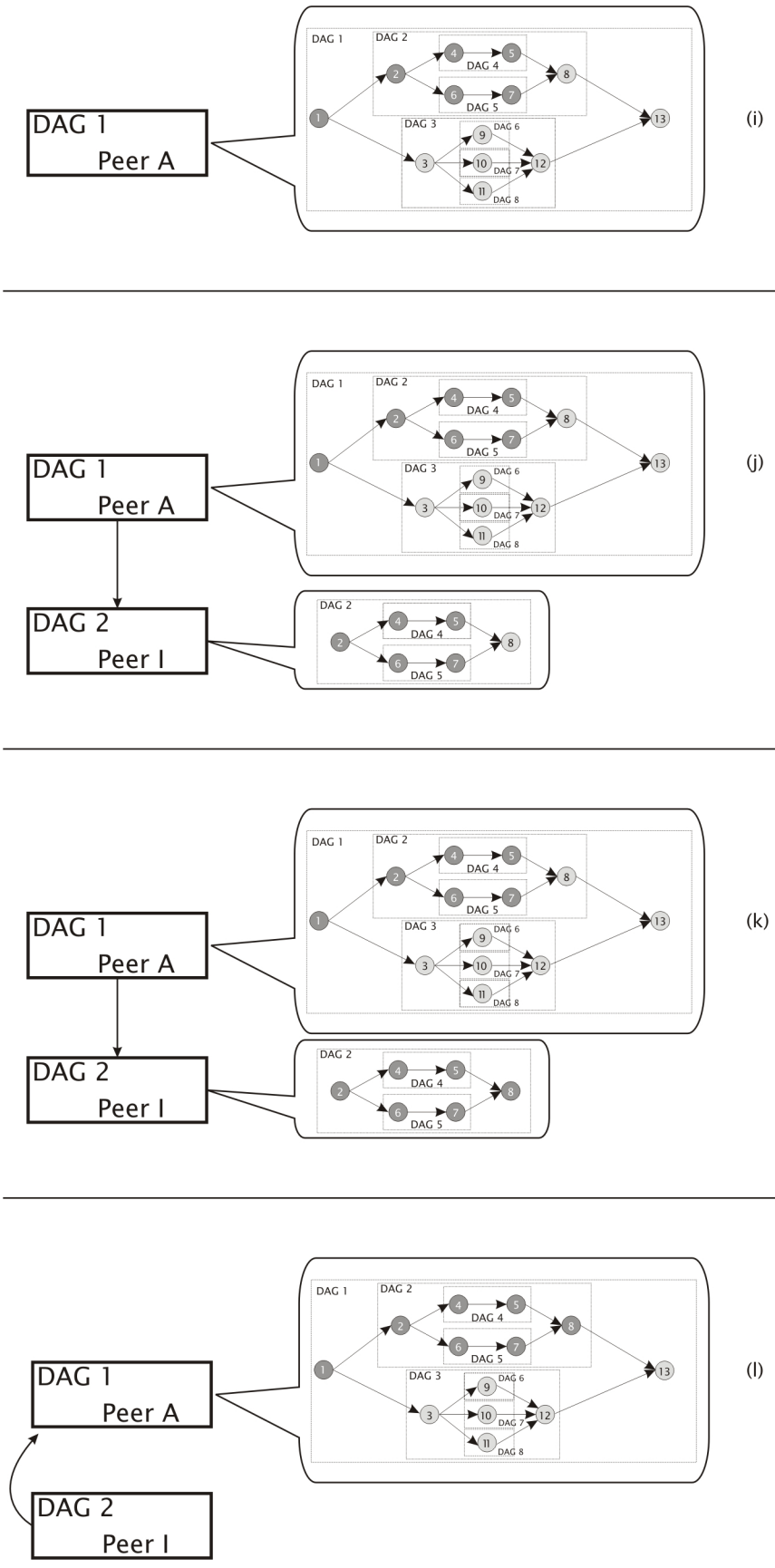


Figura 4.5: Passos envolvidos na recuperação de execução

Capítulo 5

Implementação e Resultados Experimentais

Aplicações distribuídas são, por natureza, aplicações executadas em ambientes computacionais heterogêneos. Devido à grande variedade de sistemas operacionais, plataformas, redes e linguagens de programação, as aplicações distribuídas têm utilizado, cada vez mais, tecnologias que abstraiam esta heterogeneidade de ambientes. No intuito de facilitar o desenvolvimento e obter uma maior disseminação, aspectos que vêem ganhando grande importância na implementação de aplicações distribuídas são a interoperabilidade e a independência de plataforma. Com o atendimento destes dois aspectos, desenvolvedores e engenheiros de software podem focar seus esforços na estruturação e implementação das regras de negócios, agilizando e barateando o custo do desenvolvimento, visto que questões infra-estruturais (redes, sistemas operacionais, hardware, etc) se tornam transparentes com a utilização de tecnologias interoperáveis e independentes de plataforma.

Para alcançar os requisitos de interoperabilidade desejáveis em aplicações distribuídas, o *middleware* apresentado neste trabalho foi implementado através da linguagem de programação Java. Outros produtos e tecnologias da família Java também foram utilizados na implementação do *middleware*, os quais compõem as funcionalidades P2P e infra-estrutura básica de comunicação, no caso, JXTA e RMI, detalhados nas Seções 5.1 e 5.2 respectivamente.

5.1 JXTA

As funcionalidades P2P suportadas pelo *middleware*, como a descoberta de *peers* e gerenciamento da rede *overlay*, são implementadas através da especificação JXTA (leia-se juxta), baseada em Java. JXTA pode ser definido como um conjunto de protocolos abertos que permite que qualquer dispositivo conectado a rede, desde telefones celulares, PDAs e

PCs até servidores comuniquem e colaborem entre si de forma P2P [23]. Em [27], JXTA é vista como uma tecnologia que provê uma plataforma de computação e programação baseada no paradigma P2P e que implementa em qualquer dispositivo da rede funcionalidades para que estes possam interagir entre si de forma P2P.

Com a utilização de JXTA também se verifica a criação de uma rede virtual na qual qualquer *peer* pode interagir com outros *peers* e recursos diretamente, ou seja, a criação de rede *overlay*, característica esta necessária na classificação de uma aplicação como P2P. Através da *overlay* JXTA *peers* podem se comunicar até mesmo com outros *peers* localizados atrás de *firewalls* e NATs (*Network Address Translation*), além de poderem também interagir através de diferentes protocolos de transporte.

JXTA é considerada uma nova alternativa às abordagens já existentes na implementação de aplicações distribuídas, como CORBA ou Java RMI. Visto que a especificação JXTA define um conjunto de protocolos com o objetivo de contemplar as funcionalidades básicas de um sistema P2P, JXTA provê uma maneira útil e simples de exemplificar, modelar e entender o paradigma P2P. JXTA encapsula os mecanismos básicos necessários à criação de uma aplicação P2P completa, sendo os principais:

- Descoberta de *peers*, recursos e serviços;
- Publicação de serviços disponíveis nos *peers*;
- Troca de mensagens entre *peers*;
- Roteamento de mensagens entre *peers*;
- Estabelecimento de critérios de seleção de *peers*;
- Agrupamento de *peers*.

Cada um destes mecanismos são encapsulados em protocolos JXTA específicos. Entretanto, de acordo com a conveniência e tipo de aplicação, é possível focar e utilizar apenas um subconjunto destes mecanismos, por exemplo, a descoberta e publicação de recursos dos *peers*. Respeitando esta separação de conceitos e funcionalidades é possível construir aplicações P2P sob medida, onde utiliza-se apenas as características necessárias, não havendo assim a necessidade de se implementar todos os serviços, agilizando o desenvolvimento. Outra importante característica da especificação JXTA diz respeito à independência de plataforma com relação a:

- Linguagem de programação: a implementação dos protocolos JXTA podem ser encontrados em Java, C, C++ e outras linguagens;

- Protocolo de transporte: não é necessário utilizar uma estrutura de rede específica. Aplicações P2P através de JXTA podem ser desenvolvidas sobre TCP/IP, HTTP, *Bluetooth* e outros protocolos de transporte;
- Segurança: Os desenvolvedores são livres para gerenciar as questões de segurança na plataforma utilizada.

5.1.1 Protocolos JXTA

Nesta seção são apresentados os principais protocolos componentes da especificação JXTA, além de seus relacionamentos na pilha de protocolos. Como ressaltado na Seção 5.1, os protocolos JXTA são independentes uns dos outros. A definição de um *peer* JXTA não requer a implementação de todos os protocolos para participar da rede. A especificação JXTA define um conjunto de seis protocolos básicos, sendo eles:

- *Peer Discovery Protocol* (PDP);
- *Peer Information Protocol* (PIP);
- *Peer Resolver Protocol* (PRP);
- *Pipe Binding Protocol* (PBP);
- *Endpoint Routing Protocol* (ERP);
- *Rendezvous Protocol* (RVP).

A Figura 5.1 mostra a disposição destes protocolos na especificação JXTA.

Todos os protocolos da Figura 5.1 são baseados na troca de mensagens XML entre os *peers* participantes da rede. O ***Peer Discovery Protocol*** é utilizado pelos *peers* na publicação de seus próprios recursos (grupos, serviços, etc) e para descobrir recursos de outros *peers*. Cada recurso de um *peer* é descrito e publicado através de documentos XML denominados ***advertisements***, que são propagados via *broadcast* aos demais *peers* da rede. O ***Peer Information Protocol*** pode ser visto como uma ferramenta estatística e de monitoramento dos *peers* da rede, obtendo informações como estado, níveis de tráfego e utilização dos *peers*. O ***Peer Resolver Protocol***, um protocolo base da especificação JXTA, permite que *peers* enviem requisições genéricas a um ou mais *peers* e recebam resposta (ou múltiplas respostas) da requisição enviada. O ***Pipe Binding Protocol*** é utilizado pelos *peers* no estabelecimento de um canal virtual de comunicação, os chamados ***pipes***, entre *peers*. *Pipes* constituem o principal mecanismo de comunicação entre *peers* na especificação JXTA. Os nós, ou *peers*, conectados através de um *pipe* são

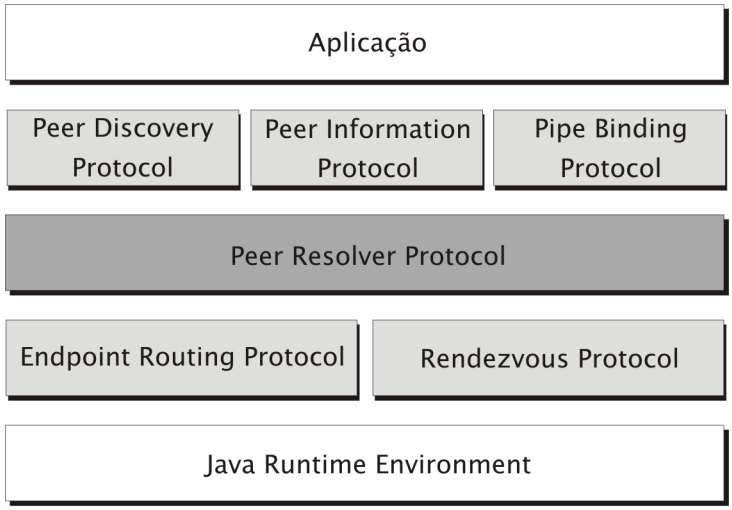


Figura 5.1: Pilha de protocolos JXTA

denominados ***endpoints***. O ***Endpoint Routing Protocol*** é o responsável em localizar rotas até o *endpoint* destino de uma dada requisição ou mensagem. As informações sobre rotas incluem uma seqüência ordenada de *peers* intermediários que podem ser usados no encaminhamento da mensagem até o *endpoint* destino. Por último, o ***Redenzvous Protocol*** oferece mecanismos pelos quais os *peers* podem se inscrever em serviços de propagação de mensagens, ou seja, mecanismos que possibilitam aos *peers* receberem as mensagens de *broadcast* de outros *peers*. O *Redenzvous Protocol* é utilizado tanto pelo *Peer Resolver Protocol* quanto pelo *Pipe Binding Protocol* na propagação de mensagens.

5.1.2 JXTA em Execução

Na Seção 5.1.1 foram apresentados os protocolos da especificação JXTA e suas principais responsabilidades. Nesta seção é explicado, de forma simples e objetiva, como uma aplicação P2P implementada sobre JXTA se comporta em tempo de execução.

Para compreender a execução de uma aplicação JXTA é necessário primeiramente classificar os *peers* componentes da aplicação com relação às suas funcionalidades. Os tipos de *peers* mais importantes na especificação JXTA são os do tipo *edge* e *rendezvous* [7]. Os *peers* do tipo *edge* são aqueles que podem enviar e receber mensagens e tipicamente armazenam os *advertisements* de outros *peers* e recursos da rede. Esta categoria de *peer* pode responder a requisições através das informações localizadas em seus *advertisements*, mas não reencaminham tais solicitações a outros *peers*. A grande maioria dos *peers* envolvidos em uma aplicação JXTA são do tipo *edge*. Os *peers* do tipo *rendezvous* são bastante semelhantes aos *peers edge*. Entretanto, o *rendezvous* realiza o reencaminhamento de requisições para ajudar na localização de *peers* e recursos ao longo da rede.

Visto que na especificação JXTA os *peers* são dispostos em grupos, quando um *peer* se junta a um dado grupo, ele automaticamente procura o *rendezvous* do grupo. Se nenhum *rendezvous* é localizado, o *peer* que se juntou ao grupo se torna o *rendezvous* para aquele grupo. Cada *rendezvous* mantém uma lista de outros *rendezvous* e também dos *peers* que o estão utilizando com o *rendezvous*. Geralmente, o *rendezvous* é utilizado nas bordas das sub-redes onde a aplicação é executada. É ele o responsável em integrar diferentes sub-redes ao longo da Internet. Como todas as requisições entre sub-redes passam pelo *rendezvous*, este pode ser visto como a porta de entrada e de saída de informações de uma sub-rede.

Após a definição dos tipos de *peers* na especificação JXTA, na Figura 5.2 é apresentada a seqüência de procedimentos realizados em um cenário onde um dado *peer* A tenta localizar um recurso oferecido por um *peer* B, residente em outra sub-rede.

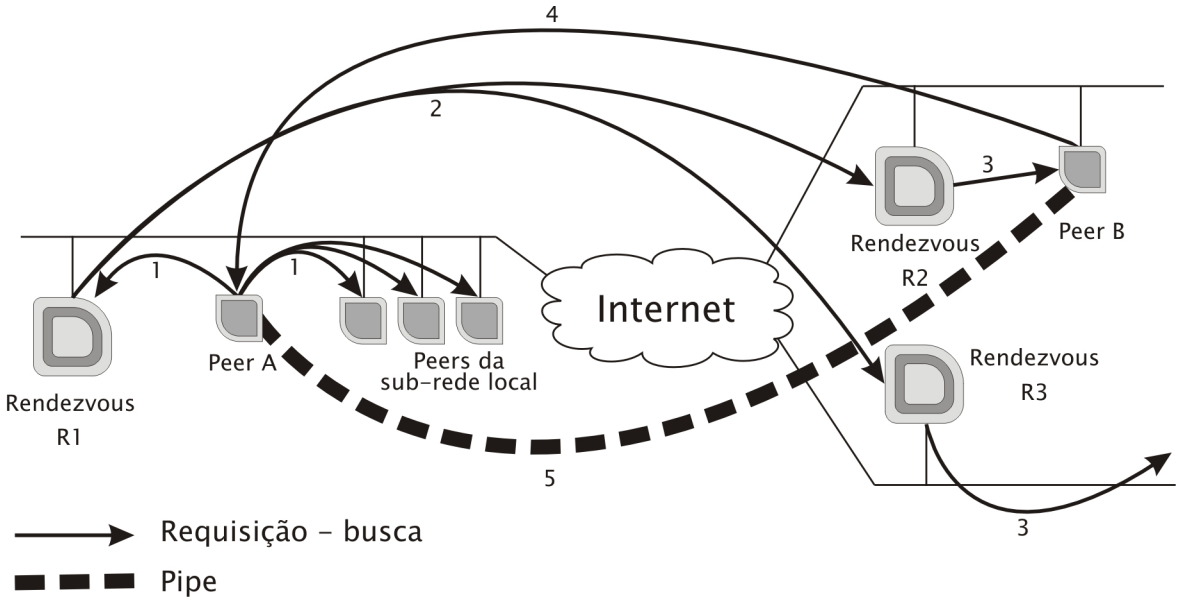


Figura 5.2: Cenário básico de uma aplicação JXTA

O *peer* A é um *peer* do tipo *edge* e está configurado para utilizar o *peer* R1 como seu *rendezvous*. Quando o *peer* A inicializa o processo de busca, seu pedido é enviado inicialmente a seu *rendezvous* - R1 no caso, e para todos os outros *peers* na mesma sub-rede via *broadcast* **(1)**. *Peers* que receberam a requisição do *peer* A responderão diretamente a ele se contiverem o recurso ou informação sobre o recurso desejado pelo *peer* A. Para o envio de requisições além da sub-rede local ao *peer* A é necessária a utilização do *rendezvous* R1. O *peer* R1 tenta primeiramente satisfazer a requisição através de seus recursos locais. Se ele contiver a informação ou o recurso desejado, ele responderá ao *peer* A e não mais encaminhará a requisição a outros *peers*. Se R1 não contém a informação

requisitada, outros *rendezvous* de outras sub-redes serão contactados **(2)**. Uma vez que o *peer* B e o recurso correspondente são localizados **(3)** o *peer* B responde diretamente ao *peer* A **(4)**. A partir de então, para poderem se interagir e comunicar, os *peers* A e B estabelecem um canal de comunicação entre si, ou seja, um *pipe* **(5)**. Ao final da interação entre os *peers* A e B, o *pipe* que os conectava é finalizado e ambos retornam a seus estados iniciais.

5.1.3 JXTA no *Middleware*

Visto na Seção 5.1.2 o comportamento básico de uma aplicação JXTA, nesta seção é discutido a forma com que a especificação e protocolos JXTA são utilizados no *middleware*, com o objetivo de prover uma plataforma flexível e descentralizada de computação de *workflows*.

Apesar da especificação JXTA oferecer uma série de funcionalidades básicas para aplicações P2P e, partindo da independência e auto-suficiência destas funcionalidades, no *middleware* o único mecanismo JXTA utilizado é o serviço de descoberta de *peers*, implementado sobre o protocolo PDP. Através deste serviço de descoberta o *middleware* é capaz de localizar e adicionar novos *peers* à rede *overlay*, onde serão utilizados em computações posteriores. Como explicado na Seção 5.1.1, toda a adição de novos *peers* ou recursos na rede é feita através da publicação de *advertisements*. O serviço de descoberta empregado no *middleware* utiliza estes *advertisements* para localizar e adicionar *peers* à *overlay*. Quando um determinado *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*. Esta última informação, o processamento a ser compartilhado na rede, é utilizada no processo de escalonamento.

Visto que no *middleware* o processo de escalonamento é realizado de forma distribuída e como todos os *peers* possuem *advertisements* de todos os outros *peers* (*broadcast*), a informação sobre a quantidade de processamento compartilhado em cada *peer* norteia a escolha de quais *peers* são capazes de computar um dado *DAG slice* ou tarefa. Logo, como mostrado no Algoritmo 1 da Seção 4.3, a decisão sobre qual *peer* é escolhido para uma determinada computação é tomada levando em consideração a quantidade de processamento disponibilizada por cada *peer*, contida nos *advertisements*, e o custo computacional de cada *DAG slice* e/ou tarefa a ser computado.

Quando um *peer* sai da rede, de forma explícita (sem a ocorrência de faltas), seus *advertisements* são removidos da *overlay* e o *peer* e seus respectivos recursos não estarão mais disponíveis aos outros *peers* da rede.

Todos os procedimentos descritos anteriormente são encapsulados no módulo de **Gerência da Rede Overlay** (Figura 4.1). Tais procedimentos também são responsáveis

pela configuração e introdução de um novo *peer* à *overlay*. Ações como localização do *peer rendezvous* e inicialização dos mecanismos de descoberta são realizados de forma transparente e automática pelo protocolo PDP. Cabe ao novo *peer*, no momento da entrada na *overlay*, apenas a determinação da quantidade de processamento que deseja compartilhar.

Visto que a especificação e protocolos JXTA são utilizados no *middleware* com os propósitos de criar e gerenciar a rede *overlay* e descobrir novos *peers*, uma outra questão de fundamental importância a ser detalhada é a forma de comunicação entre os *peers* participantes da rede. Tal funcionalidade é implementada através de Java RMI e será analisada a seguir.

5.2 Java RMI

RMI é uma interface de programação que permite a execução de chamadas remotas no estilo RPC em aplicações desenvolvidas em Java [15]. Como uma das principais tecnologias utilizadas no desenvolvimento de sistemas distribuídos utilizando-se Java como linguagem de programação, RMI provê um conjunto de ferramentas para que seja possível ao programador desenvolver uma aplicação sem se preocupar com os detalhes de comunicação entre os diversos elementos componentes do sistema. Dentre estes detalhes pode-se citar o estabelecimento, gerenciamento e encerramento de conexões, e a serialização e desserialização das informações transmitidas. A grande vantagem da utilização de RMI em relação ao RPC reside no fato de ser possível, através de RMI, a invocação de métodos de objetos remotos e a transferência eficiente e confiável de objetos complexos entre os nós participantes do sistema.

O funcionamento básico de uma aplicação implementada com RMI é bastante simples. O objeto que deseja disponibilizar um sub-conjunto de seus métodos para acesso remoto, conhecido como objeto remoto ou simplesmente servidor RMI, expõe tais métodos e suas respectivas assinaturas em uma interface remota. Em outras palavras, o objeto remoto implementa a interface remota. O objeto ou nó cliente que deseja utilizar os métodos do objeto remoto necessita primeiramente localizá-lo, o que geralmente é feito através de mecanismos de registros (*RMI Registry*, por exemplo) ou através de servidores HTTP. De posse da localização e da interface remota, o cliente RMI simplesmente invoca o método desejado, que para o cliente se assemelha a uma chamada de método local, entretanto, a execução de tal método é realizada remotamente na máquina do servidor RMI.

No *middleware*, RMI é responsável em implementar as funcionalidades envolvidas no módulo *Comunicação P2P* (Figura 4.1). Os procedimentos encapsulados neste módulo são os que realmente realizam a comunicação e transferência de dados entre os *peers* participantes do *middleware*, como serialização e desserialização de dados e gerência de conexões.

Visto que a especificação JXTA provê seus próprios mecanismos de comunicação entre *peers*, os *pipes*, uma série de questões é apresentada para justificar a utilização de RMI ao invés dos *pipes*:

- O estabelecimento de um *pipe* entre um par de *peers* é um processo caro, complexo e com tempo de duração indeterminado. Por exemplo, se um dado *peer* A deseja criar um pipe para se comunicar com um *peer* B, A deve primeiramente solicitar que B publique um *advertisement* contendo informações sobre o *pipe* a ser criado. Estas informações incluem o identificador do *pipe* (todos os recursos em uma aplicação JXTA necessitam de uma identificação única), tipo do *pipe* (existem vários tipos diferentes), identificador do *peer* B, dentre outras. Após a publicação do *advertisement* pelo *peer* B, o *peer* A pode ter que aguardar uma quantidade de tempo imprevisível para receber o *advertisement*, visto que a troca de informações na *overlay* são assíncronas e com qualidade de serviço do tipo *best effort*. De posse do *advertisement* de B, o *peer* A pode então inicializar o complexo processo de estabelecimento do *pipe* com o *peer* B;

- RMI, além de apresentar boa performance, abstrai do programador os procedimentos complexos envolvidos na comunicação entre o nó cliente e servidor;

- Os objetos que representam DAGs e tarefas são bastantes complexos e somente o *DAG owner* possui a implementação (classes) destes objetos. Para possibilitar a transferência destes objetos pela rede são necessários mecanismos de serialização e desserialização. Tais mecanismos oferecidos pelos *pipes* JXTA são ineficientes e exigem muito processamento e tempo para operarem sobre objetos complexos. Além do mais, visto que somente o *DAG owner* possui a definição da classe dos objetos transferidos, através de *pipes*, quando o *peer* destino recebe a seqüência de bytes resultado da serialização do objeto, tal *peer* não consegue realizar a desserialização e a restauração do estado do objeto, pois o *peer* destino não possui a classe do objeto;

- RMI possui uma funcionalidade que possibilita a transferência de objetos mesmo que o nó destino não tenha conhecimento de sua implementação (classe). Através do chamado **download automático de código**, assim que necessário, RMI realiza de maneira transparente e automática a transferência não só do objeto mas também dos *bytecodes* da definição do objeto, ou seja, a classe responsável pela implementação do objeto. Em adição, RMI também é capaz de realizar de forma bastante eficiente a serialização, desserialização e transferência dos mais complexos objetos através da rede.

RMI também é utilizado no *middleware* na implementação dos mecanismos de tolerância a faltas. Sua natureza semelhante ao estilo RPC, onde o cliente fica bloqueado (e

conectado) ao servidor até o fim da chamada do método, auxiliou na detecção de faltas na direção *top-down* na árvore de execução - um *peer* cliente realiza uma chamada de método a um *peer* servidor remoto para que este compute um determinado *DAG slice*. Se o *peer* servidor falha antes de retornar o método, ou seja, antes de finalizar a execução do *DAG slice*, uma exceção é lançada no *peer* cliente, apontando que o *peer* servidor se desligou da rede.

Embora tenha sido importante o comportamento RPC apresentado por RMI, a principal contribuição de tal tecnologia para os procedimentos de tolerância a faltas do *middleware* reside no mecanismo nativo de *leasing* do RMI. RMI possui toda uma especificação de *leasing* para aplicações distribuídas Java. Ainda comparando com o estilo RPC, em uma aplicação RMI, se o cliente se desconecta do servidor antes do término da execução do método remoto, nenhuma exceção ou notificação é enviada ao servidor, o qual simplesmente abortará a execução do método requisitado por aquele cliente. O mecanismo de *leasing* de RMI permite esta notificação ao servidor - quando um cliente se conecta ao servidor RMI, este último atribui ao cliente uma *lease* e um período de tempo no qual aquela *lease* é válida. Cabe ao cliente, até o fim da execução do método remoto do servidor, renovar sua *lease* de tempos em tempos. Se por algum motivo o cliente não renovar sua *lease*, em uma ocorrência de falta por exemplo, uma notificação é enviada ao servidor que pode então tomar determinada ação. No *middleware*, o mecanismo de *leasing* do RMI é utilizado na detecção de faltas na direção *bottom-up* na árvore de execução.

A utilização de RMI também facilitou o desenvolvimento do *middleware* como uma aplicação *multi-thread*, onde um dado *peer* pode estabelecer conexões RMI simultâneas com diferentes *peers* a fim de distribuir vários *DAG slices* para a computação. Desta forma, cada *thread* se encarrega do estabelecimento da conexão com o *peer* correspondente e gerencia o retorno dos resultados dos respectivos *DAG slices*, de forma independente e concorrente, o que garante ao *middleware* um maior paralelismo na execução do DAG. Esta concorrência pode ser melhor visualizada na Seção 5.4, onde através de um diagrama de seqüência de eventos, as diferentes *threads* gerenciam as execuções remotas de seus *DAG slices*.

5.3 RMI + JXTA

Ao longo das Seções 5.1.3 e 5.2 foram abordadas as formas com que tanto a especificação JXTA quanto RMI foram empregadas na implementação do *middleware*. Entretanto, em tais seções nenhuma interseção entre JXTA e RMI foi apresentada e sim suas utilizações individuais. Nesta seção é explicado como JXTA e RMI funcionam juntos no *middleware*.

A especificação JXTA é utilizada apenas para a criação e gerenciamento da rede *overlay*, na qual o *middleware*, através do serviço de busca JXTA, adiciona novos *peers* para

computações posteriores. De posse dos *advertisements* publicados pelos *peers* participantes, quando um dado *peer* deseja enviar um *DAG slice* para computação em outro *peer*, o primeiro necessita estabelecer uma conexão RMI com o segundo e invocar seu método para realizar tal computação. O *peer* que submeteu o *DAG slice* para execução remota faz então a chamada de método passando os parâmetros necessários e aguarda o retorno da execução com os resultados obtidos. Mas, como é possível aos *peers*, de posse apenas dos *advertisements*, estabelecer uma conexão RMI com o *peer* destino? Para responder a esta pergunta é necessário primeiramente abordar duas questões: as informações contidas nos *advertisements* dos *peers* e os requisitos para o estabelecimento de uma conexão RMI entre um par de nós.

Os *advertisements* publicados pelos *peers* quando estes se juntam à rede possuem informações intrínsecas à especificação JXTA. A mais importante destas informações é o identificador JXTA único atribuído a cada *peer* (e a cada recurso da rede). Este identificador de 128 bits é formado através de uma função *hash*, que toma como entrada os atributos de rede do *peer* local como nome, endereço, IP, endereço de sub-rede, endereço MAC, dentre outros. Logo, na *overlay* JXTA os *peers* participantes são conhecidos através de seus identificadores.

Com relação ao estabelecimento da conexão RMI entre dois nós, é necessário que o nó servidor seja publicamente acessível por qualquer outro nó. Isto quer dizer que o servidor RMI deve ou possuir um endereço IP fixo e público ou possuir um nome pelo qual possa ser localizado através de DNS (*Domain Name System*). De acordo com estes requisitos, não é possível estabelecer uma conexão RMI entre máquinas privadas (endereço IP privado ou envolvidas em uma NAT) distribuídas em duas sub-redes distintas. Sendo assim, como é possível então que dois *peers*, possivelmente residentes em sub-redes diferentes, estabeleçam uma conexão RMI para a computação de um *DAG slice*? A resposta para esta pergunta reside na utilização de um sub-projeto da especificação JXTA no *middleware*, o **PeerMI** [4]. O PeerMI pode ser visto como um componente intermediário entre JXTA e RMI, ou seja, uma interface entre as duas tecnologias. De posse do identificador do *peer* destino, o PeerMI possibilita a conexão RMI entre os *peers* através da *overlay* JXTA em que estão inseridos. Em outras palavras, o PeerMI é o responsável em realizar uma espécie de tunelamento na rede *overlay* para possibilitar a conexão entre *peers*. A Figura 5.3 mostra um exemplo do estabelecimento de uma conexão RMI através da utilização do PeerMI no *middleware*.

De acordo com os identificadores dos *peers* origem e destino, o PeerMI se responsabiliza, juntamente com os protocolos JXTA, em traçar uma rota na rede *overlay* para possibilitar a conexão RMI entre tais *peers*. Sendo assim, através do PerMI, *peers* localizados em sub-redes protegidas com *firewalls* ou com esquema de endereçamento do tipo NAT podem estabelecer e receber conexões RMI provindas de outras sub-redes.

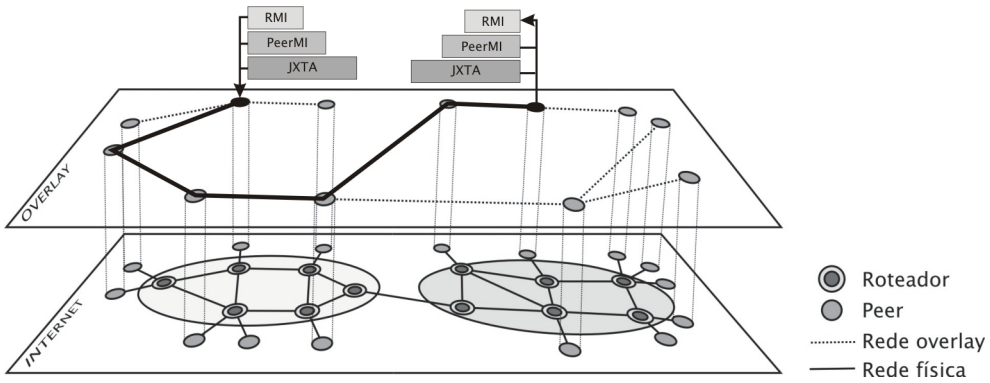


Figura 5.3: PeerMI no estabelecimento de conexão RMI

A solução proposta pelo PeerMI é semelhante àquela oferecida pelos serviços de VoIP e *instant messenger*, que, através da *overlay* permitem que *peers* com diferentes configurações de rede possam estabelecer canais de comunicação entre si.

5.4 Modelagem e Seqüência de Eventos

Durante a fase de modelagem do *middleware* apresentado neste trabalho objetivou-se a construção de uma arquitetura robusta e clara com relação aos papéis e funcionalidades desempenhados pelos diferentes módulos componentes da aplicação. As funcionalidades identificadas na fase de análise de requisitos (Figura 4.1) e as exigências das tecnologias utilizadas também auxiliaram no desenvolvimento da arquitetura do *middleware*.

A Figura 5.4 mostra o diagrama de classes UML das estruturas identificadas durante a fase de modelagem do *middleware*. A seguir são explicados em detalhes os papéis e responsabilidades de cada uma destas estruturas.

- **Classe** `Core`: a classe `Core` é a responsável em adquirir e estabelecer os parâmetros de configuração iniciais tanto do ambiente JXTA quanto do RMI. O usuário que deseja se juntar à rede comunica com a instância da classe `Core` para informar a quantidade de processamento que deseja compartilhar no *middleware*. A principal e fundamental responsabilidade da classe `Core` é a inserção do *peer* na rede *overlay*, e conseqüentemente no conjunto de *peers* participantes do *middleware*. A captura dos atributos da máquina local (endereço IP, plataforma, etc), a atribuição de identificador ao novo *peer*, a publicação de *advertisements*, a procura do *peer rendezvous* da sub-rede e o estabelecimento dos parâmetros de segurança do RMI também são atividades imprescindíveis realizadas pela classe `Core`;

- **Classe** `Server`: visto que no *middleware* (e na maioria das aplicações P2P) os *pe-*

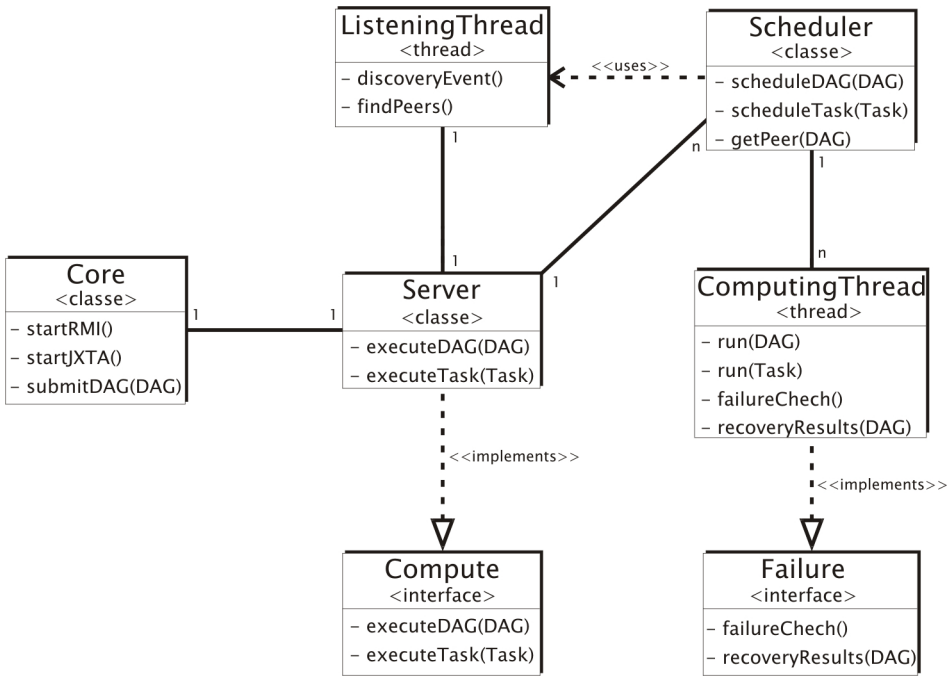


Figura 5.4: Diagrama de classe UML do *middleware*

ers participantes desempenham papéis tanto de cliente quanto de servidor, a classe `Server` encapsula as funcionalidades envolvidas no lado servidor da aplicação. Em outras palavras, a classe `Server` representa o lado servidor RMI do *middleware*. Em uma aplicação RMI, os métodos do servidor disponibilizados para acesso remoto são declarados em uma interface remota e implementados pela classe servidor. No *middleware*, a classe `Server` 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". Após a computação do *DAG slice* e/ou tarefa, e conseqüentemente após a chamada do método remoto, o *peer* cliente recebe os resultados da execução do *peer* servidor. A classe `Server` também realiza outra tarefa importante: a inicialização dos mecanismos de *leasing* do *middleware*. Assim que a classe `Server` recebe a chamada de método remoto de algum cliente, a classe `Server` inicializa para aquele cliente um processo de *leasing*. Se alguma falta ocorrer com este cliente (direção *bottom-up* na árvore de execução) a classe `Server` se encarrega em determinar e obter informações sobre o ponto onde a execução foi interrompida para que os mecanismos de recuperação possam retomar a execução do *DAG slice*. Além do mais, a classe `Server` se responsabiliza também pela inicialização do processo de descoberta de *peers* da especificação JXTA;

- ***Thread*** `ListeningThread`: a *thread* `ListeningThread` é responsável em "ouvir"a rede *overlay* JXTA à procura de novos *peers*. Implementada como uma *thread* que é executada indefinidamente, `ListeningThread` executa os procedimentos envolvidos no serviço de descoberta da especificação JXTA, implementados sobre o protocolo PDP. Através da captura de *advertisements* publicados pelos *peers*, a *thread* `ListeningThread` adiciona tais *peers* da *overlay* JXTA ao ambiente do *middleware*, tornando-os visíveis aos demais participantes. `ListeningThread` também é responsável em entregar ao mecanismo de escalonamento uma lista de *peers* disponíveis no momento, além de "tentar"manter atualizada esta lista;

- **Classe** `Scheduler`: a classe `Scheduler` implementa os procedimentos e algoritmos envolvidos nos processos de escalonamento e coordenação de execução de tarefas. Assegurar a correta execução do DAG, respeitando as dependências entre tarefas, e a escolha dos melhores *peers* para execução dos *DAG slices* são de responsabilidade da classe `Scheduler`. Além de servir como base para alguns procedimentos do mecanismo de recuperação de execução (marcação de tarefas e *DAG slices* executados com sucesso), a classe `Scheduler`, após a escolha de um *peer* para a computação de um dado *DAG slice*, realiza outras duas importantes atividades: inicializar a *thread* responsável pelo gerenciamento da execução do *DAG slice* e adicionar informações do *peer* local na lista ordenada de *peers* do *DAG slice* a ser computado. Este último procedimento é importante para o caso onde um *peer* intermediário na árvore de execução falha, desconectando-a;

- ***Thread*** `ComputingThread`: `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 determinado *DAG slice*. Além do mais, cada instância também gerencia o término da computação de seu respectivo *DAG slice*, realizada remotamente por um dado *peer* servidor. Em outras palavras, `ComputingThread` representa o lado cliente do *middleware* e, conseqüentemente, o lado cliente RMI. `ComputingThread` também se responsabiliza em inicializar o processo de recuperação de execução na ocorrência de falta no *peer* servidor (falta 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 falta.

A Tabela 5.1 mostra a relação entre os módulos da arquitetura do middleware (Figura 4.1), identificados na fase de análise de requisitos, e os componentes utilizados para implementá-los, identificados na fase de projeto.

Módulo da arquitetura	**Componente que o implementa**
Tolerância a Faltas	Classe `Server` *Thread* `ComputingThread` Interface `Compute` Interface `Failure`
Escalonamento	Classe `Scheduler`
Coordenação de Execução	Classe `Scheduler`
Gerência da Rede *Overlay*	Classe `Core` *Thread* `ListeningThread`
Comunicação P2P	*Thread* `ComputingThread`

Tabela 5.1: Relação entre módulos e componentes que os implementam

Após a descrição individual de cada componente utilizado na implementação do *middleware*, a Figura 5.5 mostra como estes componentes interagem entre si ao longo do tempo. Inicialmente, a Figura 5.5 apresenta um diagrama de seqüência de eventos UML para um caso otimista, onde não se verifica nenhuma espécie de falta nos *peers* participantes da computação. Além do mais, utiliza-se um DAG particionado em 3 *DAG slices*, sendo um *DAG slice* pai (sob a responsabilidade do *DAG owner*) e dois *DAG slices* filhos, DAG1 e DAG2.

O diagrama da figura apresenta a seqüência de atividades envolvidas na computação de um DAG. Primeiramente, a instância da classe `Core` inicializa e configura ambos os ambientes RMI e JXTA **(1)**. Logo em seguida, `Core` cria uma instância da classe `Server` **(2)**, a qual dá início ao serviço de descoberta de *peers* através da inicialização de `ListeningThread` **(3)**. A partir deste momento o usuário (*DAG owner*, no caso) já pode submeter o DAG para computação no *middleware* **(4)**. Uma instância da classe *Scheduler* é criada para gerenciar e escalonar os componentes do DAG **(5)**. Visto que o DAG de entrada possui dois *DAG slices* filhos, a instância de `Scheduler` solicita a `ListeningThread` dois *peers* capazes de computar tais *DAG slices* **(6)**. `ListeningThread` por sua vez retorna para `Scheduler` Peer X e Peer Y para a computação de DAG1 e DAG2 respectivamente. Logo em seguida, duas instâncias da *thread* `ComputingThread` são criadas para gerenciar as execuções de DAG1 e DAG2 **(7)**. Pelo diagrama da Figura 5.5 é possível notar a concorrência e independência das instâncias de `ComputingThread`, que distribuem DAG1 e DAG2 para Peer X e Peer Y e aguardam a conclusão de suas execuções. Após a junção dos resultados de DAG1 e DAG2 e a possível execução de outras tarefas por *DAG owner* **(8)**, a computação do DAG está completa e os resultados são apresentados ao usuário **(9)**.

Já a Figura 5.6 apresenta a seqüência de atividades realizadas em um cenário onde ocorre uma falta. Neste caso, é apresentado o caso da Figura 4.5, onde um *peer* inter-

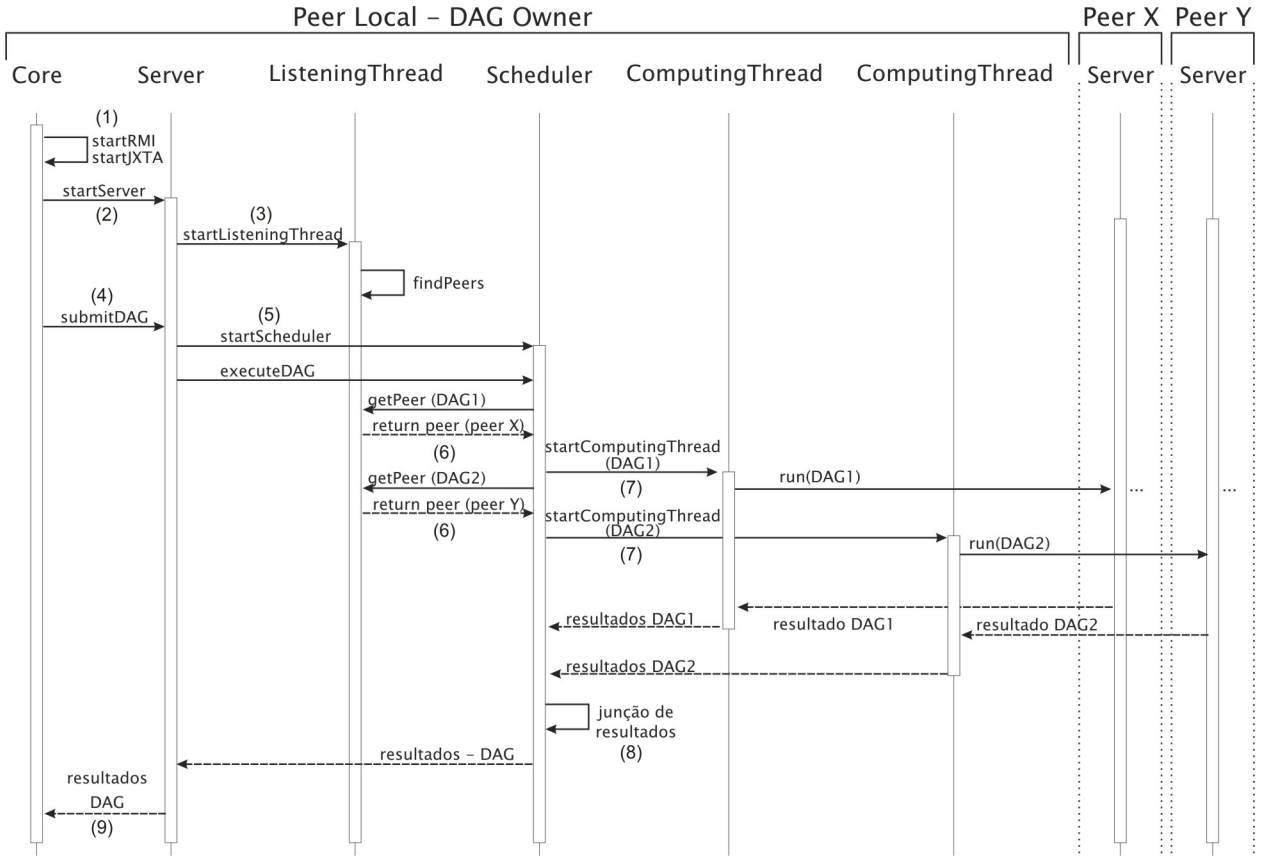


Figura 5.5: Diagrama de seqüência de eventos UML em um cenário sem faltas

mediário na árvore de execução falha. Até o momento da falta, a sequência de eventos é semelhante ao caso anterior, com a distribuição e computação remota dos *DAG slices*, retorno de resultados, etc. Peer B, que representa um *peer* intermediário na árvore de execução e que possui o *DAG slice* DAG 2 para ser computado, distribui os *DAG slices* 4 e 5 para os *peers* D e E respectivamente **(1)**. Quando Peer B falha **(2)**, Peer D e Peer E detectam que Peer B não mais pertence à rede (através de *leasing*) e notificam o *peer* pai de Peer B, no caso o próprio *DAG owner* **(3)**. Após a computação dos respectivos *DAG slices* Peer D e Peer E enviam seus resultados para o *DAG owner* **(4)**, que realiza a marcação dos DAG slices computados com sucesso **(5)** e reescalona o restante de DAG 2 **(6)**, enviando-o para execução em um outro *peer*.

5.5 Construção, Particionamento e População do DAG

Como apresentado na Seção 4.3, o *middleware* recebe como entrada um *workflow* modelado como DAG e composto por um conjunto de tarefas dependentes. Este DAG por sua

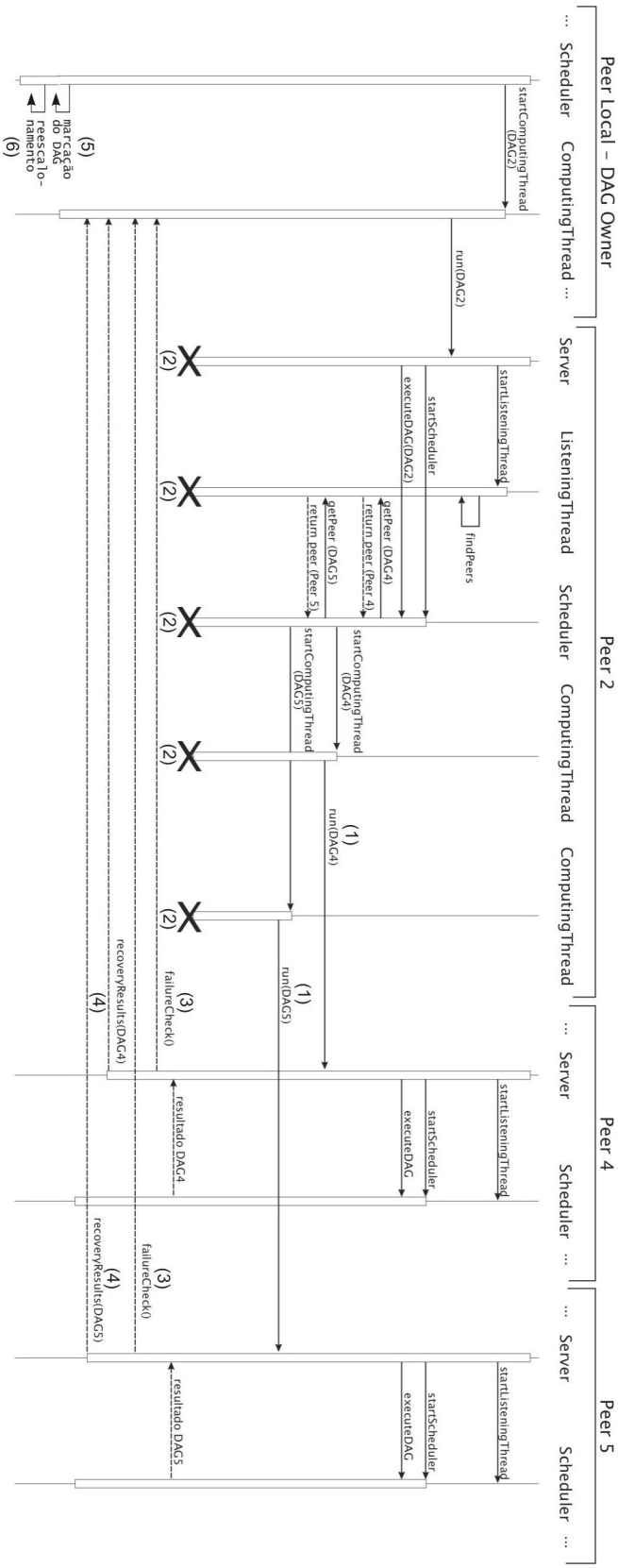


Figura 5.6: Sequência de eventos com a ocorrência de falta

vez pode ser particionado em *DAG slices*. Para a construção, particionamento e população do DAG com tarefas correspondentes, o *middleware* disponibiliza um pequeno mas expressivo conjunto de classes que especificam a estrutura de DAGs e tarefas suportadas. A Figura 5.7 apresenta o diagrama de classes UML deste pequeno *framework* para a construção de DAGs.

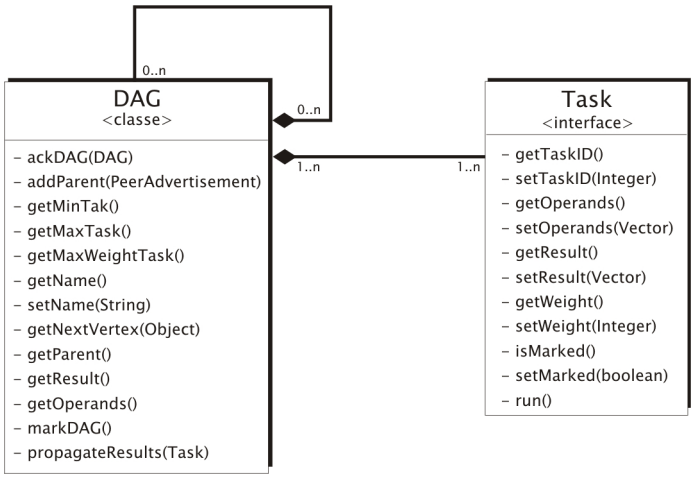


Figura 5.7: Estruturas de criação e composição de DAGs

Como mostra esta figura, um `DAG` pode ser composto por outros DAGs e tarefas, característica esta que permite o aninhamento de *DAG slices* na composição do DAG. A interface `Task` especifica o conjunto de operações que as tarefas devem possuir para que seja possível suas execuções no *middleware*. Logo, o usuário que deseja criar um DAG necessita primeiramente construir as classes concretas que implementam a interface `Task`, provendo a implementação da tarefa a ser computada. De posse de todas as classes concretas das tarefas, o usuário pode então dispô-las no DAG, agrupando-as convenientemente em *DAG slices* e estabelecendo as dependências entre tarefas e/ou *DAG slices*. Cabe também ao usuário responsável pelo DAG a correta conexão entre tarefas e *DAG slices*. Isto quer dizer que o usuário é o encarregado em modelar e prover os tipos de dados corretos para as tarefas posteriores, ou seja, estabelecer a correspondência entre os resultados de uma tarefa e os operandos das tarefas e/ou *DAG slices* sucessores. O não atendimento de tais exigências pode ocasionar erros em tempo de execução, visto que o *middleware* assume que tarefas e *DAG slices* estão conectados corretamente, com os respectivos tipos de dados sendo coerentemente transferidos entre tarefas e/ou *DAG slices*.

Para modelar o DAG como um objeto Java, utilizou-se a biblioteca JGraphT [1], que oferece um conjunto de classes responsáveis por diversas operações como criação e composição de grafos. Através da biblioteca JGraphT, o DAG é modelado como um grafo orientado, onde os nós podem conter objetos que implementam a interface Task ou

instâncias de DAG. Tal biblioteca também auxilia na determinação dos nós sucessores de um dado nó, que no *middleware* possibilita a propagação de resultados para tarefas e *DAG slices* posteriores correspondentes. Outra importante funcionalidade provida pela biblioteca JGraphT é a determinação do grau de entrada e saída de um nó, ou seja, a quantidade de arestas que chegam e saem de um nó, respectivamente. Esta funcionalidade é utilizada no *middleware* com o objetivo de determinar a quantidade de tarefas e/ou *DAG slices* predecessores de um determinado nó - para iniciar sua execução, uma tarefa necessita ter um conjunto de operandos de tamanho igual ao grau de entrada do nó que representa a tarefa.

5.6 Resultados Experimentais

Visto que a maioria das soluções de grades computacionais e aplicações P2P não consideram a interdependência de tarefas, para avaliar a performance do *middleware* apresentado no presente trabalho, nesta seção, 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 4.2, entretanto, cada nó do grafo é populado com a 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 5.8 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 5.8 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 quantidade de processamento maior, visto que o DAG é composto por tarefas ***CPU intensive***. Como resultado, os custos de comunicação tenderam a ser minimizados,

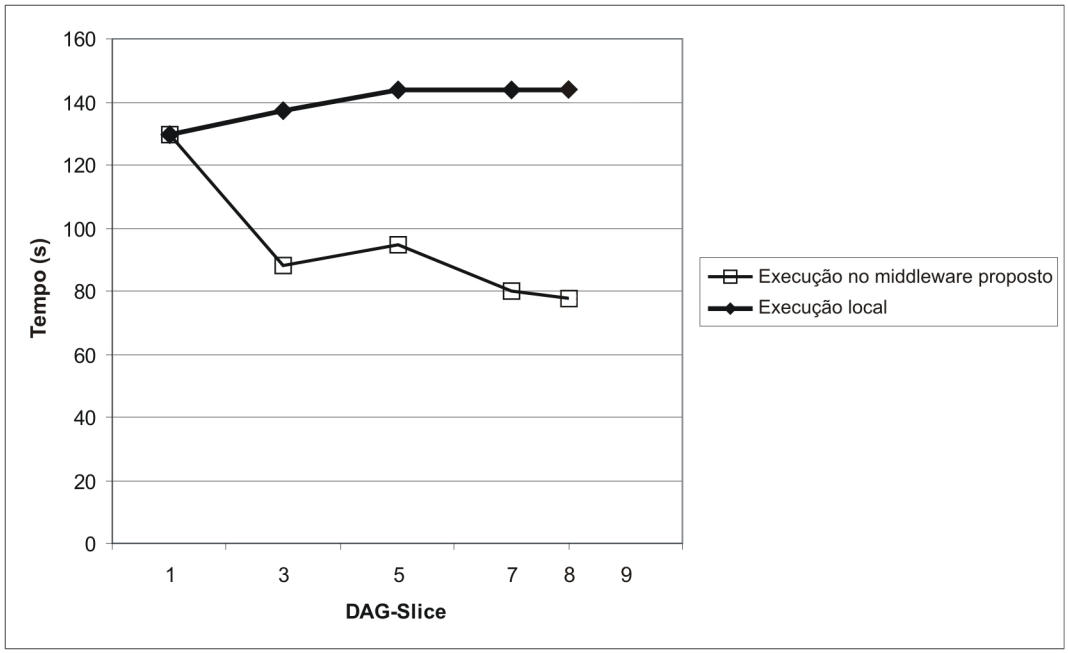


Figura 5.8: Avaliação da execução local e no *middleware* proposto

assim como o tempo total de execução. Apesar deste resultado particular, a Figura 5.8 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 falta (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 a falta. Em ambas as versões do *middleware*, avalia-se novamente o tempo total de execução do DAG, considerando

diferentes números de *peers* iniciais. A Figura 5.9 mostra os resultados obtidos.

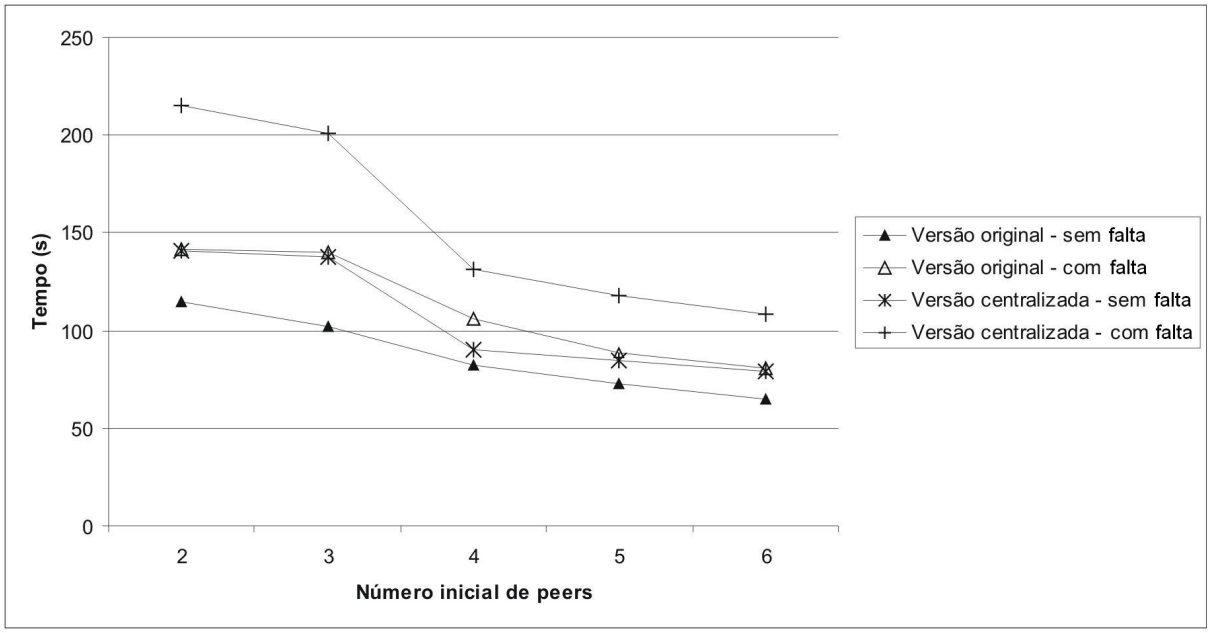


Figura 5.9: Avaliação da intermitência e efeitos de faltas

Tanto na versão centralizada quando na versão original do *middleware*, a Figura 5.9 mostra um grande *overhead* nas execuções com 2 e 3 *peers* iniciais. Estes *overheads* são justificados pela falta 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 falta, minimizando seus efeitos no tempo total de execução do DAG. A Figura 5.9 também mostra que os resultados da versão original com faltas são bastante próximos dos resultados da versão centralizada sem faltas, o que indica a boa performance dos mecanismos de tolerância a faltas e recuperação de execução do *middleware*.

Capítulo 6

Conclusão

As redes *Peer-to-Peer* (P2P) surgiram como fenômenos sociais e tecnológicos significantes na última década. 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. Um dos problemas mais importantes e desafiadores em grades computacionais, e mais recentemente em sistemas P2P, é a execução e gerenciamento distribuído de *workflows*. Para a solução deste problema, a abordagem P2P desponta como uma alternativa viável, na qual 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 *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.

A utilização da especificação JXTA no desenvolvimento da infra-estrutura P2P possibilitou ao *middleware* ultrapassar as barreiras impostas pela heterogeneidade de plataformas

e redes, tornando possível a inclusão e utilização de *peers* em quaisquer ambientes computacionais. Entretanto, ao longo do desenvolvimento do *middleware* observou-se que, apesar da grande quantidade de mecanismos, a especificação JXTA carece de algumas funcionalidades simples como a serialização / deserialização de objetos. Em adição, a linha de aprendizado da especificação JXTA é bastante íngreme e a pouca documentação também dificulta o desenvolvimento.

As funcionalidades providas pelo RMI também foram de fundamental importância para a implementação do *middleware*, as quais auxiliaram o desenvolvimento dos mecanismos de comunicação P2P e de tolerância a faltas de forma elegante e eficiente. O *download* automático de código provido pelo RMI facilitou bastante a transferência de objetos ao longo da rede e, juntamente com os mecanismos de *leasing*, estas duas funcionalidades foram consideradas ao longo do desenvolvimento do *middleware* os principais atrativos de RMI.

Os resultados obtidos mostraram a eficiência do *middleware* na execução distribuída de *workflows*, onde o *middleware* conseguiu um tempo de execução 45% menor que o da execução realizada localmente. A realização dos experimentos também sugeriu a eficácia do mecanismo de recuperação de execução, que amortizou os impactos provocados pela intermitência do ambiente P2P na execução do *workflow*.

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 Bibliográficas

[1] Jgrapht project - a free java graph library. http://jgrapht.sourceforge.net/, 2005.

[2] Emule project. http://www.emule-project.net/, 2007.

[3] Napster project. http://free.napster.com/, 2007.

[4] Peermi project. https://peermi.dev.java.net/, 2007.

[5] Ilkay Altintas, Bertram Ludaescher, Scott Klasky, and Mladen A. Vouk. Introduction to scientific workflow management and the kepler system. In *SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, pages 1039–1065, Nova York, NY, Estados Unidos, 2006. ACM.

[6] Luiz F. Bittencourt. Uma heurística de agrupamento de caminhos para escalonamento de tarefas em grades computacionais. Master's thesis, Universidade Estadual de Campinas - Unicamp, 2006.

[7] Daniel Brookshier, Darren Govoni, Navaneeth Krishnan, and Juan Carlos Soto. *JXTA: Java P2P Programming*. Sams, Indianapolis, IN, Estados Unidos, 2002.

[8] J. Cao, SA Jarvis, S. Saini, and GR Nudd. 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, Los Alamitos, CA, Estados Unidos, 2003.

[9] David Churches, Gabor Gombas, Andrew Harrison, Jason Maassen, Craig Robinson, Matthew Shields, Ian Taylor, and Ian Wang. Programming scientific and distributed workflow with triana services: Research articles. *Concurrency and Computation: Practice and Experience*, 18(10):1021–1037, 2006.

[10] F. R. L. Cicerre, E. R. M. Madeira, and L. E. Buzato. Structured process execution middleware for grid computing. *Concurr. Comput. : Pract. Exper.*, 18(6):581–594, 2006.

[11] L.B. Costa, L. Feitosa, E. Araújo, G. Mendes, W. Cirne, and D. Fireman. Mygrid: A complete solution for running bag-of-tasks applications. In *22nd Brazilian Symposium on Computer Networks - III Special Tools Session, Gramado, RS, Brasil*, pages 925–932, 2004.

[12] T. Fahringer, J. Qin, and S. Hainzer. 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, Estados Unidos, 2005. IEEE Computer Society.

[13] I. T. Foster and A. Iamnitchi. 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, Estados Unidos, 2003.

[14] Yolanda Gil, Pedro A. González-Calero, and Ewa Deelman. 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, Estados Unidos, 2007. ACM Press.

[15] William Grosso. *Java RMI*. O'Reilly & Associates, Inc., Sebastopol, CA, Estados Unidos, 2001.

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

[17] A. Iamnitchi, I. Foster, and D. Nurmi. A peer-to-peer approach to resource location in grid environments. In *Symp. on High Performance Distributed Computing*, Norwell, MA, Estados Unidos, Agosto 2002.

[18] Lichun Ji and Ralph Deters. Coordination & enterprise wide p2p computing. In *SCC '05: Proceedings of the 2005 IEEE International Conference on Services Computing*, pages 141–148, Washington, DC, Estados Unidos, 2005. IEEE Computer Society.

[19] Keong Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim. A survey and comparison of peer-to-peer overlay network schemes. *Communications Surveys & Tutorials, IEEE*, 7(2):72–93, 2005.

[20] P. Maheshwari and J. Ouyang. Supporting fault-tolerance in heterogeneous distributed applications. In *HCW '97: Proceedings of the 6th Heterogeneous Computing*

Workshop (HCW '97), pages 195 – 207, Washington, DC, Estados Unidos, 1997. IEEE Computer Society.

[21] Tom Oinn, Mark Greenwood, Matthew Addis, Nedim M. Alpdemir, Justin Ferris, Kevin Glover, Carole Goble, Antoon Goderis, Duncan Hull, Darren Marvin, Peter Li, Phillip Lord, Matthew R. Pocock, Martin Senger, Robert Stevens, Anil Wipat, and Chris Wroe. Taverna: lessons in creating a workflow environment for the life sciences: Research articles. *Concurrency and Computation: Practice and Experience*, 18(10):1067–1100, Agosto 2006.

[22] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A scalable content-addressable network. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 161–172, Nova York, NY, Estados Unidos, 2001. ACM Press.

[23] Joan Esteve Riasol and Fatos Xhafa. 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, Nova York, NY, Estados Unidos, 2006. ACM.

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

[25] Michael Schneider, Markus Aleksy, Martin Schader, and Makoto Takizawa. Leasing variants in distributed systems. *First International Conference on Complex, Intelligent and Software Intensive Systems, 2007. CISIS 2007.*, pages 68–73, 2007.

[26] R. Schollmeier. A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications. In *Proceedings of the First International Conference on Peer-to-Peer Computing (P2P'01)*, pages 101–102, Washington, DC, Estados Unidos, 2001. IEEE Computer Society.

[27] Jean-Marc Seigneur, Gregory Biegel, and Christian Damsgaard Jensen. P2p with jxta-java pipes. In *PPPJ '03: Proceedings of the 2nd international conference on Principles and practice of programming in Java*, pages 207–212, Nova York, NY, Estados Unidos, 2003. Computer Science Press, Inc.

[28] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking*, 11(1):17–32, 2003.

[29] Mallikarjun Tatipamula and Bhumip Khasnabish. *Multimedia Communications Networks: Technologies and Services*. Artech House, 1998.

[30] Prem Uppuluri, Narendranadh Jabisetti, Uday Joshi, and Yugyung Lee. P2p grid: Service oriented framework for distributed resource management. In *SCC '05: Proceedings of the 2005 IEEE International Conference on Services Computing*, pages 347–350, Washington, DC, Estados Unidos, 2005. IEEE Computer Society.

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

[32] B. Y. Zhao, Ling Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: a resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, 2004.

[33] Kai Zheng, Xueli Yu, and Ruiping Niu. Research on resource deployment of grid workflow. In *SKG '05: Proceedings of the First International Conference on Semantics, Knowledge, and Grid*, pages 95–97, Washington, DC, Estados Unidos, 2005.